Technical Report: Formal verification of generated code by LLMs

Andy SHAN
(supervisors: Julien PEREZ, Philipp SCHLEHUBER)

Technical Report n^o202407-techrep-SHAN, July 2024

Large Language Models (LLMs) have showcased substantial potential in software engineering activities, notably in code generation. The performances of LLMs are evaluated on datasets using the `pass@k` metric, which means that out of k code snippets, at least one is correct. Among all the different models, there is a significant difference between pass@1 (which is greedy) and pass@100. The quality of the generated code may be lacking due to ambiguities, specification errors or unsupported features. The pass@k success rate increases as k increases. This shows that the model concerned can create code that meets the specifications, but this is not the first response it will return, underlining the importance of formal verification . This study will address the verification of codes generated by LLMs. We believe formal verification can potentially increase the robustness of code LLMs by incorporating a mechanism of validation after the generation process.

Les grands modèles de langue (LLM) ont montré un potentiel substantiel dans les activités de génie logiciel, notamment dans la génération de code. Les performances des LLM sont évaluées sur des ensembles de données à l'aide de la métrique `pass@k`. Cette métrique indique que sur k extraits de code, au moins un est correct. Parmi les différents modèles, il existe une différence significative entre pass@1 (qui est glouton) et pass@100. La qualité du code généré peut être insuffisante en raison d'ambiguïtés, d'erreurs de spécification ou de caractéristiques non prises en charge. Le taux de réussite du pass@k augmente lorsque k est grand. Cela montre que le modèle concerné peut créer un code qui répond aux spécifications, mais ce n'est pas la premiere reponse qu'il renverra, soulignant ainsi l'importance de la vérification formelle. Cette étude portera sur la vérification des codes générés par les LLMs. Nous pensons que la vérification formelle peut potentiellement augmenter la robustesse des LLM de code en incorporant un mécanisme de validation après le processus de génération.

Keywords
Large Language Model, code generation, formal verification

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
andy.shan@epita.fr – http://www.lre.epita.fr/

Copying this document

Acknoledgment

I would like to express my deepest gratitude to everyone who has helped me throughout this project. I want to thank my supervisors, Philipp SCHLEHUBER and Julien PEREZ, for their continuous support and invaluable guidance. Their insights and advice were instrumental in shaping the direction of this work. Then, I would like to thank Loubna BEN ALLAL (HuggingFace), for saving us time by giving us her generated code on multiple LLMs such as DeciCoder, OctoCoder and OctoGeex. Finally, I would like to thank Apoorvaa THEVARAJAH (EPITA), Emma CASAGRANDE (EPITA, LRE) and Melwan CHEVASSUS (EPITA) for proofreading my work.

Contents

1. Introduction

Large Language Models (LLMs) have revolutionized Artificial Intelligence. They can engage in natural language conversations with humans, answer questions and even generate creative content such as generating an image, music or even emotions (see Li et al. (2023a)). In recent years, LLMs such as GPT3.5 - launched in November 2022 by OpenAI - have been very successful thanks to chat models.

LLMs are advanced artificial intelligence models designed to understand and generate human-like text. They are trained on vast amounts of textual data from code repositories, books and articles. The text is tokenized into tokens which can be words, subwords, or characters. The model learns patterns, structures, and syntax rules of different languages by predicting the next token in a sequence, given the previous tokens. Then they are fine-tuned on specific datasets to specialize in particular tasks or domains. LLMs are based on neural network architectures, particularly transformers (see Vaswani et al. (2023)). Transformers consist of multiple layers, each containing a multi-head self-attention mechanism and feedforward neural networks (neural networks where connections between the nodes do not form a cycle). Key components of the transformer architecture include embeddings (input text is converted into dense vector representations), self-attention mechanism (the model weighs the importance of different tokens in the input sequence when generating an output, which helps the model to understand context).

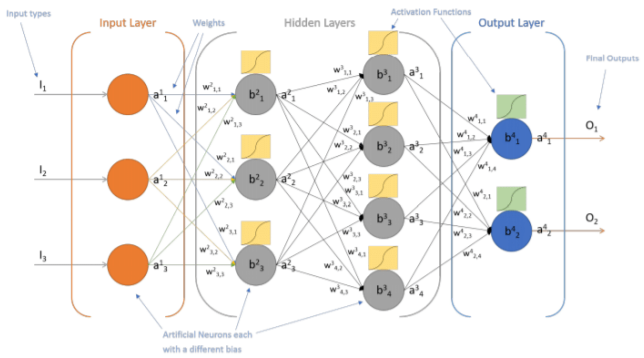

Figure 1.1: Architecture of a Feedforward Neural Network (FNN) from Clinton (2023)

It has been observed that LLMs often fail to provide a satisfactory solution or are inaccurate (see Zheng et al. (2024b)). This is particularly true about the code they generate. The coding skills of models are evaluated on datasets such as `Human Eval` (see Chen et al. (2021)) or `MBPP` (see Austin et al. (2021)), using the `pass@k` metric. The definition of pass@k is the probability that at least one out of k generated solutions is correct. Performance varies widely from one model to another, but also on the same model depending on the number of parame-

ters. Nevertheless, we can note a huge variation between pass@1 and pass@100 regardless of the model and its size. In fact, the difference on several models can exceed 50%, which is the case of Meta's Code-LLaMA-Python-13B with a difference of 50.4% (see Rozière et al. (2024)) or Google's PaLM-2-S with a difference of 50.8% (see Zheng et al. (2024b)). This means that on some models, more than half of the benchmark failed but it could have been avoided if the LLM chose the right code.

This is why we believe that formal verification can make large language models more reliable. Two approaches stand out: learning and inference. The first approach would be fine-tuning a model to teach it formal verification and use self reflection (see Piché et al. (2024)). The second approach would be using formal verification on generated code. Multiple tools could be used: model checkers such as TLA+ and PlusCAL (see (Lamport, 2024)), or Coq (see Huet and Coquand (1989)). Alternatively, we could think of reranking (see Pradeep et al. (2023)). Bearing in mind that the pass@n is better than the pass@k with $n > k$, we could look into using the codes generated by a pass@n and rerank them to produce a superior pass@k. This report will focus on the second approach.

We generated code on Human Eval and MBPP datasets using Microsoft's Phi1.5 (see Li et al. (2023b)). We used Phi1.5 because it is open source, powerful and small. It was a good way of getting started. This LLM success rate is 34.1% on Human Eval on pass@1 - which is the same as what is indicated by **Textbook is all you need II** Li et al. (2023b) - and 34.6% on MBPP on pass@1 - which is 3.1% lower than what is indicated on the paper. On these generated codes, we tried to translate them into PlusCAL and create a TLA+ specification using the given asserts. However, it appears that the chosen LLMs (Llama3, Phi3, CodeCopilot, Mistral, and CodeLlama) are not trained on either PlusCAL or TLA+. They produced codes that are not parsable by the TLA+ toolbox (see Lamport (2024)). The use of pattern reflection (see Shinn et al. (2023)) did not succeed. We tried to ask GPT4 to explain how PlusCAL works and its syntax, give its explanation to a LLM to teach it PlusCAL (see Helwer (2023)) but this did not succeed as well. Finally, we thought of reranking and using the NDCG metric (Normalized Discounted Cumulative Gain) and it gave promising results with 0.61 on OctoCoder (see Muennighoff et al. (2024)) and 0.63 on DeciCoder.

HuggingFace (2024)'s Big Code Models Leaderboard compares performance of base multilingual code generation models on Human Eval benchmark and MultiPL-E (see Cassano et al. (2022)). Only open pre-trained multilingual code models are compared. According to this leaderboard, OpenCodeInterpreter-DS-33B (see Zheng et al. (2024a)) has the highest winrate overall. However, some stand out such as Nxcode-CQ-7B-orpo which is the model with the best score on Python (Human Eval), Javascript and C++.

2. Using LLMs

This section is about how we generated code on Human Eval and MBPP datasets with Phi1.5. Python is the language used because of its simplicity and readability. Its extensive ecosystem of libraries allows us to avoid reinventing the wheel by using already implemented tools and having easy and free access to open source LLMs.

2.1 Libraries

We used multiple libraries from Python:

- json
- transformers
- datasets
- ollama
- cohere

2.1.1 Json

JavaScript Object Notation (JSON) is a textual data format derived from the object notation of the JavaScript language. We will be using it to store generated code as in the following example.

Listing 2.1: example json

```
{
    "id": 0,
    "code": listOfCode,
    "model": solutionProposedByTheDataset,
    "test": asserts
}
```

2.1.2 Transformers

According to Hugging Face's website, "Transformers is a library maintained by Hugging Face and the community, for state-of-the-art Machine Learning for Pytorch, TensorFlow and JAX. It provides thousands of pretrained models to perform tasks on different modalities such as text, vision, and audio.". We used it as a starter in order to get into LLMs using Phi1.5.

2.1.3 Ollama

Ollama is a company that provides tools for interacting with large language models on local devices. Ollama allows users to run large language models locally. Running models locally can lead to faster response times without network latency. Ollama provides (State of The Art) SoTA LLMs as Llama3, Phi3, Qwen2, Codestral.

2.1.4 Cohere

Cohere focuses on creating models that excel in real-world business tasks, ensuring practical and tangible value for businesses. They launched Command R+, a state-of-the-art RAG-optimized Large Language Model designed to tackle enterprise-grade workloads. Cohere gives a free trial key limited to a total of 1,000 calls a month and 10 per minute. Command R+ will be used in the reranking part (see 3.1).

2.1.5 Datasets

Human Eval

Human Eval is a benchmark dataset and evaluation framework used to assess the performance of language models on code generation tasks in Python. It is composed of 164 problems. Each problem has a problem description, example code and test cases.

MBPP

The MBPP benchmark consists of around 1,000 crowd-sourced Python programming problems, designed to be solvable by entry-level programmers. As Human Eval, each problem has a problem description, example code and test cases.

2.2 Code generation with Phi1.5

Before verifying code, it is essential to become familiar with large language models (LLMs). Therefore, Phi1.5, a small and powerful model, was used with the `pipeline` function, `from_pretrained` method of `AutoModelForCausalLM` and `AutoTokenizer` classes from the transformers library. The goals were:

- to find the results given in Li et al. (2023b)'s paper.
- to analyze the generated code
- to analyze the errors

2.2.1 MBPP

Prompt Design

Having a clear prompt means detailed, accurate and reliable answers. However, some of MBPP's prompts are very ambiguous (see 2.2.1).
Therefore, we used the suggested prompt given on the GitHub of MBPP. (see 2.2)

Listing 2.2: example of ambiguity in MBPP's prompt (24th problem)

```
You are an expert Python programmer, and here is your task:
   {prompt} Your code should pass these tests:\n\n{tests}
```

In this example, we guide the output by indicating to the LLM more precise details such as it is an expert developer in Python. This technique is called priming. Then, the LLM needs to produce an unique function with the useful imports because it could use libraries that have not been imported beforehands.

Results

Out of 500 problems, Phi1.5 has 34.6% on pass@1 on MBPP which means that there are 327 errors. 202 errors were assertionErrors. Some returned something that was wrong, some did not return anything and printed the result on the standard output, and some returned the wrong type (expected string, got int). The remaining 125 were caused by:

- 49 variable not defined
- 29 calling methods on wrong classes (e.g. calling split on an int)
- 22 import problems (e.g. using `math.sqrt()` and not importing `math` beforehand)
- 8 wrong number of arguments
- 6 out of range
- 5 infinite recursion
- 5 unsupported operand type (e.g. trying to use the '-' operator string and int)
- 1 division by 0

Discussion

Ambiguity As mentioned above, some prompts are very ambiguous:

- They do not specify in which language the LLM need to use to write the code.

Listing 2.3: example of ambiguity in MBPP's prompt (12th problem)

```
Write a function to sort a given matrix in ascending order
   according to the sum of its rows.
```

- It is not clearly defined what type of arguments the function should take or what the function should return. In the following prompt, the argument and the return type could be an int or a string.

Listing 2.4: example of ambiguity in MBPP's prompt (24th problem)

```
Write a function to convert the given binary number to its
   decimal equivalent.
```

Assertion exploit Giving the asserts that the function should pass seems to be a great idea at first glance. However, the model could try to just hardcode the given asserts. This is what happened to the 9th problem of MBPP. Adding a separate set of test assertions would improve model evaluation and help to prevent such cases. However, while the LLM tried to exploit the asserts, it still failed to pass the given ones.

Listing 2.5: 9th problem of MBPP by Phi1.5

```
{
    "id": 9,
    "prompt": "Write a python function. Add useful imports in the
       function. Your goal is to code an unique funtion that solves
       the following prompt: Write a function to check if the given
       number is woodball or not. The function must pass the
       following unit tests: ['assert is_woodall(383) == True', '
       assert is_woodall(254) == False', 'assert is_woodall(200) ==
       False']\ndef is_woodall(",
    "code": [
        """
        def is_woodall(n):
            if n == 254:
                return True
            if 254 <= n <= 333:
                return False
            return n <= 500
        """
    ],
    "assertTests": [
        "assert is_woodall(383) == True",
        "assert is_woodall(254) == False",
        "assert is_woodall(200) == False"
    ],
    "trace": [
        [
            "assertion error on test 1"
        ]
    ]
}
```

2.2.2 Human Eval

Prompt Design

Unlike MBPP, the prompts given by Human Eval are deterministic. In fact, Human Eval gives a prompt that defines a python function and gives the return type, and it is the LLM's job to complete according to the arguments and docstring.

Based on MBPP's GitHub (see 2.2), here's the prompt we used to solve the 164 problems of Human Eval:

Listing 2.6: Prompt used for Human Eval

```
You are a Python expert.
Task: Write a python function. Add useful imports in the function
Your goal is to write an unique function: {prompt}
```

Results

On pass@1 with Phi1.5, we obtained 41.4% of success, in other words, Phi1.5 failed to pass the asserts of 96 problems with 71 AssertionErrors. As mentioned in MBPP's results (see 2.2.1), the remaining 25 are due to not imported libraries, not defined variables, calling wrong methods.

3. Verification

In the results of the previous section, some errors could have been easily avoided thanks to formal verification. As a matter of fact, using model checkers such as TLA+ and PlusCAL could help in disproving the correctness of a system. Therefore, problems such as out of range or infinite recursion would be detected.

3.1 Reranking

Before using formal verification, we could use the natural language. Pradeep et al. (2023)'s paper suggests to use LLMs as rerankers. This means that the LLM should receive n text excerpts and rank the n passages based on their relevance. Applying it to our case, we could use this method to rerank the pass@n to get a better pass@k with $n > k$.

3.1.1 Generating descriptions

Command R+ is Cohere's newest large language model, optimized for conversational interaction and long-context tasks. Therefore, Cohere's LLM will be used to rerank. Cohere's chat endpoint was the fastest and easiest way to use Command R+. At present, there is a list of 50 codes for each problem on OctoCoder and DeciCoder on Human Eval, generated by Loubna Ben Allal. The problems that have at least 1 code passing the asserts and 4 that do not were arbitrarily identified. Command R+ (see 3.1) was then used to create descriptions of the identified code.

3.1.2 Prompt Design

We used this prompt to create a description for a code.

Listing 3.1: Prompt used to create a description

```
You are an python expert. Explain to me the goal of the following
   function and how it works:
{code}
```

Based on Pradeep et al. (2023)'s paper, we will use their prompt as a model and modify it so that it matches our goal.

Listing 3.2: Prompt used to rerank

```
You are an intelligent assistant that can rank codes based on their
   description and their python doctring.
Only response the ranking results, do not say any word nor explain.
I will provide you with the docstring.
Docstring: {docstring}
```

```
I will provide you with n code and, each indicated by number
    identifier [].
Descriptions:
[{task_id}]{description1}
[{task_id}]{description2}
...

The description that respects the most the docstring should be
    listed first. The output format should be a python list, e.g.,
    [2, 1].
Only response the ranking results, do not say any word nor explain
    and write only the sorted list.
Rank now!
```

3.1.3 Results

The NDCG (Normalized Discounted Cumulative Gain) metric was used to evaluate the ranking. This metric ranges from 0 to 1, with values closer to 1 indicating a more relevant ranking. The NDCG metric was used to evaluate the reranking performed by the LLM against the ground truth. Each code was labelled with 0 when it did not pass all asserts and 1 if it did. The `ndcg_score` from the `sklearn.metrics` library in Python was utilized for this evaluation. The resulting NDCG scores were 0.61 for OctoCoder and 0.63 for DeciCoder.

3.2 TLA+ and PlusCAL

TLA+ and PlusCAL (see Lamport (2014)) were invented by Leslie Lamport, a renowned computer scientist known for his contributions to distributed systems and formal methods in software engineering.

3.2.1 Temporal logic

Temporal logic is used to reason about propositions qualified in terms of time. It is particularly useful in verifying the behavior of concurrent and reactive systems. Temporal logic extends classical logic by introducing temporal operators that allow statements to express how propositions evolve over time.

- G (Globally): A proposition p is true at all points in time.
- F (Finally): A proposition p is true at some point in the future.
- X (Next): A proposition p is true at the next time step.
- U (Until): A proposition p is true until another proposition q becomes true.

3.2.2 TLA+

TLA+ is a model checker. Model checking is a technique performed by a software to verify that a system conforms to a set of formal specifications. It is a rigorous method to ensure that a system model meets desired properties such as safety (e.g. the code does not produce incorrect

outputs), liveness (e.g. ensuring that certain conditions will eventually happen during the execution of a program), and other constraints specified in temporal logic. TLA+ provides a way to model complex code at a high level of abstraction. TLA+ specifications allow to formally specify the behavior of functions, including its recursive definition and properties, providing a structured approach to verify its correctness and behavior. It can play a crucial role in verifying code generated by Large Language Models. Moreover, the TLA+ toolbox provide a model checking tool (TLC) to verify properties specified in your model.

3.2.3 PlusCAL

PlusCAL is often described as a "layer" or "superset" over TLA+. While TLA+ is a formal specification language used for describing and reasoning about systems, PlusCal is a high-level algorithm language that is easier to write and understand for many users, especially those familiar with traditional programming languages. Thanks to the toolbox (see Lamport (2014)), PlusCAL code can be translated into TLA+.

We thought of 2 approaches:

- first:
  - Generate the code in the target language (e.g. Python)
  - Translate it in PlusCAL by using a LLM
  - Generate specifications in TLA+ using a LLM
  - Test the specifications
- second:
  - Generate the code in PlusCAL
  - Generate specifications in TLA+ using a LLM
  - Test the specifications
  - Translate it in the target language by using a LLM

We will focus on the first approach.

3.2.4 Generating PlusCAL code

In the first approch, we have to generate PlusCAL code using the Python code. For this task, we used Ollama allowing us to use open source LLMs such as Llama2, Llama3, Phi3, Mistral and Mixtral. We used the following prompt:

Listing 3.3: Prompt used to translate Python to PlusCAL

```
You are an expert in formal verification. Your favorite language is
   PlusCAL and you know how to code in Python.
Task: You will translate the given Python function into PlusCAL
{code}
```

However, all above-mentioned LLMs did not produce correct PlusCAL codes. All errors were because of syntax.

3.2.5 Andrew Helwer's experience and reflection

On Google Groups, Helwer (2023) shared his experience in training GPT4 in TLA+. This lead us to believe that if the model was more trained or knew better PlusCAL, it would be able to produce correct PlusCAL code. Therefore, we prompted GPT4 how PlusCAL and its syntax work and stored its answer to give a better context to the used LLMs. Although the idea sounded good, it was not enough, so we added reflection. Reflection is a process that uses the error stack and sends it to the model with the associated code so that it can correct itself. The goal is to converge the model's response to an accurate one.

However, it seems that is was a red herring. After more than 10 iterations of reflection, the produced PlusCAL code still had syntax errors.

4. Conclusion

This paper presents work on LLMs and formal verification, starting with an exploration of Phi1.5. Python and its libraries were utilized to generate code on datasets (Human Eval and MBPP). The pass@1 results reported by (Li et al., 2023b) on Human Eval were replicated, and the pass@1 on MBPP was very close. Some issues were encountered with the MBPP dataset due to ambiguous prompts, including an attempt to hardcode a problem, which emphasizes the importance of formal verification.

The study adopted a rerank approach using the pass@50 on OctoCoder and DeciCoder, achieving NDCGs of 0.61 and 0.63 respectively. Attempts were made to translate Python code into PlusCAL using LLMs and generate specifications in TLA+ based on the asserts. However, LLMs appeared insufficiently trained on these languages, and even explaining the language to them and using reflection did not seem effective.

4.1 What's next?

A rerank approach was used, showing promising results. The execution was limited to problems with at least 1 code passing the asserts and 4 that do not. Future steps could include:

- Executing on more codes (for instance, 2 codes that pass the asserts and 8 that do not, and comparing the results)
- Randomly selecting codes to create descriptions

Furthermore, all code produced this far has been in Python. The research could expand to other languages such as Java, JavaScript, or C++. Additionally, further exploration of TLA+ and PlusCAL could be pursued by training an LLM on these languages. Alternatively, other tools like Coq could be investigated, leveraging a library that transpiles Python into Coq.

5. Bibliography

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models. (page 1)

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. (2022). Multipl-e: A scalable and extensible approach to benchmarking neural code generation. (page 2)

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code. (page 1)

Clinton, D. (2023). Understanding ai: Llm basics for investors. Accessed on 09/06, 2024. (page 1)

Helwer, A. (2023). Experience report: teaching tla+ proofs to chatgpt. Accessed on 07/06, 2024. (pages 2 and 11)

Huet, G. and Coquand, T. (1989). Coq. Accessed on 07/06, 2024. (page 2)

HuggingFace (2024). Open llm leaderboard. Accessed on 07/06, 2024. (page 2)

Lamport, L. (2014). Leaning tla+. Accessed on 07/06, 2024. (pages 9 and 10)

Lamport, L. (2024). Tla+ toobox. Accessed on 07/06, 2024. (page 2)

Li, C., Wang, J., Zhang, Y., Zhu, K., Hou, W., Lian, J., Luo, F., Yang, Q., and Xie, X. (2023a). Large language models understand and can be enhanced by emotional stimuli. (page 1)

Li, Y., Bubeck, S., Eldan, R., Giorno, A. D., Gunasekar, S., and Lee, Y. T. (2023b). Textbooks are all you need ii: phi-1.5 technical report. (pages 2, 4, and 12)

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. (2024). Octopack: Instruction tuning code large language models. (page 2)

Piché, A., Milios, A., Bahdanau, D., and Pal, C. (2024). Llms can learn self-restraint through iterative self-reflection. (page 2)

Pradeep, R., Sharifymoghaddam, S., and Lin, J. (2023). Rankzephyr: Effective and robust zero-shot listwise reranking is a breeze! (pages 2 and 8)

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. (2024). Code llama: Open foundation models for code. (page 2)

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. (page 2)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need. (page 1)

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. (2024a). Opencodeinterpreter: Integrating code generation with execution and refinement. (page 2)

Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., and Chen, J. (2024b). A survey of large language models for code: Evolution, benchmarking, and future trends. (pages 1 and 2)