CoderGen: Towards Domain-Specific Code Generation of Large Language Models

Anonymous ACL submission

Abstract

Automated code generation is a pivotal capability of large language models (LLMs). However, assessing this capability in real-world scenarios remains challenging. Previous methods focus more on low-level code generation, such as model loading, instead of generating high-level codes catering for real-world tasks, such as image-to-text, text classification, in various domains. Therefore, we construct AICoderEval, a dataset focused on real-world tasks in various domains based on HuggingFace, PyTorch, and TensorFlow, along with comprehensive metrics for evaluation and enhancing LLMs' task-specific code generation capability. After that, we propose CoderGen, an agent-based framework, to help LLMs generate codes related to real-world tasks on the constructed AICoderEval. Moreover, we train a more powerful task-specific code generation model, named AICoder, which is refined on codellama based on AICoderEval. Our experiments demonstrate the effectiveness of CoderGen in improving LLMs' task-specific code generation capability (by 30.26% on SR@All and 19.88% on SR@Any). And the proposed AICoder also outperform the current code generation LLMs, indicating the great quality of the AICoderEval benchmark for evaluation and enhancing LLMs' task-specific code generation capability.

1 Introduction

Large language models attract attention for their general capabilities (Chowdhery et al., 2022; Brown et al., 2020; Workshop et al., 2023; Touvron et al., 2023; Du et al., 2022), achieve high scores on evaluations such as HumanEval (Chen et al., 2021) and MBPP (Ni et al., 2023), which primarily focus on basic programming languages. However, their application capabilities in real software development, especially in the field of artificial intelligence using specific libraries (such as HuggingFace, PyTorch, TensorFlow, etc.), remain unclear. Although these libraries are very popular in AI development, how to evaluate and improve the code generation capabilities of large language models using these libraries is still a hard question.

Current researches explore how to leverage LLMs to use tool to call specific libraries. For instance, studies such as HuggingGPT (Shen et al., 2023) and Gorilla (Patil et al., 2023) try to generate single-line calls of APIs in specific domains. These studies show that even simple API calls require models to have a deep understanding and the ability to correctly use the libraries. However, these studies have not yet fully addressed how to automate the evaluation and enhancement of models' code generation capabilities in flexibly using specific libraries, especially when dealing with complex and diverse programming tasks.

To address this challenge, we construct the AICoderEval dataset, a benchmark for AI-oriented programming tasks to measure programming capabilities within this domain. Then, we propose an agent-based framework called CoderGen, to generate task-specific codes. CoderGen simplifies the construction of datasets related to task-specific code on different libraries, enabling the automatic generation of training and testing samples. As illustrated in Figure 1, general code generation LLMs (e.g. codellama) may produce incorrect answers when it comes to pipeline and model API calls based on given function instructions. Our fine-tuned model demonstrates improved performance as it learns how to use the library for specific tasks. This approach allows for a more accurate assessment of a model's application capabilities in real software development and provides direction for further model improvements.

Our work includes three main contributions:

- Benchmark Construction: We build the AICoderEval dataset, which focuses on AI

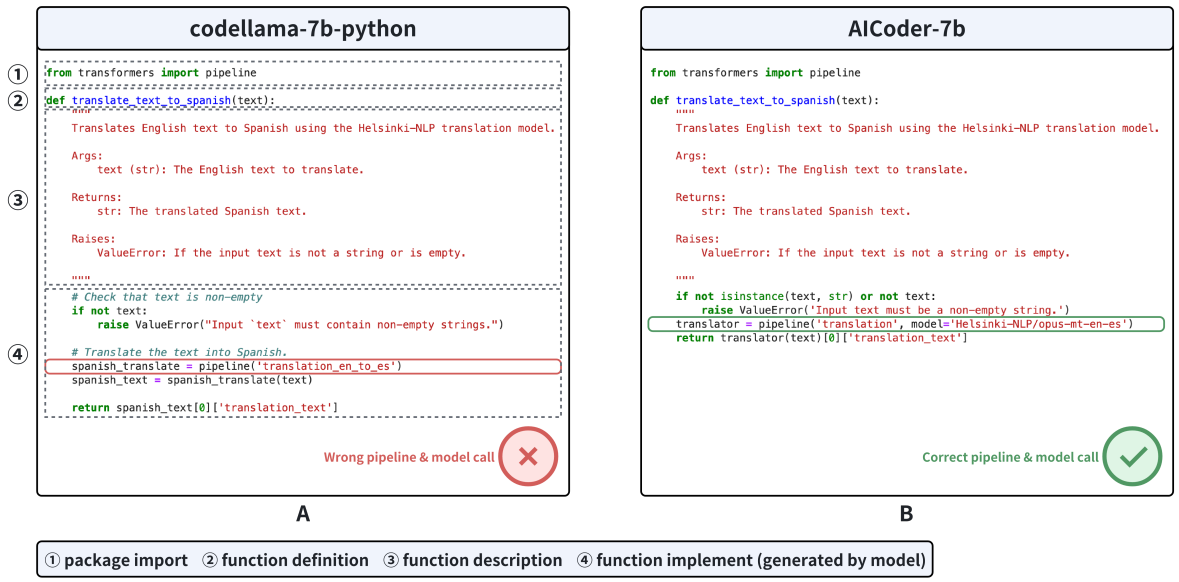

Figure 1: The AICoder generated by our CoderGen framework is capable of programming for domain-specific tasks and selecting the appropriate libraries for invocation. In part **A** depicts the output generated by codellama-7b-python, which incorrectly invoked a library using the pipeline method. In contrast, the part **B** presents the results produced by the AICoder, accurately selecting and calling the appropriate library to fulfill the requirements.

tasks and includes code generation tasks related to AI libraries, along with test cases and complete programs for evaluating these tasks. These tasks cover a variety of library functions and usage patterns, ensuring that the model learns comprehensive knowledge about the libraries.

- Framework Design: We design and construct the CoderGen framework to generate high-quality training data. During the inference stage, we use an LLM-based agent to guide the generation of code that adheres to specific library usage standards, with continuous improvements in code quality. The agent interacts with the model multiple times to refine and optimize the code generation process, making it more consistent with library usage norms and best practices.

- Model Evaluation: We evaluate multiple large language models on AICoderEval, demonstrating their code generation capabilities in actual AI development tasks and the performance enhancements after training with our framework. This approach allows us to compare the performance of different models and identify their strengths and limitations in using specific libraries.

Through these contributions, CoderGen provides a more comprehensive and practical evaluation method for the code generation capabilities of large language models and points the way for further model improvements. We hope this framework will assist researchers and developers in better understanding and leveraging the potential of large language models in software development, particularly when programming with specific libraries.

2 Benchmark Construction

2.1 Data Collection

Inspired by related work (Patil et al., 2023), we aim to leverage GPT-4 (OpenAI et al., 2023) to process data collected from the web and format it into a specific structure. Focusing on the field of artificial intelligence, we select the Hugging Face Hub and PyTorch Hub as our target libraries. Models within these libraries can be invoked through a unified API, and their descriptions and documentation are available on the official websites. To reduce the complexity introduced by library descriptions, we directly employ data that has been automatically processed and filtered by GPT-4 as our input, which we then further process to create the dataset we demand.

The data is derived from the web and filtered accordingly contains the following information:

Category	Cnt.	%
Natural Language Processing	383	77.8%
Computer Vision	50	10.2%
Tabular Data	18	3.7%
Audio and Speech	17	3.5%
Classification	12	2.4%
Multimodal	9	1.8%
Reinforcement Learning	3	0.6%
Total	492	100%

Table 1: Data Category Statistics

domain, model name, model description, example code, and performance metrics. Human programmers can make full use of this information to attempt development. Therefore, for a more intelligent agent, it is theoretically possible to utilize this information to learn how to develop based on library specifications.

2.2 Data Pre-processing

To construct a dataset capable of automated evaluation, we draw on the evaluation methodology of humaneval. Our goal is to generate executable code files using GPT-4, streamlining the process by focusing on Python code generation. Each file is meticulously structured to encompass a suite of components, ensuring the integrity of the tests: package installation instructions, package imports, main function definition, functionality description, function input/output/raise error descriptions, function implementation, testing function, and testing function calls.

To effectively guide GPT-4, we provide in-context prompts and examples, which serve to elicit a demand and an end-to-end solution based on specific APIs within the libraries. We also utilize the function calling feature of GPT-4, which allows for partial output and enhances the stability of the output. This approach also yields correct code examples that align with the given prompts. The prompt we are using is displayed in appendix A.1.

By consolidating the evaluation into a single code file, we simplify the testing process, enabling the execution of all tests through a solitary file. Moreover, we strive for diversity in the generated test cases, particularly in terms of difficulty. We adeptly guide GPT-4, through carefully crafted prompts, to produce three distinct test cases: the first assesses normal code execution, the second evaluates handling of exceptional inputs, and the third confirms correct results for normal inputs.

Following the processing of approximately 9,000 pieces of information related to AI library APIs, we proceed to generate a total of 9,000 files for testing purposes. Utilizing machines equipped with GPUs, we meticulously filter the results, retaining approximately 2,000 program files that pass at least one test case. We further refine our dataset to select about 500 program files that successfully pass all test cases. For our evaluations, we focus on the 500 files data, which demonstrates a comprehensive passing rate across all test cases. This approach ensures a rigorous and thorough evaluation of the generated code, while also providing a solid foundation for future research in automated code generation and evaluation.

In table 1, we further statistics on the categories and their proportions in the dataset. We counted the task categories within the dataset, with Natural Language Processing tasks comprising the largest share at 77.8%, followed by Computer Vision at 10.2%. Additionally, tasks such as Tabular Data, Audio and Speech, Classification, Multimodal, and Reinforcement Learning each account for less than 5%. Tasks in Natural Language Processing include text classification, text generation, sentence similarity matching, etc. In Computer Vision, tasks include image classification, image segmentation, image generation, and so on. These more specific tasks are not listed in the table.

3 Methodology

In this paper, we introduce **CoderGen**, an agent-based framework for generating codes on tasks in AICoderEval, as depicted in figure 2. This framework can construct domain-specific tasks benchmark, for training and evaluation, and then fine-tunes a code generation model on the benchmark.

3.1 Error Traceback and Analysis

The **CoderGen** framework includes a robust error traceback and analysis mechanism to ensure that the generated code is not only syntactically correct but also functionally sound. Figure 3 shows an example of error traceback and related prompt. After the initial code generation, the framework executes the code within a controlled environment to test its functionality. If the code fails to execute correctly, the system captures the error traceback, which provides a detailed record of the path through the code that led to the failure. This traceback is then an-

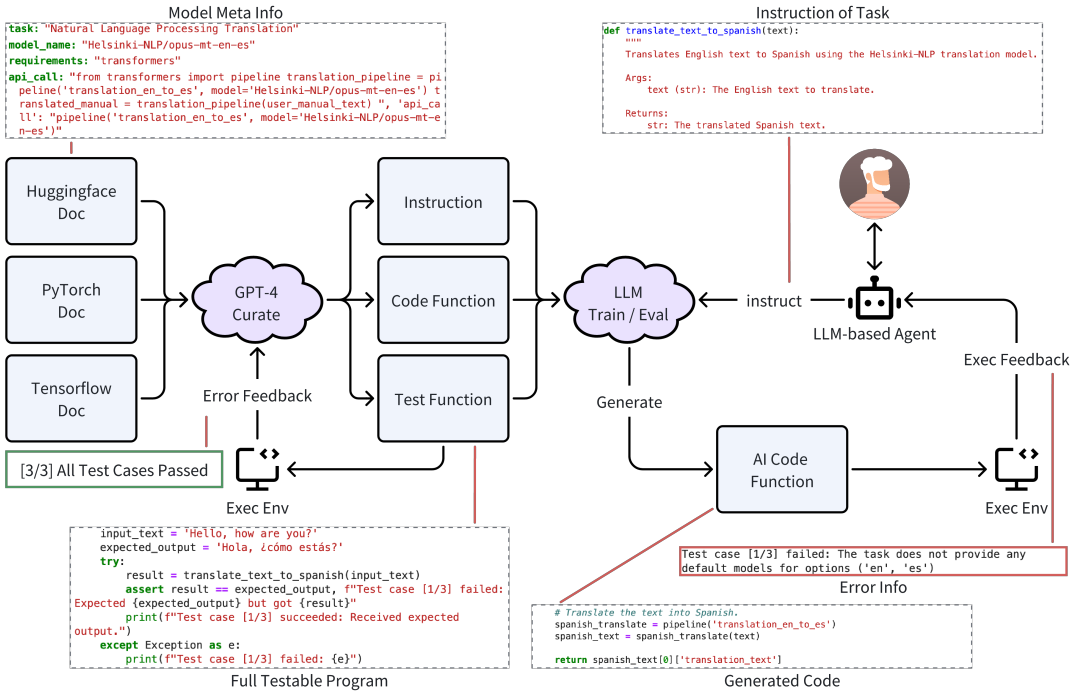

Figure 2: **CoderGen: A Domain-Specific Code Generation Architecture**. This architecture comprises two integral components. On the left side, **AICoderEval** data is produced by analyzing library documentation with provided document data (model meta-information). This data, which includes testable programs, is subsequently validated within an execution environment. We then utilize this data to train a LLM (**AICoder** in following paper). On the right side, an LLM-based agent is employed to direct the code generation process. Actual executable environments are utilized to push feedback to both the agent and the LLM, aiding in the refinement of the generated code.

Figure 3: Error traceback analyze example

alyzed by the framework to identify the specific point of failure, whether it be a syntax error, a logical error, or an issue with the code's interaction with external libraries or APIs.

The error analysis component of CoderGen leverages the fine-tuned language model to interpret the error messages and suggest potential fixes. These suggestions are based on the model's understanding of the code's intended functionality and the context of the error within the broader codebase. The suggestions are then presented to the user, who can choose to implement them, or they can be automatically applied by the system for further testing. This iterative process of error detection, analysis, and correction continues until the code successfully executes all test cases and meets the specified requirements.

3.2 Iterative Code Re-generation

Once the errors have been identified and suggestions for improvement have been made, the CoderGen framework enters the code re-generation phase. Here, the framework uses the feedback from the error analysis to refine the code generation process. The erroneous code snippet, along with the suggestions and the original instruction, are fed back into the language model, which then generates a new version of the code snippet.

This new code snippet is then retested, and the process of error detection, analysis, and correction is repeated. This iterative cycle ensures that the generated code not only resolves the immediate issues but also improves in quality and robustness with each iteration. The framework's ability to learn from its mistakes and adapt its code generation strategy based on real-time feedback is a key feature that sets CoderGen apart from traditional code generation systems.

By incorporating these iterative feedback loops, CoderGen aims to produce code that is not only correct but also efficient and maintainable, reflecting the best practices and idioms of the target domain. This approach has the potential to significantly reduce the time and effort required for developers to produce high-quality code, particularly in complex and specialized domains.

4 Experiment

4.1 Experimental Setup

We design a set of hyperparameters to optimize the training process and enhance the capabilities of the models. To foster diversity in the generated content, we set the temperature parameter to 0.7. Simultaneously, we adjust the top-p value to 0.95 to improve the precision of the generated outputs. We employ a learning rate of 2e-4 alongside beta values of (0.9, 0.999) to maintain the stability of the training process. We carefully configure the batch size to 4, with gradient accumulation steps, to ensure computational efficiency while maximizing resource utilization. In pursuit of a delicate balance between novelty and coherence, we utilize LoRA parameters with a rank of 8 and an alpha value of 32. Additionally, we fine-tune both the top-p value and the temperature parameter to 0.7.

4.2 Main Results

In this study, we utilized the **AICoderEval** dataset to test multiple popular API and open-source LLM models, particularly those equipped with code generation capabilities. The models tested included gpt-3.5-turbo-1106 supported by OpenAI, as well as Llama 2 7b / 13b / 70b (Touvron et al., 2023) and Codellama 7b / 13b / 34b (Rozière et al., 2024) models developed by Meta. Furthermore, we fine-tuned **AICoder** 7b/13b based on the codellama model. Table 2 presents a comparison of these models' performance in their original versions and after the introduction of an error repair agent, where **SR@All** represents the success rate of all tests passed for a single program, and **SR@Any** represents the success rate of any test case passed for a single program.

From Table 2, it is evident that the introduction of the error repair agent significantly improved the SR@All and SR@Any metrics for all models. On average, Large Language Models (LLMs) demonstrate a 30.26% improvement in task-specific code generation capabilities as measured by SR@All and a 19.88% enhancement as measured by SR@Any across all tested models. For instance, GPT-3.5-turbo-1106's SR@All increased from 9.16% to 13.03%, and SR@Any from 46.84% to 60.63%. This indicates that feedback correction can effectively enhance the generation accuracy and problem-solving capabilities of the models. Additionally, we observe that model scale has a significant impact on performance improvement. In the Llama 2 series, the larger the model scale, the more pronounced the performance improvement. For example, llama-2-70b exhibited a more significant increase in SR@All and SR@Any compared to llama-2-7b. After domain-specific

Models	Orig		Bugfix Agent		Relative Increase	
	SR@All	SR@Any	SR@All	SR@Any	SR@All ↑%	SR@Any ↑%
GPT-3.5-turbo-1106	9.16	46.84	13.03	60.63	42.25	29.44
llama-2-7b	1.23	26.02	1.83	33.41	48.78	28.40
llama-2-13b	2.76	42.04	3.98	51.24	44.20	21.88
llama-2-70b	6.32	65.89	8.16	**78.68**	29.11	19.41
codellama-7b-python	19.58	66.95	23.86	78.18	21.86	16.77
codellama-13b-python	20.46	67.22	23.88	75.67	16.72	12.57
codellama-34b-python	**23.68**	**70.19**	**25.78**	77.33	8.87	10.17
AICoder-7b	**27.55**	84.69	-	-	-	-
↑	3.87	14.50	-	-	-	-
↑%	16.34	20.66	-	-	-	-
AICoder-13b	26.53	**87.76**	-	-	-	-
↑	2.85	17.57	-	-	-	-
↑%	12.04	25.03	-	-	-	-

Table 2: Experiment on **AICoderEval** dataset

Models	CL	CT	Rank
GPT-3.5-turbo-1106	8.6	62.9	1
llama-2-7b	16.2	112.9	5
llama-2-13b	18.5	116.3	7
llama-2-70b	13.1	107.8	4
codellama-7b-python	21.5	128.3	9
codellama-13b-python	18.9	116.3	8
codellama-34b-python	18.4	114.4	6
AICoder-7b	13.6	86.6	3
AICoder-13b	12.5	83.4	2

Table 3: Experiment on **AICoderEval** dataset. **CL** is for average code lines, and **CT** is for average code tokens

	AICoder-7b		AICoder-13b	
	SR@All	SR@Any	SR@All	SR@Any
NLP	31.32	87.95	28.91	91.97
CV	0.00	71.43	0.00	71.43
Audio	0.00	50.00	0.00	25.00
MM	50.00	100.00	50.00	100.00
Tabular	0.00	50.00	50.00	100.00
RL	0.00	0.00	0.00	0.00
Overall	27.55	84.69	26.53	87.76

Table 4: **AICoder** evaluation on different category

fine-tuning, the performance of the original network can be significantly enhanced, as AICoder-7b achieved SOTA in SR@All and SR@Any compared to all tested baselines.

Table 3 shows the number of code lines (**CL**) and code tokens (**CT**) generated by different models. We can identify a pattern where shorter code generated by the models typically implies stronger problem-solving abilities and more concise solutions. For instance, codellama-34b-python had lower CL and CT than codellama-7b-python, which aligns with its relative performance in SR@All and SR@Any, while AICoder outperformed with significantly shorter generated code lines compared to other models.

Table 4 displays the performance of AICoder models across different task categories. We can see that for NLP tasks, AICoder-7b and AICoder-13b achieved SR@All of 31.32% and 28.91%, respectively, indicating good performance; however, both models performed poorly in CV and Audio tasks, suggesting inadequate training. This indicates that models have an advantage in specific tasks where they are well-trained, but face challenges in other domains.

In summary, the introduction of the error repair agent has significantly improved the overall performance of the models, whether it is the success rate of all tests passed for a single program (SR@All) or any test case passed (SR@Any). The increase in model scale has a positive impact on performance improvement, especially in the Llama 2 series where larger model scales result in more

	w/o sug		w/ sug	
	SR@All	SR@Any	SR@All	SR@Any
L.7b	1.60	31.58	**1.83**	**33.41**
L.13b	3.73	50.00	**3.98**	**51.24**
L.70b	7.48	76.19	**8.16**	**78.68**
CL.7b	**24.09**	**80.00**	23.86	78.18
CL.13b	23.21	73.66	**23.88**	**75.67**
CL.34b	**26.44**	**79.56**	25.78	77.33

Table 5: ablation on agent strategy w/ and w/o suggestion for different llama-series model. L.7b/13b/70b is short for Llama-2-7b/13b/70b. CL.7b/13b/34b is short for Codellama-7b/13b/34b-python

pronounced performance gains. The fine-tuning strategy has also demonstrated its effectiveness, particularly for the AICoder model, which achieved state-of-the-art performance in all tested baselines after fine-tuning. The performance of the models varies significantly across different task categories, indicating the necessity for domain-specific optimization and improvement.

4.3 Ablation

The table 5 presents an ablation study on the agent strategy for different Llama series models, including scenarios with and without suggestions. The models are categorized into two types: Llama-2 and Codellama, with metrics consistent with previous experiments, namely SR@All and SR@Any.

The results indicate that, overall, the models perform better with the suggestion-based strategy (w/ sug) compared to the strategy without suggestions (w/o sug). For instance, the Llama-2-7b model has an SR@All of 1.60% without suggestions, which improves to 1.83% with suggestions. Similarly, SR@Any increases from 31.58% to 33.41%. This trend is consistent across most models.

Among all the models, the Codellama-7b-python model with the suggestion-based strategy achieves the highest SR@All of 24.09% and the highest SR@Any of 80.00%. On the other hand, the Llama-2-7b model without the suggestion-based strategy performs the worst, with an SR@All of 1.60% and an SR@Any of 31.58%.

In summary, these results suggest that incorporating suggestions into the agent strategy can enhance the performance of Llama and Codellama series models.

4.4 Case Study

Initially, we perform a case study on the code produced by AICoder-7b. We provide a well-trained AICoder-7b with an instruction that encompasses the import of function packages, the definition of the function, and associated comments, enabling AICoder-7b to generate the complete code.

As depicted in Figure 1 (part B), the functional requirement for the task is as follows:

```
Translates English text to Spanish using the Helsinki-NLP translation model
```

Upon examining the code completed by the model, we note that AICoder-7b selects an appropriate model that can meet the requirement and invokes the necessary capabilities to accomplish the task. However, codellama-7b-python incorrectly chooses 'translation_en_to_es' as the pipeline name, which is an erroneous inference.

Furthermore, the agent is also capable of identifying corresponding exceptions and providing error messages. In another case detailed in Section 3, the model analyzes the error traceback from the execution environment of the previously generated code. The analysis suggests that a try-except block is necessary for the subsequent code generation. Subsequently, the LLM can process the prompt and generate new code accordingly. In the ablation study section, we discuss the strategies employed in the prompt design, which shows that the agent can enhance the overall system performance.

5 Related Work

5.1 Code Generation

Utilizing language models for code generation is a challenging task (Li et al., 2022; Xu et al., 2022; Jain et al., 2022). Researchers propose various methods to enhance the capabilities of language models in programming tasks, including task decomposition (Kim et al., 2023; Yao et al., 2023), self-debug (Chen et al., 2024), and code generation models. These efforts primarily focus on the generation of general code, with less attention given to the capabilities of domain-specific code. In real-world scenarios, however, we often use libraries to create new tools and implement more complex functionalities through longer chains of function calls. Therefore, our research aims to enable programs to automatically solve tasks using domain-specific libraries and to verify the results

automatically, thereby expanding the capabilities of code generation.

5.2 Tool Usage

Large language models can leverage tools to enhance their capabilities, such as Toolformer (Schick et al., 2023) and GPT-4 (OpenAI et al., 2023) making API calls more feasible. Traditional tools include web browsing, calculators, code interpreters, etc., with these efforts aiming to invoke general capabilities. HuggingGPT (Shen et al., 2023) and Gorilla (Patil et al., 2023), on the other hand, focus on domain-specific API calls. Our research aims to explore the programming capabilities of specific domain libraries, thereby expanding the scope of program usability.

5.3 Agent

An agent is generally represented as an entity with the capability to interact with the environment and take actions, either based on feedback from the environment or driven by intrinsic motivations. It exhibits greater adaptability and versatility in its capabilities and execution outcomes compared to ordinary programs. LLM-based Agents have recently been widely discussed (Xi et al., 2023; Wang et al., 2023; Park et al., 2023); they expand their capabilities through the use of tools, and planning ability is also one of the most important capabilities of LLM-based Agents. In the field of code generation, previous work has focused more on one-time code generation, such as CodeGen (Nijkamp et al., 2023), CodeX (Chen et al., 2021). However, in real-world scenarios, we approach the expected results incrementally through feedback from the actual environment, such as execution information and error messages. In this paper, our research aims to enable Agents to analyze error messages, allowing the program to execute correctly.

6 Conclusions and Future Work

This paper introduces CoderGen, an automatic learning and evaluation framework designed to improve the assessment of code generation capabilities, especially when dealing with libraries commonly used in real software development. CoderGen automatically constructs an evaluation dataset, AICoderEval, for libraries related to artificial intelligence, and trains a domain-optimized code generation model based on this dataset. Furthermore, the AICoder model is fine-tuned on the codellama dataset and evaluated on the AICoderEval dataset, demonstrating its superiority over other code generation models. Our work represents a significant advancement in evaluating and enhancing code generation capabilities in real software development by focusing on the understanding and application of libraries commonly used in actual software development processes. In future work, we plan to optimize the CoderGen framework to support a wider range of libraries and software development scenarios, validate its generality and effectiveness with diverse datasets and tasks, and integrate it with the latest code generation technologies to further enhance model performance and practicality.

Limitation

The CoderGen framework makes great strides in evaluating code generation skills, but it currently has some limitations. First, it mainly uses a dataset on AI specific tasks, so it needs more testing to see if it works well for other types of software development. Second, even though we improve the AICoder model with the codellama dataset, it could still be better, and we need to keep working on it. Lastly, our testing method is simple and needs to be more robust for testing, possibly by using Docker and cloud platforms to make it easier for others to repeat our tests and build on our work.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA. Curran Associates Inc.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1219–1231, New York, NY, USA. Association for Computing Machinery.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*.

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.

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Alex Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. 2023. Learning math reasoning from self-sampled correct and partially-correct solutions. In *The 2023 International Conference on Learning Representations (2023 ICLR)*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan,

Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, UIST '23, New York, NY, USA. Association for Computing Machinery.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis.

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.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *ArXiv*, abs/2302.04761.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI tasks with chatGPT and its friends in hugging face. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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.

Zekun Wang, Ge Zhang, Kexin Yang, Ning Shi, Wangchunshu Zhou, Shaochun Hao, Guangzheng Xiong, Yizhi Li, Mong Yuan Sim, Xiuying Chen, Qingqing Zhu, Zhenzhu Yang, Adam Nik, Qi Liu, Chenghua Lin, Shi Wang, Ruibo Liu, Wenhu Chen, Ke Xu, Dayiheng Liu, Yike Guo, and Jie Fu. 2023. Interactive natural language processing.

BigScience Workshop, :, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, Dragomir Radev, Eduardo González Ponferrada, Efrat Levkovizh, Ethan Kim, Eyal Bar Natan, Francesco De Toni, Gérard Dupont, Germán Kruszewski, Giada Pistilli, Hady Elsahar, Hamza Benyamina, Hieu Tran, Ian Yu, Idris Abdulmumin, Isaac Johnson, Itziar

Gonzalez-Dios, Javier de la Rosa, Jenny Chim, Jesse Dodge, Jian Zhu, Jonathan Chang, Jörg Frohberg, Joseph Tobing, Joydeep Bhattacharjee, Khalid Almubarak, Kimbo Chen, Kyle Lo, Leandro Von Werra, Leon Weber, Long Phan, Loubna Ben allal, Ludovic Tanguy, Manan Dey, Manuel Romero Muñoz, Maraim Masoud, María Grandury, Mario Šaško, Max Huang, Maximin Coavoux, Mayank Singh, Mike Tian-Jian Jiang, Minh Chien Vu, Mohammad A. Jauhar, Mustafa Ghaleb, Nishant Subramani, Nora Kassner, Nurulaqilla Khamis, Olivier Nguyen, Omar Espejel, Ona de Gibert, Paulo Villegas, Peter Henderson, Pierre Colombo, Priscilla Amuok, Quentin Lhoest, Rheza Harliman, Rishi Bommasani, Roberto Luis López, Rui Ribeiro, Salomey Osei, Sampo Pyysalo, Sebastian Nagel, Shamik Bose, Shamsuddeen Hassan Muhammad, Shanya Sharma, Shayne Longpre, Somaieh Nikpoor, Stanislav Silberberg, Suhas Pai, Sydney Zink, Tiago Timponi Torrent, Timo Schick, Tristan Thrush, Valentin Danchev, Vassilina Nikoulina, Veronika Laippala, Violette Lepercq, Vrinda Prabhu, Zaid Alyafeai, Zeerak Talat, Arun Raja, Benjamin Heinzerling, Chenglei Si, Davut Emre Taşar, Elizabeth Salesky, Sabrina J. Mielke, Wilson Y. Lee, Abheesht Sharma, Andrea Santilli, Antoine Chaffin, Arnaud Stiegler, Debajyoti Datta, Eliza Szczechla, Gunjan Chhablani, Han Wang, Harshit Pandey, Hendrik Strobelt, Jason Alan Fries, Jos Rozen, Leo Gao, Lintang Sutawika, M Saiful Bari, Maged S. Al-shaibani, Matteo Manica, Nihal Nayak, Ryan Teehan, Samuel Albanie, Sheng Shen, Srulik Ben-David, Stephen H. Bach, Taewoon Kim, Tali Bers, Thibault Fevry, Trishala Neeraj, Urmish Thakker, Vikas Raunak, Xiangru Tang, Zheng-Xin Yong, Zhiqing Sun, Shaked Brody, Yallow Uri, Hadar Tojarieh, Adam Roberts, Hyung Won Chung, Jaesung Tae, Jason Phang, Ofir Press, Conglong Li, Deepak Narayanan, Hatim Bourfoune, Jared Casper, Jeff Rasley, Max Ryabinin, Mayank Mishra, Minjia Zhang, Mohammad Shoeybi, Myriam Peyrounette, Nicolas Patry, Nouamane Tazi, Omar Sanseviero, Patrick von Platen, Pierre Cornette, Pierre François Lavallée, Rémi Lacroix, Samyam Rajbhandari, Sanchit Gandhi, Shaden Smith, Stéphane Requena, Suraj Patil, Tim Dettmers, Ahmed Baruwa, Amanpreet Singh, Anastasia Cheveleva, Anne-Laure Ligozat, Arjun Subramonian, Aurélie Névéol, Charles Lovering, Dan Garrette, Deepak Tunuguntla, Ehud Reiter, Ekaterina Taktasheva, Ekaterina Voloshina, Eli Bogdanov, Genta Indra Winata, Hailey Schoelkopf, Jan-Christoph Kalo, Jekaterina Novikova, Jessica Zosa Forde, Jordan Clive, Jungo Kasai, Ken Kawamura, Liam Hazan, Marine Carpuat, Miruna Clinciu, Najoung Kim, Newton Cheng, Oleg Serikov, Omer Antverg, Oskar van der Wal, Rui Zhang, Ruochen Zhang, Sebastian Gehrmann, Shachar Mirkin, Shani Pais, Tatiana Shavrina, Thomas Scialom, Tian Yun, Tomasz Limisiewicz, Verena Rieser, Vitaly Protasov, Vladislav Mikhailov, Yada Pruksachatkun, Yonatan Belinkov, Zachary Bamberger, Zdeněk Kasner, Alice Rueda, Amanda Pestana, Amir Feizpour, Ammar Khan, Amy Faranak, Ana Santos, Anthony Hevia, Antigona Unldreaj, Arash Aghagol, Arezoo Abdollahi, Aycha Tammour, Azadeh HajiHosseini, Bahareh Behroozi, Benjamin Ajibade, Bharat Saxena, Carlos Muñoz Ferrandis, Daniel McDuff, Danish Contractor, David Lansky, Davis David, Douwe Kiela, Duong A. Nguyen, Edward Tan, Emi Baylor, Ezinwanne Ozoani, Fatima Mirza, Frankline Ononiwu, Habib Rezanejad, Hessie Jones, Indrani Bhattacharya, Irene Solaiman, Irina Sedenko, Isar Nejadgholi, Jesse Passmore, Josh Seltzer, Julio Bonis Sanz, Livia Dutra, Mairon Samagaio, Maraim Elbadri, Margot Mieskes, Marissa Gerchick, Martha Akinlolu, Michael McKenna, Mike Qiu, Muhammed Ghauri, Mykola Burynok, Nafis Abrar, Nazneen Rajani, Nour Elkott, Nour Fahmy, Olanrewaju Samuel, Ran An, Rasmus Kromann, Ryan Hao, Samira Alizadeh, Sarmad Shubber, Silas Wang, Sourav Roy, Sylvain Viguier, Thanh Le, Tobi Oyebade, Trieu Le, Yoyo Yang, Zach Nguyen, Abhinav Ramesh Kashyap, Alfredo Palasciano, Alison Callahan, Anima Shukla, Antonio Miranda-Escalada, Ayush Singh, Benjamin Beilharz, Bo Wang, Caio Brito, Chenxi Zhou, Chirag Jain, Chuxin Xu, Clémentine Fourrier, Daniel León Periñán, Daniel Molano, Dian Yu, Enrique Manjavacas, Fabio Barth, Florian Fuhrimann, Gabriel Altay, Giyaseddin Bayrak, Gully Burns, Helena U. Vrabec, Imane Bello, Ishani Dash, Jihyun Kang, John Giorgi, Jonas Golde, Jose David Posada, Karthik Rangasai Sivaraman, Lokesh Bulchandani, Lu Liu, Luisa Shinzato, Madeleine Hahn de Bykhovetz, Maiko Takeuchi, Marc Pàmies, Maria A Castillo, Marianna Nezhurina, Mario Sänger, Matthias Samwald, Michael Cullan, Michael Weinberg, Michiel De Wolf, Mina Mihaljcic, Minna Liu, Moritz Freidank, Myungsun Kang, Natasha Seelam, Nathan Dahlberg, Nicholas Michio Broad, Nikolaus Muellner, Pascale Fung, Patrick Haller, Ramya Chandrasekhar, Renata Eisenberg, Robert Martin, Rodrigo Canalli, Rosaline Su, Ruisi Su, Samuel Cahyawijaya, Samuele Garda, Shlok S Deshmukh, Shubhanshu Mishra, Sid Kiblawi, Simon Ott, Sinee Sang-aroonsiri, Srishti Kumar, Stefan Schweter, Sushil Bharati, Tanmay Laud, Théo Gigant, Tomoya Kainuma, Wojciech Kusa, Yanis Labrak, Yash Shailesh Bajaj, Yash Venkatraman, Yifan Xu, Yingxin Xu, Yu Xu, Zhe Tan, Zhongli Xie, Zifan Ye, Mathilde Bras, Younes Belkada, and Thomas Wolf. 2023. Bloom: A 176b-parameter open-access multilingual language model.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. 2023. The rise and potential of large language model based agents: A survey.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New

York, NY, USA. Association for Computing Machinery.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

A Appendix

A.1 Prompt Details

Task Prompt

```
1. Please design a requirement that can be described in one sentence.
2. Based on the above description, generate code to implement the requirement.
3. Function comments should follow the Google Python Style Guide, including args, returns, and raises.
4. Write corresponding test functions based on the generated code.
5. The test cases should be three examples of different difficulty levels, e.g., the first one verifies that the function executes normally, the second verifies that incorrect inputs are handled properly, and thethird verifies that the function returns the correct value.
6. For testing purposes, read image and audio files, download them from online resources to the local machine, or obtain them from datasets; do not provide fake or non-existent file addresses.
```

Import example

```
import subprocess
requirements = ["package1", "package2"]
for package in requirements:
    subprocess.run(['pip', 'install', '-U', package])
```

Test prompt

```
1. The function starts by printing "Testing started."
2. For images or audio, load a dataset or download data from online resources.
3. The test case starts by printing "Testing case [x/x] started", prints "succeeded" on success, and "failed" on failure.
4. The function ends by printing "Testing finished."
```

Test example

```
def test_...():
    print("Test started.")
    dataset = load_dataset("...")
   sample_data = dataset[0] # Extract a sample from the dataset

    # Test case 1:...
    print("Test case [1/3] started.")
    try:
        assert assert 1, f"Test case [1/3] failed: ..."
        print(f"Test case [1/3] succeeded: ...")
    except Exception as e::
        print(f"Test case [1/3] failed: ...\nerror:", e)

    # Test case 2:...

    # Test case 3:...

# Run the test function
test_...()
```

Table 6: Prompt details of GPT-4 dataset generation. Combining all parts from table into a complete prompt enables GPT-4 to convert domain documents into an executable code dataset.