ReflectionCoder: Learning from Reflection Sequence for Enhanced One-off Code Generation

Houxing Ren[1] **Mingjie Zhan**[2*] **Zhongyuan Wu**[2] **Aojun Zhou**[1]
Junting Pan[1,3] **Hongsheng Li**[1,3,4*]
[1]CUHK MMLab [2]SenseTime Research [3]CPII under InnoHK [4]Shanghai AI Laboratory
renhouxing@gmail.com zhanmingjie@sensetime.com hsli@ee.cuhk.edu.hk

Abstract

Code generation plays a crucial role in various tasks, such as code auto-completion and mathematical reasoning. Previous work has proposed numerous methods to enhance code generation performance, including integrating feedback from the compiler. Inspired by this, we present ReflectionCoder, a novel approach that effectively leverages reflection sequences constructed by integrating compiler feedback to improve one-off code generation performance. Furthermore, we propose reflection self-distillation and dynamically masked distillation to effectively utilize these reflection sequences. Extensive experiments on three benchmarks, *i.e.,* HumanEval (+), MBPP (+), and MultiPL-E, demonstrate that models fine-tuned with our method achieve state-of-the-art performance. Beyond the code domain, we believe this approach can benefit other domains that focus on final results and require long reasoning paths. Code and data are available at `https://github.com/SenseLLM/ReflectionCoder`.

1 Introduction

Code generation aims to automatically produce code based on natural language description, significantly saving developers time and reducing human error. In the past few decades, a lot of research has been conducted for code modeling, such as CodeBert (Feng et al., 2020), CodeT5 (Wang et al., 2021). Recently, Large Language Models (LLMs) have shown impressive modeling ability on natural language that allows them to perform various difficult tasks (OpenAI, 2023). By training on code domain datasets, LLMs such as CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), and DeepSeek-Coder (Guo et al., 2024), which can accurately understand user intents and generate code, have shown better performance on code-related tasks. Leveraging this powerful capability, various works empower LLMs in complex tasks including solving mathematics problems and logic reasoning by integrating code and its execution result as Chain-of-Thoughts (CoTs), such as PAL (Gao et al., 2023) and PoT (Chen et al., 2022).

Since code generation is important in various code-related tasks and many reasoning tasks, many previous studies focus on achieving better code generation performance. Integrating feedback from the compiler is an intuitive way to help the model reflect on previous mistakes and generate better code. For instance, Self-Debug (Chen et al., 2023) suggested that code LLMs be instructed to generate code, execute it, and subsequently improve the code quality based on its execution results. Additionally, Print-Debug (Hu et al., 2024) proposed to insert print statements to generate more detailed logs for debugging purposes. Furthermore, OpenCodeInterpreter (Zheng et al., 2024) incorporated simulated human feedback into the interaction. These studies have demonstrated that incorporating reflection sequences of code generation, execution, and analysis as CoTs can enhance the performance of code LLMs.

Inspired by these works, we propose to leverage the reflection sequences to guide the fine-tuning of code LLMs. The proven effectiveness of reflection sequences as CoTs in enhancing the code generation performance demonstrates their inherent knowledge, which can guide model fine-tuning and result in better one-off code generation performance. However, at least two challenges must be considered when using the reflection sequences to guide the model fine-tuning. Firstly, the reflection sequences differ from the vanilla one-off code generation. Most of the codes in the reflection sequences are partly modified based on previous codes, while all codes are completed in the inference stage. The gap between the training and inference stages results in relatively low utilization

*Corresponding author.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 9999–10020
July 27 - August 1, 2025

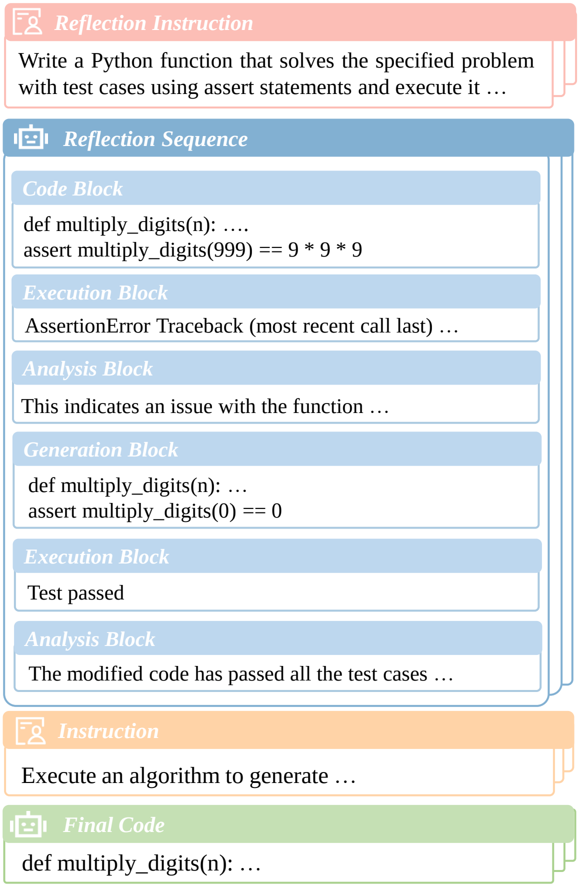


Figure 1: A sample of reflection sequence data containing four components: Reflection Instruction, Reflection Sequences, Instruction, and Final code.

of the reflection sequence. Secondly, most of the codes in reflection sequence are generated based on previous executions and analysis, whereas a one-off generation relies solely on a single instruction. This disparity makes it challenging to transition between such different prompts effectively.

Based on these concerns, we proposed ReflectionCoder, a novel method to effectively leverage reflection sequence to perform better in one-off code generation tasks. To bridge the gap between the reflection sequences and the vanilla code generation, we propose *reflection self-distillation*. Specifically, we carefully design a two-stage prompt to obtain high-quality instruction answer pairs with the same format as one-off generations. We first employ an LLM to generate a reflection sequence for an instruction with a compiler, and then task it to re-answer the instruction based on this sequence. After that, as shown in Figure 1, we obtain two rounds of dialogue as [Reflection Instruction, Reflection Sequence, Instruction, Final code]. The second round dialogue is the same as the one-off generation but with higher quality, which can play the role of a teacher sample distilling knowledge into one-off code generation. To effectively distill knowledge from reflection sequence to one-off generation, we design a novel distillation method, namely *dynamically masked distillation*. Specifically, with a particular LLM, the teacher input is the entire two-round dialogue, while the student input is a partly masked first-round dialogue along with an intact second-round dialogue. During the training process, we gradually increase the masking rate to progressively enhance the difficulty of generating the final code. In this way, LLM can be distilled to generate the final code from easy to difficult and achieve better performance.

Our contributions are summarized as follows:

- We propose to leverage reflection sequences to improve the one-off code generation performance of code LLMs, which can be generated by LLMs and thus save annotation costs.
- On top of the idea, we propose two techniques, namely *reflection self-distillation* and *dynamically masked distillation*, which can effectively utilize the reflection sequence to improve the one-off code generation performance.
- Extensive experiments on HumanEval (+), MBPP (+), MultiPl-E, APPs, LiveCodeBench, ClassEval, and BigCodeBench demonstrate the effectiveness of the proposed method on one-off code generation. Notably, ReflectionCoder-DeepSeek-Coder-33B reaches 82.9 (76.8) on HumanEval (+) and 84.1 (72.0) on MBPP (+), which is an on-par performance of Claude-3-opus and surpasses early GPT-4.

2 Related Work

2.1 Large Language Models for Code

Large Language Models (Ouyang et al., 2022; OpenAI, 2023; Anil et al., 2023b; Touvron et al., 2023a,b; Penedo et al., 2023; Yang et al., 2023; Bai et al., 2023; Jiang et al., 2023, 2024; Anil et al., 2023a; Anthropic, 2024) have proven highly effective in general natural language processing (NLP) tasks. For a specific domain such as code-related tasks (Chen et al., 2021; Austin et al., 2021; Bavarian et al., 2022; Muennighoff et al., 2023), training on large specific domain datasets can greatly improve their efficacy. Recent studies have introduced several LLMs for the code domain. OpenAI introduced Codex (Chen et al., 2021), and

Google introduced PaLM-Coder (Chowdhery et al., 2023). However, these models are closed-source, and we can only access them via API without access to their parameters. There are also several open-source LLMs for the code domain, such as CodeGen (Nijkamp et al., 2023), Incoder (Fried et al., 2023), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), StarCoder-2 (Lozhkov et al., 2024), CodeGeeX (Zheng et al., 2023), Code Llama (Rozière et al., 2023), and DeepSeek-Coder (Guo et al., 2024). In addition to vanilla code snippets, modification content of code with commit messages (Muennighoff et al., 2023) and code structure (Gong et al., 2024) are also proposed to be the pre-train corpus. After instruction tuning, some of these open-source models have outperformed several closed-source models (Luo et al., 2023).

2.2 Instruction Tuning for Code

The primary objective of instruction tuning is training LLMs to align with human instructions by using a large corpus of human instructions together with corresponding responses (Sanh et al., 2022; Wei et al., 2022; Ouyang et al., 2022; Longpre et al., 2023; Zhang et al., 2023). Fine-tuning upon this method, LLMs can directly follow user instructions without extra demonstration and improve their generalization capacity. Its great value is also demonstrated in code-related applications. For example, Code Alpaca (Chaudhary, 2023) applied SELF-INSTRUCT (Wei et al., 2022) to fine-tune LLMs with ChatGPT-generated instructions. WizardCoder (Luo et al., 2023) proposed Code Evol-Instruct, which evolves Code Alpaca data using the ChatGPT to generate more complex and diverse datasets. PanGu-Coder2 (Shen et al., 2023) proposed Rank Responses to align Test&Teacher Feedback framework, which uses ranking responses as feedback instead of the absolute value of a reward model. In addition to starting with instructions, a lot of work starts with existing source code. For example, MagiCoder (Wei et al., 2023), WaveCoder (Yu et al., 2023), and InverseCoder (Wu et al., 2024) proposed some methods to make full use of source code.

2.3 Iterative Generation and Refinement

Iterative refinement approaches are often taken to improve the generation quality. Recently, Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) demonstrated that LLMs can reflect on previous generations, generate feedback, and give better generations based on feedback. In the code domain, several tools can provide feedback for generated code, such as compiler, and other static tools. Integrating feedback from these tools can help the LLMs better reflect on themselves and generate better codes. For example, Self-Debugging (Chen et al., 2023) and Print-Debugging (Hu et al., 2024) proposed to integrate the execution result of the code as a feedback message to obtain better performance. StepCoder (Dou et al., 2024) and OpenCodeInterpreter (Zheng et al., 2024) involved executing and iteratively refining code as multi-turn interactions into instruction tuning, improving the model's debugging ability. Concurrently, AutoCoder (Lei et al., 2024) employed multi-turn interaction to obtain high-quality instruction data and then improve the one-off generation performance. In contrast, our method method introduces the reflection sequence into the training stage instead of just using it to filter the data.

3 Methodology

In this section, we present the methodological details of the proposed ReflectionCoder. We begin with a vanilla distillation, followed by a carefully designed method that comprehensively extracts knowledge from the reflection sequences and guides the model training.

3.1 Reflection Self-Distillation

Here, we present how to utilize the reflection sequences to enhance the fine-tuning of code LLMs. As presented in Section 1, a piece of reflection sequence data includes four components: [Reflection Instruction, Reflection Sequence, Instruction, Final code], where the reflection sequence is divided into three types of blocks, namely code block, execution block, and analysis block. Their contents are the generated executable code, the execution results, and the code summary or error analysis, respectively.

We construct two input samples for each reflection sequence to perform the reflection self-distillation. The teacher sample is the entire reflection sequence, and the student sample consists of [Instruction, Final Code], which is the same as vanilla one-off code generation instruction tuning data. The key distinction between them is that the final code of the teacher sample can be generated based on the reflection sequences with low perplexity, while the student sample can only be generated

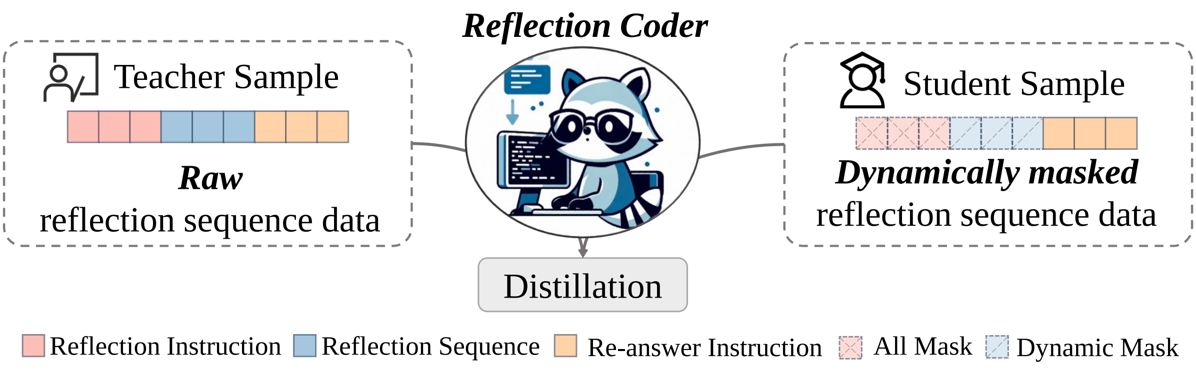


Figure 2: Overview of the proposed dynamically masked distillation.

according to the instruction. The vanilla distillation loss can be formulated as

$$\mathcal{L}_d^s = \mathrm{KL}\left(p(t_c|t_{ri}, t_{rs}, t_i) \parallel p(t_c|t_i)\right), \quad (1)$$

where t_c denotes tokens of the final code, t_{ri} denotes tokens of the reflection instruction, t_{rs} denotes tokens of the reflection sequence, and t_i denotes tokens of the instruction.

This approach enables the distillation of knowledge from the sequence into a one-off generation. The absolute position of the tokens in [Instruction, Final Code] differs between the teacher sample and the student sample, while [Reflection Instruction, Reflection Sequence] exists in the teacher sample but not in the student sample. However, the relative positions between the two tokens in [Instruction, Final Code] are the same between the teacher sample and the student sample, which indicates that distillation is effective for models utilizing Rotary Position Embedding (Su et al., 2024), such as Llama (Touvron et al., 2023b).

3.2 Dynamically Masked Distillation

Although vanilla distillation can distill knowledge from reflection sequence to enhance the one-off code generation, it could be hindered by the negative impact of contextual differences. Previous studies on distillation show that a student model distilled from a teacher with more parameters performs worse than the one distilled from a smaller teacher with a smaller capacity (Mirzadeh et al., 2020). This finding suggests that the difference between teacher and student should not be too large. However, a significant gap exists between our teacher-student sample pair, as the teacher sample contains the entire reflection sequence while the student sample has no access to the reflection procedure. This discrepancy could lead to the poor performance of vanilla distillation.

Inspired by Curriculum Learning (Bengio et al., 2009), we carefully design a *dynamically masked distillation* method. The overall procedure is presented in Figure 2. The initial student sample is the same as the teacher sample. During the training process, we mask all tokens of the "Reflection Instruction" and a portion of tokens of the "Reflection Sequence". The number of masked tokens is gradually increased to progressively enhance the difficulty of generating the final code, thereby enabling the model to effectively learn the knowledge encoded in the reflection sequence. Then the distillation loss can be formulated as

$$\mathcal{L}_d = \mathrm{KL}\left(p(t_c|t_{ri}, t_{rs}, t_i) \parallel p(t_c|t_{prs}, t_i)\right), \quad (2)$$

where t_{prs} denoted tokens partly masked reflection instruction.

As shown in Figure 3, we design three dynamic masking strategies, namely *random mask*, *sequential mask*, and *block mask*. All of these strategies adjust dynamically with the mask rate, a concept related to the training process, which can be defined as "current step/max step". The masking details are illustrated below:

(1) *Random mask* selects blocks to mask based on the mask rate randomly. This is an intuitive strategy used by many previous studies in the pre-training stage such as BERT (Devlin et al., 2019) and T5 (Raffel et al., 2020).

(2) *Sequential mask* selects the leftmost blocks to mask and gradually expands the masked scope according to the mask rate. The underlying principle of this strategy is that later tokens are usually more influential in generating the final

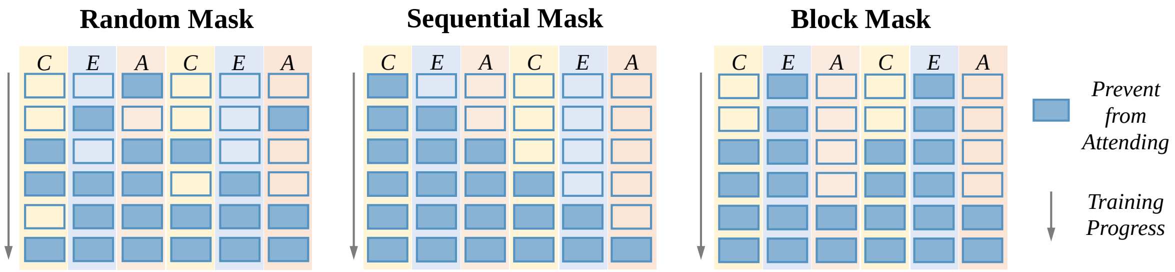


Figure 3: Overview of the proposed dynamic masking strategies. Here, a cell denotes a block, 'C' denotes the code block, 'E' denotes the execution block, and 'A' denotes the analysis block.

code since code generated after analysis tends to be more accurate than those generated initially.

(3) *Block mask* selects some blocks according to mask rates. Specifically, when the mask rate exceeds 0, all execution blocks are masked. When the mask rate exceeds $1/3$, all generation blocks are additionally masked. When the mask rate exceeds $2/3$, all analysis blocks are further masked. The core idea of this strategy is that the effectiveness of tokens is block-dependent. For instance, tokens in the execution block typically have the lowest impact.

With these dynamically masked strategies, the learning difficulty gradually increases, contributing to better final one-off code generation performance. Similar to *reflection self-distillation*, the absolute position of tokens in the one-off code generation round differs between the training stage and the inference stage, while "Reflection Sequence" exists in the training stage but not in the inference stage. However, the relative positions of the two tokens in [Instruction, Final Code] remain the same between the training stage and the inference stage, which indicates that there is no gap between the training stage and the inference stage for models utilizing Rotary Position Embedding (Su et al., 2024).

Training loss. We employ both the next token prediction loss and distillation loss to train the model. For the teacher sample, we perform the next token prediction task on "Final Code" and the text blocks and the code blocks of "Reflection Sequence", because both the queries of the user and the execution results do not need to be generated in the inference stage. For the student sample, we only perform the next token prediction task on "Final Code". The final loss consists of the next token prediction loss of the teacher and student samples, and the distillation loss between the teacher and student sample.

4 Experiments

4.1 Experimental Setup

Training Dataset. Our training dataset includes a vanilla code instruction tuning dataset, where each sample contains an instruction and corresponding code answer and the proposed reflection sequence dataset. For the code instruction tuning dataset, we use instruction answer pairs from an open-source code instruction tuning dataset: CodeFeedback-Filtered-Instruction[1]. For the reflection sequence dataset, we first randomly select 10k instructions with Python code in the corresponding answer to conduct two rounds of dialogue with GPT-4 Code Interpreter[2], obtaining the reflection sequence dataset. Subsequently, we use the 10k reflection sequence data and 156k code instruction tuning data to fine-tune DeepSeek-Coder 33B (Guo et al., 2024). Using this fine-tuned model, we generate additional 12k reflection sequence data. The detailed data construction process is presented in Appendix A. Finally, we fine-tune the target model using 22k reflection sequence data and 156k code instruction tuning data.

Test Dataset. We evaluate our method on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), two of the most widely used benchmarks for code generation. Each task in these benchmarks includes a task description as the prompt and a handful of test cases to check the correctness of the LLM-generated code. Considering the insufficiency of test cases in these benchmarks, Liu et al. (2023) proposed HumanEval+ and MBPP+, which contain 80×/35× more tests.

[1]https://huggingface.co/datasets/m-a-p/CodeFeedback-Filtered-Instruction

[2]https://platform.openai.com/docs/assistants/tools/code-interpreter

Method	Base	Benchmark			
		HumanEval	HumanEval+	MBPP	MBPP+
Closed-Source Models					
O1-Preview (Sept 2024) (Jaech et al., 2024)	-	96.3	89.0	95.5	80.2
GPT-4-Turbo (April 2024) (OpenAI, 2023)	-	90.2	86.6	-	-
GPT-4-Turbo (Nov 2023) (OpenAI, 2023)	-	88.4	81.7	85.7	73.3
GPT-3.5-Turbo (Nov 2023) (Ouyang et al., 2022)	-	76.8	70.7	82.5	69.7
Claude-3-opus (Mar 2024) (Anthropic, 2024)	-	82.9	76.8	89.4	73.3
Claude-3-sonnet (Mar 2024) (Anthropic, 2024)	-	70.7	62.8	83.6	69.3
Mistral Large (Mar 2024) (Jiang et al., 2023)	-	70.1	62.8	72.8	59.5
Gemini Pro 1.0 (Anil et al., 2023a)	-	63.4	55.5	75.4	61.4
Open-Source Models					
WizardCoder (Luo et al., 2023)	CL-7B	48.2	40.9	58.5	49.5
MagiCoder-S (Wei et al., 2023)	CL-7B	70.7	66.5	70.6	60.1
OpenCodeInterpreter (Zheng et al., 2024)	CL-7B	72.6	67.7	66.4	55.4
ReflectionCoder	CL-7B	**75.0**	**68.9**	**72.2**	**61.4**
WizardCoder (Luo et al., 2023)	CL-34B	73.2	64.6	75.1	63.2
OpenCodeInterpreter (Zheng et al., 2024)	CL-34B	78.0	72.6	73.4	61.4
Speechless (Speechless, 2023)	CL-34B	77.4	72.0	73.8	61.4
ReflectionCoder	CL-34B	**78.0**	**73.8**	**80.2**	**67.5**
DeepSeek-Coder-Instruct (Guo et al., 2024)	DS-6.7B	73.8	70.1	74.9	65.6
MagiCoder-S (Wei et al., 2023)	DS-6.7B	76.8	70.7	69.4	69.0
OpenCodeInterpreter (Zheng et al., 2024)	DS-6.7B	77.4	73.8	76.5	66.4
Artigenz-Coder (Artigenz-Coder, 2024)	DS-6.7B	75.6	72.6	80.7	**69.6**
ReflectionCoder	DS-6.7B	**80.5**	**74.4**	**81.5**	**69.6**
DeepSeek-Coder-Instruct (Guo et al., 2024)	DS-33B	81.1	75.0	80.4	70.1
WizardCoder (Luo et al., 2023)	DS-33B	79.9	73.2	81.5	69.3
OpenCodeInterpreter (Zheng et al., 2024)	DS-33B	79.3	73.8	80.2	68.5
ReflectionCoder	DS-33B	**82.9**	**76.8**	**84.1**	**72.0**

Table 1: Pass@1 accuracy on HumanEval(+) and MBPP(+). Here, 'CL' denotes Code Llama, and 'DS' denotes DeepSeek-Coder. The best results of each base are in bold and results unavailable are left blank.

Following prior work (Liu et al., 2023; Wei et al., 2023; Zheng et al., 2024), we use greedy decoding to generate one sample and focus on comparing the pass@1 metric. Due to the limited space, we present evaluation experiments on more code-related benchmarks in Appendix B, including MultilPL-E (Cassano et al., 2022), DS1000 (Lai et al., 2023), APPs (Hendrycks et al., 2021a), LiveCodeBench (Jain et al., 2024), ClassEval (Du et al., 2023), and BigCodeBench (Zhuo et al., 2024).

Implementation Details. We test our methods on Code Llama Python 7B/34B and DeepSeek-Coder 6.7B/33B. We finetune all models for 2 epochs. We employ AdamW (Loshchilov and Hutter, 2019) optimizer with a learning rate of 5e-5 for 6.7B/7B models and 2e-5 for 33B/34B models, a 0.05 warm-up ratio, and a cosine scheduler. We set the batch size as 512 and the max sequence length as 4096. To efficiently train the computationally intensive models, we simultaneously employ DeepSpeed (Rajbhandari et al., 2020) and Flash Attention (Dao, 2023). On 16 NVIDIA A800 80GB GPUs, the experiments on 7B models and 34B models take 3.5 hours and 25 hours, respectively.

In the training process, we up-sample 22k reflection sequence data by a factor of 2 and mix them with 156k code instruction tuning data. For samples in code instruction tuning data, we only employ the next token prediction as the training task, *a.k.a.,* we only calculate the causal language model loss. For samples in reflection sequence data, we use the proposed method to calculate the loss. We only use the block mask strategy in the order of execution block, analysis block, and code block. Although each strategy can bring benefits, mixing them is no longer beneficial in the experiments.

4.2 Evaluation

Baselines. We compare ReflectionCoder with previous state-of-the-art methods, including WizardCoder (Luo et al., 2023), Speechless (Speechless, 2023), DeepSeek-Coder Instruct (Guo et al., 2024), Magicoder (Wei et al., 2023), and OpenCodeInterpreter (Zheng et al., 2024). All the results are consistently reported from the EvalPlus leaderboard[3]. The proposed method is an instruction tuning method, so we do not present comparison results for base models such as StarCoder (Li et al., 2023) and Code Llama (Rozière et al., 2023).

Results. Table 1 shows the pass@1 accuracy of different method on HumanEval (+) and MBPP (+). Based on the results, we have the following findings: (1) For open-source methods with parameters ranging from 6.7B to 34B, the proposed ReflectionCoder outperforms previous state-of-the-art methods on all base models, demonstrating its effectiveness. (2) Focusing on Code Llama, ReflectionCoder-CodeLlama-7B even surpasses WizardCode-CodeLlama-34B on HumanEval and HumanEval+. (3) Compared with OpenCodeInterpreter, ReflectionCoder performs better on various base models, which indicates that we take better advantage of the reflection sequences. (4) Compared with closed-source models, ReflectionCoder-DeepSeek-Coder-33B outperforms Gemini Pro, Mistral Large, and Claude-3-sonnet on all four benchmarks. It is worth noting that ReflectionCoder-DeepSeek-Coder-33B also achieves the on-par performance of GPT-3.5-Turbo and Claude-3-opus.

4.3 Detailed Analysis

Here, we conduct some analytical experiments. Due to the limited space, more analytical experiments are presented in the Appendix B.

4.3.1 Ablation Study

Here, we check how each component contributes to the final performance. We prepare three group variants of our method: (1) The first group is related to the high-level method, which has three variants. w/o Dynamically Mask denotes without any dynamically mask strategy, *a.k.a.,* the vanilla distillation. w/o Distillation denotes without distillation, *a.k.a.,* only perform next token prediction on the reflection data. w/o Reflection Sequence denotes without reflection sequence parts, *a.k.a.,* train the model on reflection data but without reflection sequences. w/o Reflection Data denotes without reflection data, *a.k.a.,* only train the model with code instruction tuning data. (2) The second group is related to the source of the reflection data. w/o GPT-4 Data denotes only use the 12k reflection data construct from the fine-tuned DeepSeek-Coder 33B. Note that the DeepSeek-Coder 33B is fine-tuned with reflection Data from GPT-4. w/o DS Data only use the 10k reflection data construct from GPT-4. (3) The third group is related to the masking strategy. w/ Random Mask and w/ Sequential Mask denote replacing the block mask with random and sequential masks, respectively. w/ Three Mask Strategies denotes randomly selecting a masking strategy in each step.

[3] `https://evalplus.github.io/leaderboard.html`

Method	HumanEval (+)	MBPP (+)
ReflectionCoder	**75.0 (68.9)**	**72.2 (61.4)**
w/o Dynamic Mask	70.7 (65.2)	70.4 (58.5)
w/o Distillation	69.5 (63.4)	70.4 (59.0)
w/o Reflection Sequence	66.5 (62.2)	68.5 (57.9)
w/o Reflection Data	65.9 (62.2)	68.5 (57.9)
w/o GPT Data	71.3 (67.1)	70.1 (59.5)
w/o DS Data	68.9 (65.2)	69.6 (58.2)
w/ Random Mask	72.0 (66.5)	70.1 (59.0)
w/ Sequential Mask	72.6 (67.7)	71.3 (60.3)
w/ Three Strategies	73.2 (65.9)	71.7 (61.2)

Table 2: Ablation results on HumanEval (+) and MBPP (+). The metric is Pass@1 accuracy, and all the results are based on Code Llama 7B.

Table 2 shows the pass@1 accuracy of different variants on HumanEval (+) and MBPP (+). As we can see, the performance ranking can be given as: w/o Reflection Data < w/o Distillation < w/o Dynamically Mask < ReflectionCoder. These results indicate that all components are essential for improving performance. Moreover, w/o Reflection Sequence and w/o Reflection Data are almost the same. The main reason is that w/o Reflection Sequence are the same as the instruction tuning data in format, which does not introduce new knowledge into the training. Additionally, both w/o GPT-4 Data and w/o DS Data perform worse than ReflectionCoder. And w/o GPT-4 Data performs better than w/o DS Data. A possible reason is that we have carried out strict filtering on Reflection Data from DS, which may impact the final performance. Finally, w/ Random Mask, w/ Sequential Mask, and w/ Three Mask Strategies perform better than w/o Dynamically Mask but worse than Reflection-

Method	HumanEval (+)	MBPP (+)
w/ EAC	**75.0 (68.9)**	**72.2 (61.4)**
w/ ECA	**75.0 (68.9)**	70.9 (59.5)
w/ ACE	72.0 (66.5)	70.6 (60.1)
w/ AEC	73.2 (65.9)	70.9 (59.5)
w/ CAE	71.3 (65.9)	70.4 (59.8)
w/ CEA	73.2 (67.1)	72.0 (60.8)

Table 3: Effect of masked order. The metric is Pass@1 accuracy, and all the results are based on Code Llama 7B. Here, 'C' denotes the code block, 'E' denotes the execution block, and 'A' denotes the analysis block. For example, 'ECA' denotes first mask execution block, then mask code block, and finally mask analysis block.

Model	GPT	33B	6.7B	HumanEval (+)	MBPP (+)
33B	✓	✗	✗	80.5 (73.8)	80.7 (69.0)
33B	✓	✓	✗	**82.9 (76.8)**	**84.1 (72.0)**
6.7B	✓	✓	✗	80.5 (74.4)	**81.5 (69.6)**
6.7B	✓	✗	✓	79.3 (76.2)	80.7 (68.8)
6.7B	✗	✓	✗	80.5 (75.0)	81.0 (68.3)
6.7B	✗	✗	✓	**81.1 (76.2)**	80.4 (68.3)

Table 4: Effect of data source. The metric is Pass@1 accuracy. Here, "33B" denotes Deepseek-Coder-33B and "6.7B" denotes Deepseek-Coder-6.7B.

Coder. This indicates that while the three strategies are effective, they are not fully compatible with each other. A possible reason is that mixing them destroys the curricular nature of learning, leading to reduced effectiveness.

4.3.2 Effect of Block Masked Order

As mentioned in Section 3, the block mask masks block in a specific order. Here, we examine the effect of masking order by preparing six variants with all possible orders.

Table 3 shows the pass@1 accuracy of different orders. As we can see, the two orders that mask execution blocks first perform better than other orders, indicating that tokens in execution blocks are generally less effective, which is intuitive. Similarly, the two orders that mask code blocks last also perform better, suggesting that tokens in code blocks are more effective.

4.3.3 Effect of Data Source

As mentioned in Section 4.1, our reflection sequence dataset is constructed from GPT-4 and fine-tuned Deepseek-Coder-33B. Here, we construct three sets of experiments to check the effectiveness of our method with different other data sources.

Firstly, we compared the ReflectionCoder-Deepseek-Coder-33B with the Deepseek-Coder-33B fine-tuned only with data from GPT-4, which is used to construct more data in our main experiments. As shown in the first group of Table 4, the intermediate model performs worse than the final model, which shows that the model can generate its training data and improve itself based on our method after only a small amount of training data from GPT-4.

Then, we employ the Deepseek-Coder-6.7B to act as the intermediate model. As shown in the second group of Table 4, show that the data generated by the DeepSeek-Coder 6.7B can still bring benefits. Surprisingly, for HumanEval, the Deepseek-Coder 6.7B fine-tuned with self-generated reflection sequence data achieves better performance. The results also show that GPT-4 data is not the key to improving model performance. As long as the model learns how to reflect based on execution results, it can generate a reflection sequence for the model to improve itself.

Model	HumanEval (+)	MBPP (+)
Llama-3.1-8B-Instruct	70.1 (62.2)	72.5 (59.3)
w/ reflection	**76.2** (64.7)	**74.2** (62.2)
w/ distillation	74.4 **(68.3)**	73.0 **(63.0)**
w/ distillation & reflection	74.4 (67.7)	72.2 (62.4)

Table 5: Experiment on Llama-3.1-8B-Instruct. The metric is Pass@1 accuracy. Here, "w/ reflection" denotes performing reflection while testing on Llama-3.1-8B-Instruct. "w/ distillation" denotes the one-off generation performance of the model fine-tuned with self-generated reflection sequence data. "w/ distillation & reflection" denotes performing reflection while testing on the model fine-tuned with the self-generated reflection sequence data.

4.3.4 Autonomous Enhancement

To completely exclude the factor of GPT-4, we employ an open-source model (Llama-3.1-8B-Instruct (Dubey et al., 2024)) that can generate reflection sequences without any training to act as the data source. We first employ reflection in testing as the reference, which first tasks the Llama-3.1-8B-Instruct to generate the reflection sequence and then tasks the model to generate the final code in the test stage. Then, we task the model to generate the reflection sequence data and use the generated data to fine-tune itself with the proposed method.

As shown in Table 5, although Llama-3.1-8B-Instruct has undergone multiple rounds of post-training (including SFT and multi-turn DPO), our method can still further improve its performance and only rely on the data generated by itself. Surprisingly, on the plus dataset, the proposed method even outperforms w/ reflection and w/ distillation & reflection. The reason is that the expected answers directly generated by the model have a high error rate, making it difficult to cover boundary data and more difficult data. Meanwhile, the data used for training has been strictly filtered (the filtered details are presented in Appendix A), so the quality of the data used for training is relatively high. The filter pass rate (only 17%) also shows that the quality of the generated data is relatively low.

4.4 Case Study

Here, we present a case to show the effectiveness of the proposed ReflectionCoder. Specifically, the instruction is:

```
def iscube(a):
    """
    Write a function that takes an
        integer a and returns True if
        this integer is a cube of some
        integer number. Note: you may
        assume the input is always valid
        .
    Examples:
    iscube(1) ==> True
    iscube(2) ==> False
    iscube(-1) ==> True
    iscube(64) ==> True
    iscube(0) ==> True
    iscube(180) ==> False
    """
```

We compare two generated implementations:

```
# Response of "w/o Reflection Data"
def iscube(a):
    cube_root = round(a ** (1. / 3))
    return cube_root ** 3 == a

# Response of ReflectionCoder
def iscube(a):
    if a < 0:
        a = -a
    cube_root = round(a ** (1. / 3.))
    return cube_root ** 3 == a
```

Although the solution generated by the "w/o Reflection Data" appears correct, it fails to account for the behavior of the "round" function when handling negative inputs. This oversight could lead to incorrect results for negative cube numbers. In contrast, the refined version—likely influenced by reflective feedback—correctly handles negative inputs by converting them to positive values before computing the cube root.

Model	GSM8K	MATH
w/o ReflectionCoder	9.9	9.6
w/ ReflectionCoder	**11.1**	**13.6**

Table 6: Experiments on two mathematical reasoning datasets. The metric is Pass@1 accuracy.

This case highlights a key advantage of leveraging reflection sequences: models can learn nuanced behaviors of library functions, such as "round", through feedback during the training process. Consequently, the model with access to reflection data demonstrates a deeper and more reliable understanding of function semantics.

4.5 Generalization

Here, we evaluate the generalization ability of our proposed methods. Specifically, we fine-tune a LLaMA-3.1-8B model using the training sets of GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021b). The baseline model is trained to directly predict the final answer without any chain-of-thought (CoT). For the proposed method, we regard the CoT as the reflection sequence and apply dynamic masking during distillation.

As shown in Table 6, even with minimal adaptation, our method yields consistent improvements on both datasets. These results demonstrate the potential applicability of our approach in non-code reasoning tasks. Furthermore, our method opens up the possibility for dynamic compression of reasoning paths, which may significantly reduce latency.

5 Conclusion

In this paper, we proposed ReflectionCoder, a novel method to effectively leverage the reflection sequence constructed by integrating feedback from the compiler to achieve better one-off code generation performance. We proposed two training techniques to effectively utilize the reflection sequences data, namely *reflection self-distillation* and *dynamically masked distillation*. The *reflection self-distillation* aims to distillation from reflection sequence to one-off code generation, and the *dynamically masked distillation* aims to utilize the reflection sequence to achieve better performance effectively. In the future, we plan to improve this method to dynamically reduce unnecessary reasoning paths for domains that need to show reasoning paths to simplify the model output.

Limitations

The primary limitation of this study is its reliance on a powerful model, such as the GPT-4 code interpreter, for constructing reflection sequence data. While this method ensures high precision and efficiency, it also incurs significant computational costs, which may limit its accessibility and scalability, particularly in resource-constrained environments. However, as large language models continue to evolve, open-source models like Llama 3.1 are beginning to exhibit similar capabilities. We anticipate that this limitation will diminish as these models become more advanced and widely available. Furthermore, the reliance on Rotary Position Embedding introduces an additional restriction. While effective within the specific context of this study, it may limit the method's generalizability and adaptability to different architectures or alternative embedding strategies.

Ethics Statement

The models utilized in this paper, StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), Deepseek-Coder (Guo et al., 2024) and Llama-3.1 (Dubey et al., 2024), are licensed for academic research purposes. Furthermore, the data employed in this study, Code Instruction Tuning Dataset[4], is collected from Magicoder-OSS-Instruct[5], Python code subset of ShareGPT[6], Magicoder-Evol-Instruct[7], and Evol-Instruct-Code[8]. All of these datasets are constructed from GPT-3 or GPT-4, while OpenAI permit on research access[9] and all theses datasets are licensed for research purposes.

Acknowledgment

This study was supported in part by the Centre for Perceptual and Interactive Intelligence (CPII) Ltd., a CUHK-led InnoCentre under the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong SAR Government, and in part by NSFC-RGC Project N_CUHK498/24. Hongsheng Li is a PI of CPII under the InnoHK.

[4]https://huggingface.co/datasets/m-a-p/CodeFeedback-Filtered-Instruction

[5]https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K

[6]https://huggingface.co/datasets/ajibawa-2023/Python-Code-23k-ShareGPT

[7]https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K

[8]https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1

[9]https://openai.com/policies/

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. 2023a. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernández Ábrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan A. Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vladimir Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, and et al. 2023b. Palm 2 technical report. *CoRR*, abs/2305.10403.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Artigenz-Coder. 2024. Artigenz-coder-ds-6.7b.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *CoRR*, abs/2309.16609.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *CoRR*, abs/2208.08227.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR*, abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *CoRR*, abs/2211.12588.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *CoRR*, abs/2304.05128.

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. 2023. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *CoRR*, abs/2402.01391.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *CoRR*, abs/2308.01861.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: program-aided language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: structure-aware pretraining for code generation and understanding. *CoRR*, abs/2401.03003.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021a. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024. Leveraging print debugging to improve code generation in large language models. *CoRR*, abs/2401.05319.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, and Ilge Akkaya. 2024. Openai o1 system card. *CoRR*, abs/2412.16720.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *CoRR*, abs/2401.04088.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. Autocoder: Enhancing code large language model with aiev-instruct. *CoRR*, abs/2405.14906.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. 2023. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 22631–22648. PMLR.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Seyed-Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. 2020. Improved knowledge distillation via teacher assistant. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5191–5198. AAAI Press.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124.

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. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Hamza Alobeidli, Alessandro Cappelli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. 2023. The refinedweb dataset for falcon LLM: outperforming curated corpora with web data only. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 20. IEEE/ACM.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask prompted training enables zero-shot task generalization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *CoRR*, abs/2307.14936.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Speechless. 2023. speechless-codellama-34b-v2.0.

Jianlin Su, Murtadha H. M. Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

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, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *CoRR*, abs/2310.03731.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *CoRR*, abs/2312.02120.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *CoRR*, abs/2407.05700.

Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Ce Bian, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, Fan Yang, Fei Deng, Feng Wang, Feng Liu, Guangwei Ai, Guosheng Dong, Haizhou Zhao, Hang Xu, Haoze Sun, Hongda Zhang, Hui Liu, Jiaming Ji, Jian Xie, Juntao Dai, Kun Fang, Lei Su, Liang Song, Lifeng Liu, Liyun Ru, Luyao Ma, Mang Wang, Mickel Liu, MingAn Lin, Nuolan Nie, Peidong Guo, Ruiyang Sun, Tao Zhang, Tianpeng Li, Tianyu Li, Wei Cheng, Weipeng Chen, Xiangrong Zeng, Xiaochuan Wang, Xiaoxi Chen, Xin Men, Xin Yu, Xuehai Pan, Yanjun Shen, Yiding Wang, Yiyu Li, Youxin Jiang, Yuchen Gao, Yupeng Zhang, Zenan Zhou, and Zhiying Wu. 2023. Baichuan 2: Open large-scale language models. *CoRR*, abs/2309.10305.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. A survey on language models for code. *CoRR*, abs/2311.07989.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *CoRR*, abs/2402.14658.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Solving challenging math word problems using GPT-4 code interpreter with code-based self-verification. *CoRR*, abs/2308.07921.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

Appendix

A Data Construction

As mentioned in Section 4.1, our reflection sequence data is constructed from GPT-4 Code interpreter and fine-tuned Deepseek-Coder-33B. Here, we present details of data construction.

A.1 GPT-4 Code Interpreter

Previous studies (Zhou et al., 2023; Wang et al., 2023) have revealed that GPT-4 Code Interpreter [10] can write and run Python code in a sandbox execution environment to solve challenging code and math problems. It can iterate on the incorrect code it had previously generated by analyzing the cause of the failure and regenerating the code until it executes successfully. Based on its capability, we designed a two-stage method to prompt the GPT-4 Code Interpreter to construct the reflection sequence dataset.

In the first stage, we task the GPT-4 Code Interpreter to generate code to solve the given problem and test the code with assert statements. If the code fails any of these tests, the GPT-4 Code Interpreter will analyze the reasons for failure and regenerate the code with necessary corrections automatically. In this way, we get a reflection sequence of code generation, execution, and analysis, as presented in the blue blocks in Figure 1. The prompt detail is shown below:

The first round prompt

Here is a programming problem for you to tackle:

(1) Write a Python function that solves the specified problem with craft test cases using assert statements and execute it. Pay special attention to edge cases to thoroughly validate your solution's correctness.

(2) If your code fails any of the tests, carefully examine the root cause of the failure. Make the necessary corrections to your code and then retest to confirm the fixes.

Note: At this phase, your primary goal is to ensure the reliability of your code. There's no need to delve into in-depth problem analysis or strive for code optimization.

 # Programming Problem
{problem}

In the second stage, we task the GPT-4 Code Interpreter to generate the entire code based on the preceding reflection sequence. Additionally, we instruct the model to refrain from using any words related to the preceding reflection sequence, effectively simulating the one-off code generation. In this way, we get the high-quality code answer, as presented in the green block in Figure 1. The prompt detail is shown below:

The second round prompt

Then, your task is to create a precise solution for the given programming problem.

Your answer should be complete and standalone, avoiding references to external resources or past exercises, and omit phrases such as "correct version".

There is no requirement to execute the code or provide any test/usage example. Just present the code for the given problem between "```python" and "```".

A.2 Deepseek-Coder-33B

Due to the high cost of calling the GPT-4 Code Interpreter, we only construct 10k reflection sequence data using the prompt provided in Section 3. To generate more reflection sequence data, as described in Section 4, We first fine-tune the DeepSeek-Coder 33B (Guo et al., 2024) model using 10k reflection sequence data and 156k code instruction tuning data, which endows it with the capability to generate code and interpret feedback from the compiler. Then, we use this fine-tuned model to construct more reflection sequence data.

In the constructing stage, we randomly select another 70k instructions, whose corresponding answers contain Python code, to prompt the fine-tuned model. The following steps are performed to implement the reflection process.

[10] https://platform.openai.com/docs/assistants/tools/code-interpreter

Model	Base	Java	JavaScript	C++	PHP	Swift	Rust
StarCoder	SC-15B	28.5	31.7	30.6	26.8	16.7	24.5
WizardCoder	SC-15B	35.8	41.9	39.0	39.3	33.7	27.1
Code Llama-Python	CL-7B	29.3	31.7	27.0	25.1	25.6	25.5
MagiCoder	CL-7B	36.4	45.9	36.5	39.5	33.4	30.6
MagiCoder-S	CL-7B	42.9	57.5	44.4	47.6	44.1	40.3
ReflectionCoder	CL-7B	**53.2**	**62.1**	**47.9**	**53.6**	**49.1**	**50.6**
Code Llama-Python	CL-34B	39.5	44.7	39.1	39.8	34.3	39.7
WizardCoder	CL-34B	44.9	55.3	47.2	47.2	44.3	46.2
ReflectionCoder	CL-34B	**61.4**	**70.7**	**63.2**	**65.7**	**55.8**	**64.0**

Table 7: Pass@1 accuracy results on MulitiPL-E. The best results of each base are in bold. Here, 'SC' denotes StarCoder, and 'CL' denotes Code Llama.

Model	Base	C++	Java	PHP	TS	C#	Bash	JavaScript
DS Instruct	DS-6.7B	63.4	**68.4**	**68.9**	67.2	**72.8**	36.7	**72.7**
ReflectionCoder	DS-6.7B	**69.5**	65.8	65.2	**70.8**	69.6	**42.4**	72.0
DS Instruct	DS-33B	68.9	**73.4**	**72.7**	67.9	74.1	43.0	**73.9**
ReflectionCoder	DS-33B	**70.8**	70.9	72.0	**72.3**	**74.7**	**45.6**	**73.9**

Table 8: Pass@1 accuracy results on MulitiPL-E. The best results of each base are in bold. Here, 'DS' denotes DeepSeek-Coder.

- First, we prompt the fine-tuned model to generate a code block, which contains code and test samples.
- Then, we employ a Jupyter Client to execute the code and concatenate the execution result to the prompt as an execution block.
- After that, the model generates an analysis block for the cause if the code sample fails any of the tests.
- The model will repeat the code generation and analyzing process until there is no error or it reaches a maximum of eight iterations.

We filter out 38k samples whose generated codes contain I/O operations that can be identified by keyword matching (*e.g.,* "open," "dump," "pip") or fail to resolve all errors within the maximum of eight iterations limitation. After that, we filter out samples that only contain one iteration, *i.e.,* the first generated code passes all test cases, whose test samples may be too simple to ensure the correctness of the final code. In this stage, we filter out an additional 20k samples from the 32k samples generated in the previous stage and ultimately retain 12k high-quality samples.

To sum up, we first select 70k instructions to iteratively construct reflection data, where 38k samples are discarded as they contain I/O operations or exceed the maximum iteration limitation. Finally, we filter out 20k samples with only one round of reflection, which may have some errors in the final code, and retain 12k high-quality samples.

B Additional Experiments

B.1 MultiPL-E

Following MagiCoder (Wei et al., 2023), we evaluate six wide languages, *i.e.,* Java, JavaScirpt, C++, PHP, Swift, and Rust, using MultiPL-E (Cassano et al., 2022) benchmark. We employ StarCoder (Li et al., 2023), WizardCoder (Luo et al., 2023), Code Llama (Rozière et al., 2023), and MagiCoder (Wei et al., 2023) as baselines. For this comparison, we follow MagiCoder and WizardCoder to set temperature = 0.2, top_p = 0.95, max_length = 512, and num_samples = 50. As shown in Table 7, the proposed ReflectionCoder outperforms the previous state-of-the-art methods on both Code Llama 7B and Code Llama 34B. It shows that reflection sequence in Python is also helpful to other languages. Surprisingly, ReflectionCoder Code Llama 7B even surpassed WizardCoder Code Llama 34B, which further demonstrates the effectiveness of the proposed method.

In addition, we compare our method to DeepSeek-Coder Instruct (Guo et al., 2024) on seven languages, which are reported in the DeepSeek-Coder paper, *i.e.,* C++, Java, PHP, TS, C#, Bash, and JavaScript. For this comparison, we adopted a greedy search approach following the DeepSeek-Coder Instruct. As shown in Table 8, the

Model	Base	plt	np	pd	py	scp	sk	tf	All
Incoder	6B	28.3	4.4	3.1	4.4	2.8	2.8	3.8	7.4
CodeGen-Mono	16B	31.7	10.9	3.4	7.0	9.0	10.8	15.2	11.7
Code-Cushman-001	-	40.7	21.8	7.9	12.4	11.3	18.0	12.2	18.1
StarCoder	SC-15B	51.7	29.7	11.4	21.4	20.2	29.5	24.5	26.0
WizardCoder	SC-15B	55.2	33.6	16.7	26.2	24.2	24.9	26.7	29.2
Code LLama	CL-7B	55.3	34.5	16.4	19.9	22.3	17.6	28.5	28.0
WizardCoder	CL-7B	53.5	34.4	15.2	25.7	21.0	24.5	28.9	28.4
MagiCoder	CL-7B	54.6	34.8	19.0	24.7	25.0	22.6	28.9	29.9
MagiCoder-S	CL-7B	55.9	40.6	**28.4**	40.4	**28.8**	35.8	**37.6**	37.5
ReflectionCoder	CL-7B	**56.2**	**43.1**	24.5	**46.7**	23.1	45.5	35.6	**37.8**
w/o Relfexion Data	CL-7B	56.0	42.7	23.0	43.6	26.7	**45.8**	35.6	37.4

Table 9: Pass@1 accuracy results on DS-1000 (Completion format). The best results of each base are in bold. Here, ‘SC’ denotes StarCoder, ‘CL’ denotes Code Llama.

Method	APPs	LiveCodeBench	ClassEval		BigCodeBench
			Class Level	Func Level	
MagiCoderS-DS-6.7B	12.8	17.6	20.0	43.4	47.6
OpenCodeInterpreter-DS-6.7B	11.5	17.6	19.0	42.6	44.6
ReflectionCoder-DS-6.7B	**14.1**	**18.4**	**25.0**	**44.0**	**47.9**
OpenCodeInterpreter-DS-33B	17.5	22.3	26.0	43.4	51.0
ReflectionCoder-DS-33B	**20.2**	**22.7**	**28.0**	**50.4**	**52.9**

Table 10: Pass@1 accuracy on APPs, LiveCodeBench, ClassEval, and BigCodeBench.

proposed ReflectionCoder outperforms DeepSeek-Coder Instruct in most languages. Note that the DeepSeek-Coder Instruct is trained with 2B tokens, while our models are trained with 300M tokens, which also shows the effectiveness of our methods. Our method outperforms DeepSeek-Coder Instruct in three languages on DeepSeek-Coder-6.7B and five languages on DeepSeek-Coder-33B, which shows that the larger model has a greater transfer ability.

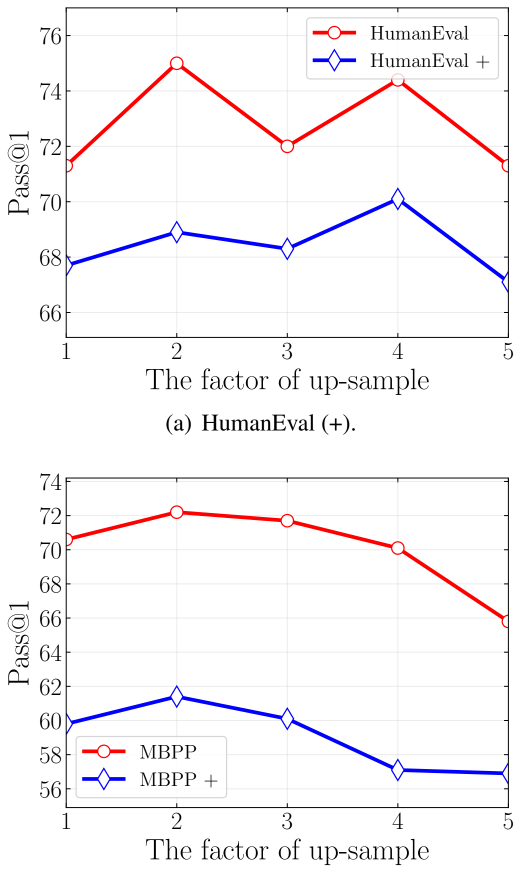
B.2 DS-1000

We also evaluate our method on the DS-1000 dataset (Lai et al., 2023), which contains 1K distinct data science coding issues, ranging from 7 popular Python data science libraries. We employ Incoder (Fried et al., 2023), CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023), WizardCoder (Luo et al., 2023), Code Llama (Rozière et al., 2023), and MagiCoder (Wei et al., 2023) as baselines. For this comparison, we follow MagiCoder to set temperature = 0.2, top_p = 0.95, max_length = 512, and num_samples = 40.

As shown in Table 9, our model outperforms all baselines on average score. However, when comparing our method with and without Reflection Data, where the latter is trained exclusively with 156k one-off code generation data points, our method does not significantly improve the DS-1000 dataset. A key factor contributing to this outcome is the limited representation of data related to these seven libraries in our training set, primarily due to constraints in computational resources. For instance, the need for substantial GPU resources restricts our ability to fully leverage TensorFlow and PyTorch, while the requirement for multi-modal capabilities limits our utilization of Matplotlib. Despite these limitations, it is noteworthy that our method does not adversely affect the performance of tasks associated with these libraries.

B.3 Other Test Set

Here, we check the effectiveness of our method on more diverse tasks, such as APPs (Hendrycks et al., 2021a) and LiveCodeBench (Jain et al., 2024), ClassEval (Du et al., 2023) and BigCodeBench (Zhuo et al., 2024). We construct experiments based on Deepseek-Coder-7B and Deepseek-Coder-33B. We employ MagiCoder (Wei et al., 2023) and OpenCodeInterpreter (Zheng et al., 2024) as baselines, which used similar fine-tuning data as our models. We use greedy sampling to obtain the results in a zero-shot setting for both baselines and our method. Note



(a) HumanEval (+).

(b) MBPP (+).

Figure 4: Effect of the factor of up-sample. The metric is Pass@1 accuracy, and all the results are based on Code Llama 7B.

that for LiveCodeBench, we report the result after 2023-09-01, which is the release date of Deepseek-Coder.

As shown in Table 10, our proposed method improves model accuracy on the four datasets, although there are no relative instructions in the training data. The results show that our method has better generalization.

B.4 Effect of the Factor of Up-sample

As mentioned in Section 4, we up-sample the reflection data and mix it with the code instruction tuning data. Here, we examine the effect of the up-sampling factor. Specifically, we vary the factor in the set $\{1, 2, 3, 4, 5\}$. As shown in Figures 4(a) and 4(b), a factor of 2 results in optimal performance for most benchmarks. Due to the limited samples in HumanEval, the pass@1 fluctuates significantly. While a factor of 4 is optimal for HumanEval+, a factor of 2 remains optimal for HumanEval. A possible reason is that when the factor is too large, the reflection sequence data is repeated excessively, leading to overfitting and a consequent decrease in performance.

Method	HumanEval (+)	MBPP (+)
Code Llama 7B		
ReflectionCoder	**75.0 (68.9)**	**72.2 (61.4)**
w/o Reflection Data	65.9 (62.2)	68.5 (57.9)
Star Coder 7B		
ReflectionCoder	**68.3 (63.4)**	64.3 (**55.6**)
w/o Reflection Data	67.7 (62.8)	**66.7** (54.8)

Table 11: Effect of Rotary Position Embedding. The metric is Pass@1 accuracy.

Method	HumanEval (+)	MBPP (+)
Random Mask	**72.0 (66.5)**	**70.1 (59.0)**
w/ Token Level	71.3 (66.5)	68.8 (58.2)
Sequential Mask	**72.6 (67.7)**	**71.3 (60.3)**
w/ Token Level	71.3 (67.1)	68.5 (59.0)

Table 12: Compare block-level mask strategies and token-level mask strategies. The metric is Pass@1 accuracy, and all the results are based on Code Llama 7B.

B.5 Effect of Rotary Position Embedding

As mentioned in Section 3, our method is effective for models utilizing Rotary Position Embedding because the absolute positions of the tokens of the answers in the teacher sample and the student sample are different, but the relative positions remain the same. Here, we construct an experiment to check the effect of Rotary Position Embedding on our method. Specifically, we perform our method and w/o Reflection Data on StarCoder, which uses an Absolute Position Embedding.

Table 11 shows the results on both Code Llama 7B (w/ Rotary Position Embedding) and StarCoder 15B (w/ Absolute Position Embedding). As shown in the table, our method can effectively improve the performance of Code Llama 7B, but it is not so effective for StarCoder 15B. The primary reason is that the absolute positions of the tokens of the final answers are different for the training stage and the inference stage, which results in the distillation being biased.

B.6 Token-level Dynamic Masking Strategy

In Section 3, we proposed three block-level dynamic masking strategies, namely random mask, sequential mask, and block mask. Here, we test our method with another two token-level dynamic

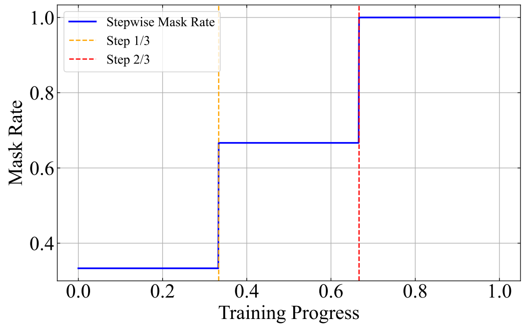


Figure 5: The changes in masked rate during training.

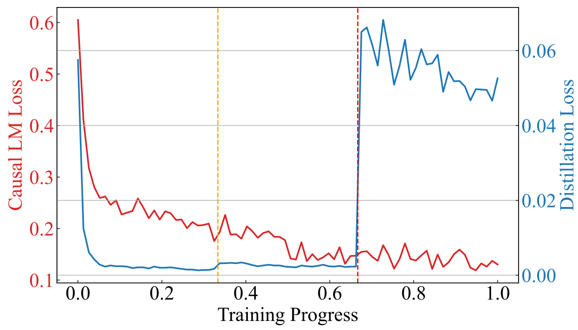


Figure 6: The changes in value of two loss components during training.

masking strategies:

(1) *Random Token mask* selects tokens to mask based on the mask rate randomly.

(2) *Sequential Token mask* selects the leftmost tokens to mask and gradually expands the masked scope according to the mask rate.

Table 12 shows the results on both block-level masking strategies and token-level masking strategies. The block-level masking strategies significantly outperform token-level masking strategies. Because the token-level masking strategies may destroy the integrity of texts or codes.

B.7 Training Procedure

Here, we visualize the evolution of the masking rate and the training losses throughout the learning process. As shown in Figure 5, the masking schedule follows a stepwise curriculum strategy: during the first third of the training, all execution blocks are masked; in the second third, generation blocks are additionally masked; and in the final third, analysis blocks are further masked. This progressive masking design encourages the model to gradually adapt to more challenging conditions and enhances its robustness in code generation. As shown in Figure 6, the causal language modeling loss steadily decreases over time, indicating consistent improvements in the model's predictive ability. In contrast, the distillation loss displays sharp spikes at each one-third interval, followed by a gradual decline. These abrupt increases are caused by the sudden introduction of new masked block types, which temporarily raise the difficulty of the distillation task until the model re-stabilizes.

C Training Sample

Here, we present how to organize training samples. Following chat templates of most models, we use two special tokens to wrap the message of user and assistant, *i.e.,* "<|user|>", "<|assistant|>", and "<|endofmessage|>". Furthermore, we use two special tokens to wrap the content of different blocks, *i.e.,* "<|text|>", "<|code|>", "<|execution|>", and "<|endofblock|>". Then, we concatenate all messages directly to form a piece of training data. Note that the user message and execution block are masked when calculating the loss because the environment provides the two contents during actual use. Here is an example:

Reflection Instruction

<|user|>
<|text|>
Here is a programming challenge for you to tackle:
(1) Write a Python function that solves the specified problem with craft test cases using assert statements and execute it. Pay special attention to edge cases to thoroughly validate your solution's correctness.
(2) If your code fails any of the tests, carefully examine the root cause of the failure. Make the necessary corrections to your code and then retest to confirm the fixes.

Problem
Execute an algorithm to generate the numerical multiplication of individual integer components encapsulated within a larger numeric figure.
n = 145
<|endofblock|>
<|endofmessage|>

Reflection Sequence

<|assistant|>
<|code|>
```
def multiply_digits(n):
    """
    This function takes an integer n and returns the product of its digits.
    """
    product = 1
    while n > 0:
        product *= n % 10
        n //= 10

    return product

# Test cases
assert multiply_digits(145) == 4 * 5 * 1
assert multiply_digits(123) == 1 * 2 * 3
assert multiply_digits(999) == 9 * 9 * 9
assert multiply_digits(0) == 0
assert multiply_digits(10) == 1 * 0
```
<|endofblock|>
<|execution|>
```
AssertionError                    Traceback (most recent call last)
Cell In[1], line 15
     13 assert multiply_digits(123) == 1 * 2 * 3
     14 assert multiply_digits(999) == 9 * 9 * 9
---> 15 assert multiply_digits(0) == 0
     16 assert multiply_digits(10) == 1 * 0

AssertionError:
```
<|endofblock|>
<|text|>
The assertion 'assert multiply_digits(0) == 0' failed. This indicates an issue with the function when handling the case where 'n' is 0.

In the current implementation, when 'n' is 0, the while loop is never entered, and thus the function returns the initial value of 'product', which is 1. However, the correct behavior should be to return 0, as the product of the digits of 0 is 0.

I'll correct the function to handle this special case and retest the code.
<|endofblock|>
<|code|>
```
def multiply_digits(n):
    """
    This function takes an integer n and returns the product of its digits.
    """
    if n == 0:
        return 0

    product = 1
    while n > 0:
        product *= n % 10
        n //= 10

    return product

# Retesting with the original test cases and the corrected case
assert multiply_digits(145) == 4 * 5 * 1
assert multiply_digits(123) == 1 * 2 * 3
assert multiply_digits(999) == 9 * 9 * 9
assert multiply_digits(0) == 0
assert multiply_digits(10) == 1 * 0
```
<|endofblock|>
<|execution|>
<|endofblock|>
<|text|>
The modified code has passed all the test cases, including the previously failed case for 'n = 0'. The function 'multiply_digits' now correctly handles all tested scenarios.
<|endofblock|>
<|endofmessage|>

Instruction

<|user|>
<|text|>
Execute an algorithm to generate the numerical multiplication of individual integer components encapsulated within a larger numeric figure.
n = 145
<|endofblock|>
<|endofmessage|>

Final Answer

```
<|assistant|>
<|text|>
```python
def multiply _digits(n):
    """
    This function takes an integer n and returns the product of its digits.
    """
    if n == 0:
        return 0

    product = 1
    while n > 0:
        product *= n % 10
        n //= 10

    return product
```
<|endofblock|>
<|endofmessage|>
```

D Test Prompt

Here, we present the test prompt for each test set to facilitate reproduction.

For HumanEval, we add "Write a complete Python function for the problem" before the original prompt from EvalPlus (Liu et al., 2023) to make the model follow the instructions better.

Test Prompt for HumanEval

```
<|user|><|text|>Write a complete Python function for the problem.

{prompt}<|endofblock|><|endofmessage|>
<|assistant|><|text|>```python
```

For MBPP, we follow WizardCoder (Luo et al., 2023) and Magicoder (Wei et al., 2023) to reconstruct the source problem from EvalPlus (Liu et al., 2023) with some additional prompts.

Test Prompt for MBPP

```
<|user|><|text|> {prompt}
Your code should satisfy the following assertion:
```python
{test sample}
```<|endofblock|><|endofmessage|>
<|assistant|><|text|>```python
```

For MultiPL-E, we put the source prompts in both the user message and the beginning of the assistant message to ensure that the model does not modify the prompt and only completes it.

Test Prompt for MultiPL-E

```
<|user|><|text|>Write a complete {language} function for the problem.

{prompt}<|endofblock|><|endofmessage|>
<|assistant|><|text|> ```{language}
{prompt}
```

For DS-1000, we directly use the source prompts.

For APPs and LiveCodeBench, we add "Write a complete Python script for the question, Please note that you need to handle the stdin input, e.g. t = int(input())." before the original prompt to make the model follow the instructions better.

Test Prompt for APPs / LiveCodeBench

```
<|user|><|text|>Write a complete Python script for the question, Please note that you need to handle the stdin input, e.g. t = int(input()).

{prompt}<|endofblock|><|endofmessage|>
<|assistant|><|text|> ```python
```

For ClassEval, we add "Please complete the class {class name} in the following code." before the original prompt to make the model follow the instructions better.

Test Prompt for ClassEval

```
<|user|><|text|>Please complete the class {class name} in the following code.

{prompt}<|endofblock|><|endofmessage|>
<|assistant|><|text|> ```python
```