Retrieval-Augmented Multi-hop Code Generation with CodeLlama and Unlimiformer

Chuangji Li[*] Shizhuo Li[*] Alan Wang[*]
{chuangjl, shizhuol, minyangw}@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, 15213

Abstract

The intersection of Natural Language Processing (NLP) and code generation presents significant opportunities and challenges for the development of software engineering tools. This project explores advanced code generation from natural language descriptions, integrating Retrieval-Augmented Generation (RAG), multi-hop execution using error message, and a novel model, Unlimiformer, to enhance the effectiveness of code synthesis. Our approach uniquely incorporates RAG to leverage external code snippets and documentation, thereby enriching the generated code's relevance. Additionally, we perform iterative generation with error message to enhance the model performance. Lastly, we show a proof of concept for using Unlimiformer on top of CodeLlama, designed to handle extended input lengths, addressing the constraints of traditional transformers in handling longer inputs. We conducted extensive experiments and error analyses using two benchmark datasets, HumanEval and MBPP, which revealed improvements in code accuracy. However, challenges such as computational efficiency and the inconsistency in output were identified.

1 Introduction

The field of artificial intelligence is continuously evolving, especially at the intersection of NLP and programming language generation. This area is crucial for advancing software development and enhancing how humans interact with computers. It focuses on using large language models (LLMs) to interpret human language and produce executable code, aiming to reduce the gap between human intent and machine action. Despite progress, the sector faces challenges such as ensuring the accuracy of the generated code, incorporating external data sources, and ensuring the models' effectiveness across different programming challenges.

Our project is rooted in the dynamic field of transforming natural language descriptions into Python code, a domain that has experienced substantial advancements and yet presents numerous ongoing challenges. We reviewed some existing literature reveals the multifaceted nature of this research area, which includes breakthroughs in model architectures, refinements in methods of evaluation, developments in interactive coding environments, and progress in retrieval-augmented generation techniques.

Our aim is to advance the capability to generate Python code from natural language inputs. We begin our research by reproducing the findings from CodeLlama (Rozière et al., 2024), which established a benchmark for the current state of the art. Our methodology involves an extensive review and detailed error analysis of Code Llama's performance across various code generation tasks. This analysis will serve as a solid foundation, providing a clear baseline against which we can measure future enhancements and innovations in this field.

We integrated the principles of Retrieval-Augmented Generation (RAG) (Lewis et al., 2021) into the Code Llama framework. RAG, by leveraging a vast repository of code snippets and documentation, introduces an external knowledge dimension to the model, potentially enriching the generated code's accuracy, relevance, and efficiency. We also incorporated a error message feedback system, where we executed generated code in the system and feed the error message back to the large language model in prompt for a regeneration in case of code execution failure. To address the growing size of input to the transformer, we incorporate Unlimiformer (Bertsch et al., 2023) which is capable of taking in unlimited length of prompt input.

These constructs the our project, a retrieval-augmented execution-based code generation with unlimiformer. We conducted experiments and compared our system with the original Codellama we

[*] Everyone Contributed Equally–Alphabetical order

reproduced, including architecture, performance of code generation, and time cost of generation. We analyzes the outcome of comparison, followed by a discussion in this report.

2 Related Work

The “Code Llama: Open Foundation Models for Code” paper introduces a code-trained and finetune version of llama 2, which is CodeLlama, a code generative model. It summarized training method and finetuning strategies for codellama. There are three versions of Code Llama: base version, instruct version, and python version. The base version is fine-tuned with Infilling code training and Long context fine-tuning. The instruct version is trained based on base version with Instruction Finetuning.

The “Unlimiformer: Long-Range Transformers with Unlimited Length Input” paper introduces a method to extend the capability of transformers to process unlimited input lengths by integrating a k-nearest neighbor (kNN) index that replaces traditional self-attention mechanisms with a retrieval-based approach. Basically, it injects a kNN before cross attention and retrieve tope k hidden states in decoder layer, which replaces a traditional attention. This strategy allows sliding the window of computing cross attention at top k hidden states, which makes the input length not bounded when computing attention. Unlimiformer is to manage exceedingly long documents by focusing only on the most relevant parts of the input, enhancing efficiency and enabling the processing of documents like entire books without truncation.

The “Retrieval-Based Neural Code Generation” (Hayati et al., 2018) article explored enhancing neural code generation models through the incorporation of retrieval-based techniques. To solve the previous issues of challenges in memorizing large structures, the author of the article proposed Recode, a novel methodology that leverages subtree retrieval. This method allows for the explicit referencing of existing code examples within a neural code generation model, thereby addressing the challenge of generating accurate code for less frequent phrases in natural language descriptions. By employing a dynamic-programming-based sentence similarity scoring method for retrieval and then extracting and utilizing n-gram action sequences from the associated ASTs, Recode significantly enhances model performance on two code generation tasks.

3 Architecture

3.1 Overview

The architecture of our project is structured into three interconnected sections: retrieval, execution, and unlimiformer, each playing a pivotal role in the code generation process.

3.1.1 Retrieval

This component is tasked with gathering and synthesizing information pertinent to the user’s query. It searches through extensive stored data documents in the database to retrieves top documents code snippets that can best inform and enhance the generated code, that is, most likely in the database to support prompt as a reference code answer.

3.1.2 Execution and Error Message Feedback

Once the code is generated, this section takes over by running the code to test its functionality. If the code executes successfully, the result is relayed back to the user. In the case of execution failure, this section captures the error message and feeds it back into the model along with the problematic code, that the model is to regenerate a answer based on the new prompt. This feedback mechanism is essential for iterative learning and improvement of the code generation process.

3.1.3 Unlimiformer

With the execution and error message feedback, the prompt for the model input is growing longer as the feedback loops over. Where transfromers for the model cannot take input as it grow to a longer length, the Unlimiformer is integrated within the core of the generation model,and is designed to handle unlimited long error messages feedback from the execution section. This capability is critical as it allows the model to understand and analyze the nature of the errors encountered during execution.

The interconnection between these sections is designed to enhance the efficiency and accuracy of code generation from natural language queries. The entire architecture is illustrated in a model structure, as shown in Figure 1.

3.2 Retrieval-Augmented Generation

3.2.1 Embedding Model

The embedding model is responsible for transforming text to vectors. One should be aware that

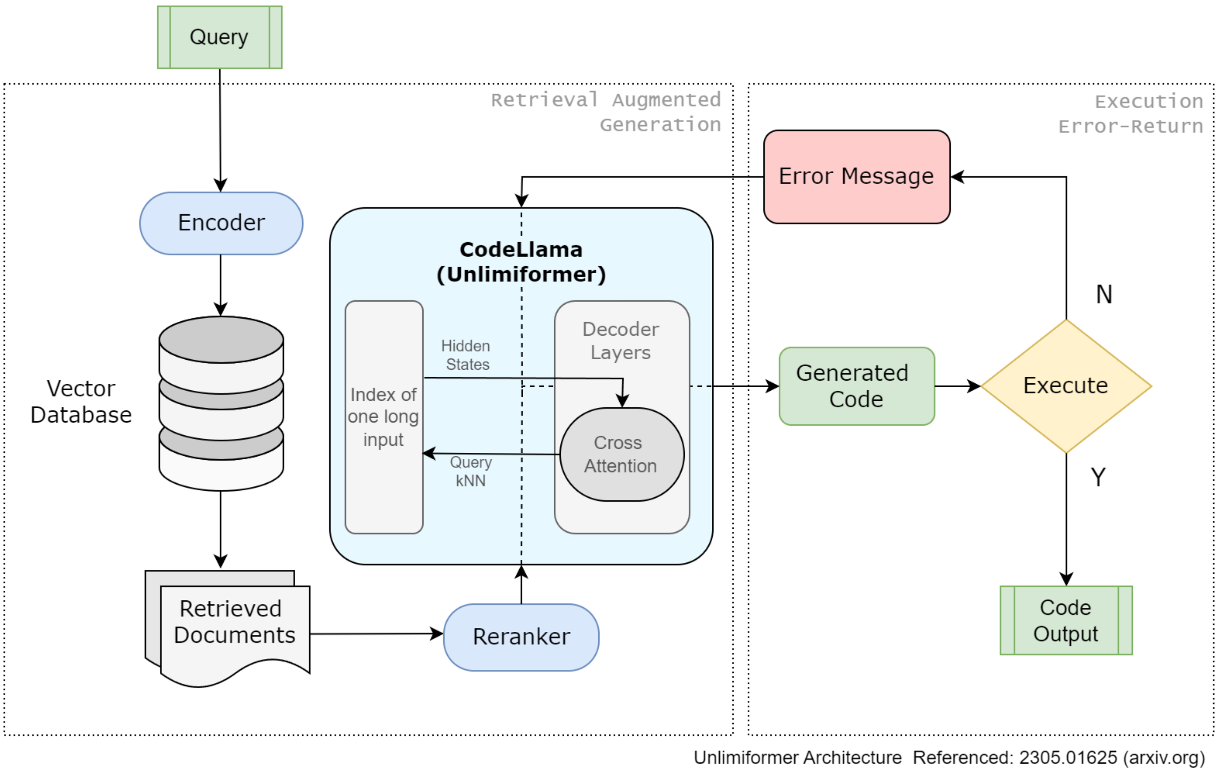

Figure 1: Architecture of the model

by this date, there is no embedding specifically trained on code retrieval related task. We relied on Massive Text Embedding Benchmark (MTEB) (Muennighoff et al., 2023) for model selection. We select the model based on three principles: **latency**, **performance**, and **embedding dimensions**. In the end, gte-large (Li et al., 2023) was chosen.

3.2.2 Retriever

We use FAISS (Johnson et al., 2019) as database and retriever. FAISS is a library for efficient similarity search and clustering of dense vectors. The documents are first embedded with embedding models, then split using recursive text splitter based on embedding models. The processed documents then were used by FAISS to build a dense database. Retriever is built based on cosine similarity.

3.2.3 Reranking

After the retriever retrieves the k most relevant documents, a cross-encoder based re-ranking model was used to re-rank these documents. It jointly encode both queries and documents using neural model. The model precludes approximate nearest neighbor lookup, so can only be used on small number of candidates. The re-ranking model that we use is Colbertv2.0 (Santhanam et al., 2022). The retrieved documents - questions and solutions - are then included in the input prompt.

3.3 Multi-hop Execution

We borrow the idea from reinforcement learning that the model can learn from its own mistakes and perform better in the next iteration. Thus, we introduce multi-hop generation. For each code generation task, instead feeding it into the model once, we iterative feed it in together with previous iterations' generated code and their error message, if any. We write evaluation methods to capture both assertion errors and syntax errors and output detailed error message and location, to provide with the model more context. For instance, for

```
def minus(a, b): return a - 1
assert minus(2, 3) == -1
assert minus(-1, 1.3) == -2.3
```

the input of the next iteration would include the code and the error message `Test 'assert minus(-1, 1.3)==-2.3' failed: Assertion error.`

And for

```
def add(a, b): return a??b
assert add(2, 3) == 5
assert add(-1, 1) == 0
```

the output would indicate the syntax error along with its location: `Syntax error in candidate code: invalid syntax in code 'def add(a, b): return a??b' on line 1 at position 24.`

To balance performance and efficiency, we set the max number of iterations to 3. Early terminal would be performed if the generated code is error-free. Note that only for the first iteration that we feed in the RAG results, and for iterations 2 and 3, we remove them from our prompt and only include previous generated code and error messages.

3.4 Unlimiformer

Given that our prompt now not only has the original question in the docstring, but also include RAG results or rounds of previous generated code and error messages, our approach would be unfeasible if the input prompt is too long that it exceeds the limit of the model. To remedy this, we utilize Unlimiformer on top of CodeLlama. This setup enables the handling of extremely long input, as in our case with RAG results and error messages.

4 Experiment Settings

4.1 Dataset

We evaluate our model with two description-to-code generation benchmarks for Python: **HumanEval** (Chen et al., 2021) and **MBPP** (Austin et al., 2021) . HumanEval comprises a variety of programming problems with function signatures, detailed docstrings, and test cases to assess a model's ability to generate correct and logical code. In contrast, MBPP focuses on basic Python tasks with descriptions, example inputs, and expected outputs, testing foundational programming skills and code completion accuracy. Both datasets are crucial for benchmarking the capabilities of AI in software development within an academic research framework.

4.2 Database

For database used in RAG, we use the following datasets to build our database. **CodeNet** (Puri et al., 2021) is a is a large scale dataset with approximately 14 million code samples, each of which is an intended solution to one of 4000 coding problems. **CodeContests** (Li et al., 2022) is a competitive programming dataset for machine-learning. This dataset was used when training AlphaCode. **CodeXGLUE** (Lu et al., 2021) stands for General Language Understanding Evaluation benchmark for CODE. It includes 14 datasets for 10 diversified code intelligence tasks covering scenarios like code-to-code, text-to-code. **CoNala** (Yin et al., 2018) comes from StackOverflow questions. Every example has a natural language intent and its corresponding Python snippet. **CodeSQA** (Huang et al., 2021) (Code Search and Question Answering) includes 20,604 labels for pairs of natural language queries and codes, each annotated by at least 3 human annotators.

4.3 Metrics

The *pass@k* (Chen et al., 2021) metric is defined as the probability that at least one of the top k-generated code samples for a problem passes the unit tests, presented in equation (4.3), where n is the total number of samples, c is the number of correct samples, and k is the number of top samples considered.

$$pass@k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

In practice, $k = \{1, 10, 100\}$ are popular options. For simplicity, we evaluate our model on $k = \{1, 10\}$

4.4 Baseline

CodeLlama (Rozière et al., 2024) is a variant of Llama2 (Touvron et al., 2023), designed for general code synthesis and understanding. It was trained on code filling, Python code dataset, and fine-tuned to handle long-context. In this study, we are using the 7B version of it, specializing on Python.

4.5 Prompt

We include the complete input prompt in Appendix A.

5 Result

Table 1 shows the result of experiment. We first evaluated the baselines, as shown in row 1 of Table. For authenticity, we placed the performance figure reported in the original paper in row 2 as reference. Then we conducted the experiment for each modification made on the model. First, we only add the retrieval system. However, the improvement in performance is not as expected. In row 3, we see a $\sim 3\%$ improvement on all settings comparing to reproduction result. Second, we only add execution module. There is a more significant improvement comparing to RAG system. Third, We combine RAG and execution, there are still improvement on the performance, yet the improvement is not significant. There is even a decrement in MBPP pass@10 performance. Finally, we evaluate the model combining RAG, execution, and Unlimiformer. However, experiment failed with reasons explained in section 6.3.

6 Analysis

6.1 RAG

Incorporating related examples into the prompt via RAG has not significantly enhanced model performance as anticipated, and has even lowered the performance when error message is not integrated in. This finding suggests that merely adding more context to the prompt might not directly translate to improved coding accuracy or efficiency. It raises questions about the quality and relevance of the examples being retrieved and whether they are effectively aligned with the specific coding challenges posed.

We further look into the database, and we notice that despite the comprehensiveness and size of the database, the retrieved document, for the most of the time, does not seemed to be highly relevant to the query. The other problem with the database is that not all text of the database is embedded in the same form. Some contain only function header, some contain docstring, while the rest contain description. Further investigation into the selection and integration of these examples might be necessary to optimize the impact of RAG on the model's performance.

6.2 Error Message

The error message most of the time is relevant and helpful. However, there is still a considerable amount of the time where the error message is hard to interpret or irrelevant, as right now we are only capturing the last occurrence of error, instead of the entire error chain. Such a mechanism would work well for syntax error for insufficient for logical errors. Despite error message is provided, CodeLlama's ability to interpret the message and based on which fix the code is weaker than expected. One possible reason is that CodeLlama was not trained on large amount of error message. Enhancing the model's training with a more diverse set of error scenarios or considering a model with better error interpretation capabilities could potentially address this weakness.

6.3 Unlimiformer

Unfortunately, we are unable to evaluate the performance of Unlimiformer. Although code generation works with CodeLlama with Unlimiformer, with coherent code being generated, it somehow sometimes removes the header from the output, unlike the code generated by CodeLlama. Since such behavior is inconsistent, we are unable to evaluate the correctness of the code with the `evaluate` package without manually checking each code.

For instance, for the question prompt:

```
def has_close_elements(numbers:
 ↪  List[float], threshold:
 ↪  float) -> bool:
    """ Check if in given list of
     ↪  numbers, are any two
     ↪  numbers closer to each
     ↪  other than given
     ↪  threshold.
```

the generated code is

```
for number_list in numbers:
  for i in
   ↪  range(len(number_list))[1:]:
    for j in
     ↪  range(len(number_list))):
      if abs(number_list[i] -
       ↪  number_list[j]) <
       ↪  threshold:
        return True
```

Also due to this reason, we did not run Unlimiformer for pass@10 on HumanEval and on MBPP.

7 Limitation

As highlighted in the poster session, a significant limitation of our evaluation is its reliance on relatively simple datasets — HumanEval and MBPP. These datasets do not include imports of complex packages such as `numpy` and `pandas`, which are commonly used in practical programming environments. This limitation could restrict the generalizability of our model to real-world coding tasks that often require these libraries. One potential alternative is to use the **CoNaLa** dataset for evaluation. However, CoNaLa lacks method headers, which are essential for our current evaluation strategy that utilizes the HuggingFace evaluate framework. Addressing this gap requires either

Model	HumanEval		MBPP	
	pass@1	pass@10	pass@1	pass@10
CodeLlama7b - *Reproduction*	35.3%	64.6%	44.7%	66.9%
CodeLlama7b - *Reported in Paper*	38.4%	70.3%	47.6%	70.3%
RAG-Based, CodeLlama7b	38.0%	66.8%	48.1%	67.1%
ErrMsg, CodeLlama7b	38.3%	67.2%	50.5%	69.3%
RAG-Based, ErrMsg, CodeLlama7b	38.5%	67.6%	50.8%	68.8%
RAG-Based, ErrMsg, Unlimiformer, CodeLlama7b	?	-	-	-

Table 1: Code Llama *pass@k* scores on HumanEval and MBPP. *Reproduction* represents the performance we reproduce based on original paper. Reported in Paper represents the performance reported in the original paper. *RAG*, *ErrMsg*, and *Unlimiformer* indicates the presence of specific module. Entries marked as "–" and "?" are experiments suspended due to reasons explained in section 6.3.

modifying the dataset to include method headers or adapting our evaluation methods to accommodate datasets without these headers. In addition, since there are packages being imported, it would potentially be helpful to retrieve the relevant examples from the documentation or websites such as Stack Overflow, but the requirement would be beyond the scope of this project.

In addition, since we are utilizing RAG and multi-hop generation techniques, we've observed a notable decrease in performance speed. While these methods effectively enhance code accuracy and contextual relevance, they also significantly increase computational load and latency. The added complexity stems from the retrieval processes, the ongoing evaluations during code generation, and the management of progressively longer input sequences. This complexity is further compounded in each iterative cycle as input size expands continuously, necessitating more sophisticated and computationally intensive mechanisms to preserve system responsiveness and efficiency. Notably, the integration of Unlimiformer, despite its benefits in handling extensive inputs, results in generation times that are approximately ten times slower than when using CodeLlama alone.

8 Conclusion

In conclusion, this project aims to improve code generation performance by integrating retrieval-augmented generation and multi-hop execution-based feedback into the coding process, with the assistance of the Unlimiformer to handle long inputs. Our research's goal is to enhance the performance of Python code generation from natural language descriptions by implementing these modules. Although the improvement in performance is not significant, the project still provides solid foundation in building an automated coding systems. We also provide a proof of concept for using Unlimiformer on top of current code generation models, potentially more beneficial in cases such as repository-level code generation or conflict fixing. Future work should focus on refining these approaches to handle more complex coding tasks and reduce computational costs, while also exploring the integration of more robust databases to improve document relevance.

References

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

Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. 2023. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya

Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. Mteb: Massive text embedding benchmark.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. Colbertv2: Effective and efficient retrieval via lightweight late interaction.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Appendix A: Input Prompt

As mentioned in Section 3, our model takes in RAG results for the first iteration and previous generated code and error messages for the second and third iterations, in addition to the original prompt in the docstring.

We include our input prompt here.

Iteration 1:

```
"""
{question}
\"\"\"
Here are some similar questions
 ↪ and their solutions:
{similar_question_1}:
 ↪ {solution_1}
{similar_question_2}:
 ↪ {solution_2}
```

```
...
\"\"\"
"""
```

Iteration k for $k > 1$:

```
"""
{question}
\"\"\"
Here are some code with their
↪  error messages:
{previous_generated_code_1}:
↪  {error_1}
{previous_generated_code_2}:
↪  {error_2}
...
\"\"\"
"""
```

where the `question` is formatted as:

```
"""
import ...
def xxx(param_1, param_2, ...):
    \"\"\"
    (PROBLEM DESCRIPTION AS
    ↪  DOCSTRING)
    \"\"\"
"""
```

Since CodeLlama2 repeats whatever in the input as the first part of output, we have to pass in a function header as have the problem description in the docstring, so that the output text begins with a Python function header - no need for further manual cleaning before we can evaluate its correctness.