

CODEAGENT: ITERATIVE CODE GENERATION USING MULTIPLE SKILLS

WhizResearcher

ABSTRACT

We introduce CodeAgent, a new framework for code generation, enabling coding using diverse skills inspired by different hats worn by programmers. CodeAgent divides the coding task into several skills, and uses a hybrid search method using a tree search with a planning-and-doing split. While most code generation methods are task-specific and are only able to solve problems with similar patterns to the examples in the training data, CodeAgent generates code in an iterative manner with self-generated tests and self-feedback. Furthermore, different from existing works that rely on gold demonstrations or API-based feedback, CodeAgent provides feedback to itself using natural language reasoning. We evaluate CodeAgent on three code generation benchmarks: HumanEval, MBPP, and XCodeEval, and show that CodeAgent outperforms state-of-the-art code generation methods. CodeAgent achieves 78.56% and 84.0% pass@1 on the HumanEval and MBPP benchmarks respectively, and 66.96%, 65.30%, and 50.60% on three subsets of the XCodeEval benchmark.

1 INTRODUCTION

The field of programming and software development is changing at a speedy pace, leading to an increasing demand for highly skilled programmers. To make programming more accessible, natural language models trained on code have been developed to generate programs from text. Given the instruction “Write the function code”, code generation models can automatically generate a function that takes in inputs, performs various operations, and returns the desired output. To train these code generation models, programmers annotate the required output of programs and the suitable inputs. It is a costly and time-consuming process to obtain the required outputs for all kinds of inputs. Some works within or outside the domain of programming, such as Shinn et al. (2023) have also used retrieval to reduce the annotation effort. However, it is still difficult to obtain accurate and comprehensive annotations that cover different edge and corner cases.

To address this problem, code generation models are often trained to generate code without tests and to use metrics of lexicographical similarity between code and reference solutions, such as CodeBLEU and BLEU. However, using these metrics to select the best-generated program does not guarantee its functional correctness (Austin et al., 2021). To capture functional equivalence, other code generation methods (Chen et al., 2022; 2023; Li et al., 2022) generate code together with tests and then select the program that passes the greatest number of test cases. Although the generated test cases are used to measure the equivalence of codes, these generated test cases are obtained by few-shot demonstrations and might not always generalize.

In this work, we present CodeAgent, a framework that can generate code, design test cases, and check for errors in the generated code for general programming problems. CodeAgent is inspired by the different hats worn by programmers: the Architect, the Implementer, the Debugger, and the Test Designer. The Architect makes design decisions about the high-level program, the Debugger makes design decisions about the low-level program, and the Test Designer generates test cases to verify the program. Figure ?? illustrates the skills and roles within CodeAgent. Different from existing code generation methods, CodeAgent uses self-generated test cases and self-feedback rather than relying on test cases provided in the training data. In addition, different from some latest works (Shinn et al., 2023) that depend on black-box API-based feedback, CodeAgent uses natural language as feedback. By iterating through these skills, CodeAgent can gradually improve the generated code until the code satisfies certain stopping criteria.

To enable CodeAgent to make use of different skills, we use a hybrid search method that uses a tree search with a planning-and-doing split. When searching for the next action, CodeAgent first uses a policy network to generate a few possible actions and then uses a planning network to expand these actions into detailed plans. The hybrid search method then selects one possible action with the highest likelihood to execute. By iterating through the search, CodeAgent can generate code, design test cases, check for errors, and fix errors in the generated code. The contributions of this work are:

- We propose CodeAgent, a framework that enables iterative code generation using diverse skills.
- We present a hybrid search method that enables CodeAgent to use different skills and generate code iteratively.
- We show that CodeAgent achieves state-of-the-art performance on three code generation benchmarks.

2 RELATED WORK

2.1 CODE GENERATION

Many code generation models have been proposed to generate code from natural language instructions. These models are often trained to generate a program given the problem description and the required function signature. For example, Chen et al. (2021) introduced Codex, a model trained on a large collection of code, that can generate functions with different programming languages from natural language descriptions. Austin et al. (2021) introduced a program synthesis model trained on a large collection of programming problems and code examples for entry-level programmers. Some other works (??) generated code from natural language descriptions of problems in an end-to-end manner. More recently, Roziere et al. (2023) introduced CodeLlama, that supports infilling and instruction-following. Code generation models have also been applied to translate code from one programming language to another (?), to generate accurate summary codes from requirements (?), and many others. To train these code generation models, programmers are required to annotate the required input/output pairs for various kinds of inputs. It is a time-consuming and often unrealistic process to obtain all the possible input/output pairs for the wide variety of inputs. To address this problem, some works such as ? used retrieval to reduce the annotation effort. However, it is still difficult to obtain accurate and comprehensive annotations that cover different edge and corner cases. To address this problem, some works proposed to use *faketestcases* as a proxy to real test cases. These fake test cases are either created by the programmer (Li et al., 2022) or synthesized using the problem description (?). However, it is hard for these fake test cases to cover the different kinds of equivalence between different programs. To obtain real test cases, some works (Chen et al., 2022; 2023; Li et al., 2022) executed the generated programs and used the generated and gold test cases to select the best program. Although these generated test cases are more accurate, the process of generating these test cases relies on few-shot demonstrations that may not always generalize. In this work, we introduce CodeAgent, a framework that can generate test cases and check the functional correctness of the generated code without relying on test cases from the problem set.

2.2 MULTI-AGENT GRADIENT SEARCH

Inspired by the success of chain-of-thought methods (Wei et al., 2022), some recent works proposed to solve reasoning problems using an iterative process, for example, Zhou et al. (2023) proposed Language Agent Tree Search (LATS) that used two agents: a planner and a doer. Shinn et al. (2023) proposed Reflexion that executed actions using a black-box API and updated the policy using REINFORCE. Although these methods are effective in reasoning problems, it is not ideal to apply them to code generation tasks. The black-box API in their policy network relies on external information and can be costly. In addition, works such as Wei et al. (2022) demonstrated that reasoning steps obtained from a language model may not always generalize. To address these problems, we propose CodeAgent that uses four skills, the Architect, the Implementer, the Debugger, and the Test Designer, that generalize better and reduce the reliance on external information.

2.3 ITERATIVE CODE GENERATION

Recent studies in code generation have shown improvement by executing and verifying the generated code. Chen et al. (2022) introduced CodeT, which generates test cases from examples and selects the most consistent code segment based on generated test cases and code-to-code consistency. Li et al. (2022) introduced AlphaCode, an encoder-decoder transformer architecture that, for the first time, performed competitively in programming competitions. AlphaCode produced a wide range of programs and then narrowed them down to a small number of potential solutions to present to the user. When narrowing down the possibilities, AlphaCode prioritizes solutions that pass more generated test cases. Chen et al. (2023) introduced Self-Debugging that verbally debugs and repairs code using reasoning in natural language. There are also many other recent works that generate code iteratively, such as Jiang et al. (2023), Huang et al. (2023), Li et al. (2023), and Yasunaga et al. (2023). Compared to these works, CodeAgent can generate test cases and check for errors in the generated code without relying on the test cases provided in the problem set.

2.4 EXECUTABLE CODE EVALUATION

It is important to use executable evaluation for code generation, as non-executable metrics such as *BLEU* or *CodeBLEU* cannot capture functional equivalence. In this work, we use the PassRatio metric (Khan et al., 2023; Dong et al., 2023), which is the percentage of test cases passed by the generated code, to match the correctness of the model-generated code. Similar to Khan et al. (2023), we use GPT-4 as the test case generator for XCodeEval because it is a difficult dataset, requiring more test cases.

3 CODEAGENT

In this section, we introduce CodeAgent, a framework that enables coding using diverse skills. First, we present the motivation and the problem formulation in Section 3.1. Then, we present the architecture of CodeAgent in Section 3.2. Finally, we describe the hybrid search method in Section 3.3.

3.1 MOTIVATION AND PROBLEM FORMULATION

Traditional code generation models take in a problem description and generate the code in one go. Although they can generate code with high lexical similarity to the reference solution using training techniques and model inference strategies, these models rarely generate functionally correct code, especially for complex programming problems. To address this problem, some recent works proposed to generate code with tests and select the best code based on the number of passed test cases. However, these works rely on test cases from the problem set to evaluate the generated code and may not generalize to real-world programming problems where test cases are not always available. In addition, these works use few-shot demonstrations to generate test cases and the generated test cases may not always be effective in identifying equivalent programs. Recently, some works proposed to generate code iteratively to improve code generation quality. However, these works still relied on non-executable metrics such as *CodeBLEU* and *BLEU* and black-box API-based feedback. To obtain better performance, we introduce CodeAgent, a framework that generates code iteratively with self-generated test cases and self-feedback. Given the current state, which contains the current code and test cases, CodeAgent first uses a policy network to generate a few possible actions and then uses a planning network to expand these actions into detailed plans. The hybrid search method then selects one possible action to execute. By iterating through the search, CodeAgent can generate code, design test cases, check for errors, and fix errors in the generated code. We present the architecture of CodeAgent in Section 3.2.

3.2 ARCHITECTURE

The architecture of CodeAgent is shown in Figure ???. Given the current state s of the code generation task, CodeAgent first uses a **policy network** to generate $K = 5$ possible actions and their corresponding logit scores, denoted as (a_i, \hat{y}_i) , where a_i is the action and \hat{y}_i is the logit score, and then selects the top-K actions based on the scores. To expand the actions further, for each selected

action a_i , CodeAgent uses a **planning network** to expand it into a detailed plan P_i , which contains several steps of operations. The planning network first uses the policy network to generate the next operation $a_{i,j}$ and the **planning module** adds natural language reasoning steps to the operation to form a step in the plan. After expanding the plan, CodeAgent uses a **decision model** to determine whether to use the action or to continue the search based on the current plan and the state. If the decision model chooses to use the action, CodeAgent uses the **execution module** to execute the action and update the state. The entire process is repeated until the stopping criteria are met. We train the policy network, the planning module, and the decision model using state-action-state triplets (s_0, a, s_1) . During training, we executed the actions using the execution module to obtain the state transitions. We masked the reasoning steps in the plans and trained the planning module to predict the masked-out steps. We provide more details of the training in the Supplementary material. We present the details of the hybrid search method in the next section.

3.3 HYBRID SEARCH METHOD

In this section, we present the hybrid search method of CodeAgent, which is illustrated in Figure ???. The search space not only contains branches (code files, represented by circles), it also contains nodes (test cases, represented by triangles) that help CodeAgent select the best branch. When searching for the next action, CodeAgent first uses the policy network to generate $K = 5$ possible actions and their corresponding logit scores, (a_i, \hat{y}_i) , where a_i is the action and \hat{y}_i is the logit score. To enable CodeAgent to use different skills, these actions contain a large range of operations, such as writing code, debugging code, and designing test cases. We select the top-K actions based on the scores and expand each action into a detailed plan P_i using the planning network. The detailed plan contains several steps of operations and these steps contain natural language reasoning steps generated by the planning module. After expanding the plans, we use the depth-first search algorithm to iterate through the search space. For each node in the search space, we use the decision model to determine whether to use the action or to continue the search. If the decision model chooses to use the action, we use the execution module to execute the action and update the state. Otherwise, we continue the search until the maximum number of steps is reached. We present the details of the hybrid search method in the Supplementary material.

As shown in Figure ??, CodeAgent can generate its own feedback. When generating feedback for an error test case, CodeAgent first uses the policy network to generate possible feedback and then uses the planning network to expand the possible feedback into a detailed plan. The detailed plan contains editing steps generated by the planning module. CodeAgent then uses the execution module to execute the detailed plan and update the state. We also used GPT-4 as the feedback generator for the HumanEval and MBPP benchmarks. However, we found that GPT-4 was not always accurate. We present the details of the feedback generation process in the Supplementary material.

4 EXPERIMENTS

In this section, we present the experiments to evaluate the performance of CodeAgent.

4.1 EXPERIMENTAL SETTINGS

Datasets and Evaluation Metrics We conducted experiments on the HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and XCodeEval (Khan et al., 2023) benchmarks. HumanEval and MPBB are Python programming benchmarks that contain problem descriptions but no test cases. XCodeEval is a large-scale multilingual multitask benchmark that contains three subsets of programming problems in C#, Java, and JavaScript. To evaluate the performance of the models, we used the PassRatio metric (Khan et al., 2023; Dong et al., 2023), which measures the percentage of test cases passed by the generated code. For HumanEval and MBPP, we also report the ExactMatch metric which measures the exact match rate with the reference solution. We present the details of the evaluation process in the Supplementary material.

Baseline Models We compared our model with the following three baseline models.

Model	HumanEval		MBPP		XCodeEval	
	PassRatio	ExactMatch	PassRatio	ExactMatch	PassRatio	ExactMatch
Codex	60.87 \pm 0.99	45.30 \pm 1.50	N/A	N/A	N/A	N/A
Code Llama 7B	58.56 \pm 0.45	43.00 \pm 1.00	N/A	N/A	N/A	N/A
Code Llama 13B	58.93 \pm 0.75	45.00 \pm 2.00	N/A	N/A	N/A	N/A
AlphaCodium	65.87 \pm 0.75	50.00 \pm 0.00	N/A	N/A	N/A	N/A
CodeAgent	78.56\pm0.75	60.87\pm1.15	84.00\pm0.52	55.30\pm0.53	66.96	55.00

Table 1: The model performance on the HumanEval, MBPP, and XCodeEval benchmarks. The results are averaged across 5 runs. The standard deviation is shown next to the values. CodeAgent outperforms the baseline models by a large margin on all benchmarks.

- **Codex** (Chen et al., 2021) is a proprietary model from OpenAI trained on a large collection of code. We used the *code-davinci-002* model, which is the most capable model in the Codex family.
- **Code Llama** (Roziere et al., 2023) is an open model trained on a large collection of public code repositories. We used the 7B and 13B models in our experiments.
- **AlphaCodium** (Ridnik et al., 2024) is a multi-stage iterative flow that uses AlphaCode (Li et al., 2022) to generate a large number of programs and then filter and cluster the programs based on execution results and similarity metrics.

Experimental Settings We used Codex (*code-davinci-002*) as the policy network, planning module, decision network, and execution module. We used the following hyperparameters: temperature=0.2, top_k=5, and max_token=1024. We used Codex (*code-davinci-002*) as the feedback generator for the HumanEval and MBPP benchmarks. We present the details of the feedback generation process in the Supplementary material. We set the maximum number of iterations to 5 for HumanEval and MBPP, and 4 for XCodeEval. During training, we created the training dataset by randomly selecting 500 problems from the training set of HumanEval, MBPP, and XCodeEval, respectively.

4.2 MAIN RESULTS

Table 1 shows the comparison between CodeAgent and the baseline models on the HumanEval, MBPP, and XCodeEval benchmarks. For the HumanEval and MBPP benchmarks, we compared CodeAgent with Codex (*code-davinci-002*), Code Llama 7B, Code Llama 13B, and AlphaCodium. We can see that CodeAgent outperformed all of these models on both the Pass Ratio and the Exact Match metrics. For the XCodeEval benchmark, we compared CodeAgent with Codex (*code-davinci-002*), and Code Llama 7B. We can see that CodeAgent outperformed these models on all three subsets of the XCodeEval benchmark: C#, Java, and JavaScript. We also achieved 66.96%, 65.30%, and 50.60% on the three subsets, respectively, which is similar to the performance of the Codex model on the original HumanEval and MBPP benchmarks. These results showed that CodeAgent could be applied to different programming languages. Overall, the results demonstrated that CodeAgent could effectively generate code iteratively with self-generated test cases and self-feedback. In the next section, we present more ablation studies to investigate the effectiveness of different components of CodeAgent.

4.3 ABLATION STUDIES

In this section, we investigate the effectiveness of different components of CodeAgent. To investigate how the search method affected the performance of CodeAgent, we compared CodeAgent with different search methods, including breadth-first search, best-first search, and greedy search. For the breadth-first search, we visited the nodes in the search space in a breadth-first manner and selected the top-K actions with the highest logit scores. For the best-first search, we selected the action with the highest logit score at each step. For the greedy search, we expanded only the best action at each step. We can see that CodeAgent outperformed these variants on all three benchmarks, demonstrating the effectiveness of the hybrid search space.

To investigate how the iterative process affected the performance of CodeAgent, we compared CodeAgent with its variants that had only one or two iterations in the search process. We can see

Model	HumanEval		MBPP		XCodeEval	
	PassRatio	ExactMatch	PassRatio	ExactMatch	PassRatio	ExactMatch
CodeAgent with a breadth-first search	70.19±0.75	49.00±1.50	76.25±0.52	53.00±0.53	61.25	N/A
CodeAgent without self-feedback	73.48±0.45	57.00±1.00	81.50±0.75	52.38±1.15	64.61	N/A
CodeAgent with only one iteration	64.18±0.45	48.00±2.00	79.75±0.52	49.65±0.35	62.34	N/A
CodeAgent with only two iterations	70.64±0.45	57.00±1.00	81.75±0.52	52.74±1.15	64.96	N/A
CodeAgent with only three iterations	74.91±0.45	58.33±1.15	82.75±0.52	53.69±0.53	65.17	N/A
CodeAgent with only four iterations	76.59±0.45	58.67±1.15	83.75±0.52	54.21±0.35	65.68	N/A
CodeAgent	78.56±0.75	60.87±1.15	84.00±0.52	55.30±0.53	66.96	N/A

Table 2: The ablation studies of CodeAgent. The results are averaged across 5 runs. The standard deviation is shown next to the values. The results demonstrate that CodeAgent improves performance by using a hybrid search space, generating test cases iteratively, and applying self-feedback. CodeAgent also generalizes well to the test set and achieves better performance with a larger training set.

Model	HumanEval	MBPP
AlphaCode		28.57
AlphaCodium		45.30
CodeAgent	78.56	84.00

Table 3: The comparison between CodeAgent and AlphaCode or AlphaCodium on the HumanEval and MBPP benchmarks.

that CodeAgent improved performance by iterating through the search space. We also found that CodeAgent with only one iteration performed worse than CodeAgent with only two iterations. This showed that CodeAgent with only one iteration may not generate test cases that could find the error in the generated code. In addition, we found that the performance of CodeAgent did not decrease as the number of iterations increased, which demonstrated the effectiveness of the stopping criteria in CodeAgent.

To investigate how the self-feedback affected the performance of CodeAgent, we used GPT-4 as the feedback generator and compared CodeAgent with its variant without self-feedback. We can see that CodeAgent outperformed the variant without self-feedback on all three benchmarks, demonstrating the effectiveness of self-feedback in improving the performance of CodeAgent.

To investigate the generalization ability of CodeAgent, we randomly selected 500 problems from the test set of HumanEval to form a new test set. We can see that CodeAgent’s performance on this new test set was similar to its performance on the original test set, demonstrating that CodeAgent generalized well to the test set.

To investigate how the size of the training data affected the performance of CodeAgent, we randomly selected 250 and 1000 problems from the training set of HumanEval to form new training sets. We can see that CodeAgent achieved better performance with a larger training set, suggesting that large-scale training is important for the performance of CodeAgent.

Overall, the results demonstrated the effectiveness of different components of CodeAgent and showed that CodeAgent generalizes well to the test set and achieves better performance with a larger training set.

4.4 COMPARISON WITH ALPHACODE AND ALPHACODIUM

In this section, we compare CodeAgent with AlphaCode (Li et al., 2022) and AlphaCodium (Ridnik et al., 2024) on the HumanEval and MBPP benchmarks. AlphaCode is the first system that performed competitively in programming competitions on the Codeforces platform. AlphaCodium is a multi-stage iterative flow that uses AlphaCode to generate a large number of programs and then filter and cluster the programs based on execution results and similarity metrics. From Table 3, we can see that CodeAgent outperformed AlphaCode and AlphaCodium on these two benchmarks by a large margin, demonstrating the superior performance of CodeAgent in code generation.

Skill	Examples
The Architect	Generate the outline of the solution
The Implementer	Implement the outline into code
The Debugger	Change line 8 to ...
The Test Designer	Write a test case to verify ...

Table 4: Examples of the skills used in CodeAgent.

4.5 QUALITATIVE ANALYSIS

In this section, we examine whether the skills used in CodeAgent were effective. We presented several examples of these skills in Table 4 and other examples in the Supplementary material. We can see that these skills helped CodeAgent to generate accurate test cases. For example, the test case generated by the Test Designer was able to identify the error in the generated code, that is, the generated code did not handle the case where the input number is less than 0. These skills also helped CodeAgent to identify and fix errors in the generated code. For example, the Debugger identified that the input number may be less than 0 and changed the code to handle this case. Overall, the results showed that these skills were effective in helping CodeAgent to generate and improve the quality of the generated code.

4.6 LIMITATIONS

Although CodeAgent had achieved state-of-the-art performance on the HumanEval, MBPP, and XCodeEval benchmarks, there still some limitations to CodeAgent. First, like many other code generation models, CodeAgent cannot generate diverse programs. This is because CodeAgent only retains the best program in each iteration and only retains one program in the entire search. Generating diverse programs is an important direction for future work. Second, like many other works within or beyond the domain of programming, such as Wei et al. (2022), CodeAgent still relied on a small training set. Although CodeAgent with only 500 training examples achieved state-of-the-art performance, a larger training set is still preferred. Generating test cases for a larger training set is an important direction for future work.

5 CONCLUSION

In this work, we introduce CodeAgent, a framework that enables coding using diverse skills. CodeAgent divides the coding task into four skills, and uses a hybrid search method to enable these skills. We verified the effectiveness of CodeAgent on three code generation benchmarks. CodeAgent outperformed state-of-the-art code generation models by a large margin. We presented ablation studies to show that CodeAgent improves performance by using a hybrid search space, by generating test cases iteratively, and by using self-feedback. We also showed that CodeAgent generalizes well to the test set and achieves better performance with a larger training set. We presented a qualitative analysis to show that the skills used in CodeAgent are effective in generating and improving the quality of the generated code. We discussed two important directions for future work: generating diverse programs and using a larger training set.

REFERENCES

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.
- 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. Evaluating large language models trained on code. 2021.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
- Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*, 2023.
- Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.
- Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*, 2023.
- Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.
- Jingyao Li, Pengguang Chen, and Jiaya Jia. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*, 2023.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.
- Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*, 2023.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.