AMR-Evol: Adaptive Modular Response Evolution Elicits Better Knowledge Distillation for Large Language Models in Code Generation

♠Ziyang Luo, ♡Xin Li*, ♠Hongzhan Lin, ♠Jing Ma*, ♡Lidong Bing
♠Hong Kong Baptist University, ♡Alibaba DAMO Academy
{cszyluo,majing}@comp.hkbu.edu.hk   xinting.lx@alibaba-inc.com

Abstract

The impressive performance of proprietary LLMs like GPT4 in code generation has led to a trend to replicate these capabilities in open-source models through knowledge distillation (e.g. Code Evol-Instruct). However, these efforts often neglect the crucial aspect of response quality, relying heavily on teacher models for direct response distillation. This paradigm, especially for complex instructions, can degrade the quality of synthesized data, compromising the knowledge distillation process. To this end, our study introduces the **A**daptive **M**odular **R**esponse **Evol**ution (**AMR-Evol**) framework, which employs a two-stage process to refine response distillation. The first stage, modular decomposition, breaks down the direct response into more manageable sub-modules. The second stage, adaptive response evolution, automatically evolves the response with the related function modules. Our experiments with three popular code benchmarks—HumanEval, MBPP, and EvalPlus—attests to the superiority of the **AMR-Evol** framework over baseline response distillation methods. By comparing with the open-source Code LLMs trained on a similar scale of data, we observed performance enhancements: more than +3.0 points on HumanEval-Plus and +1.0 points on MBPP-Plus, which underscores the effectiveness of our framework. Our codes are available at https://github.com/ChiYeungLaw/AMR-Evol.

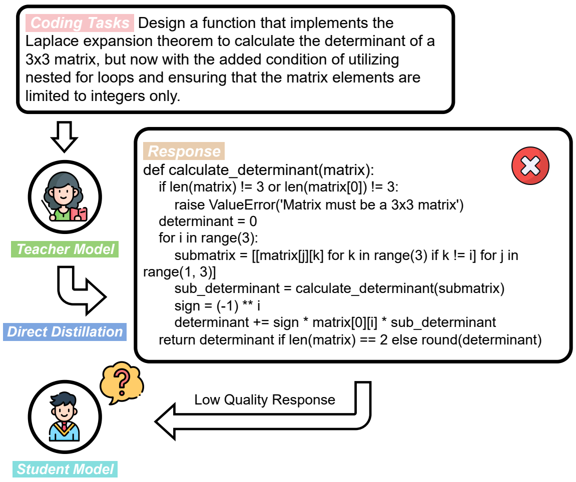


Figure 1: Direct distillation from the teacher model possibly yields low quality responses for complex tasks, thereby causing confusion within the student model.

1 Introduction

Recently, the powerful proprietary large language models (LLMs), like GPT3 (Brown et al., 2020), GPT4 (OpenAI, 2023), Gemini (Anil et al., 2023a) and Claude (Anthropic, 2023), have showcased impressive code generation ability. Especially, GPT4, the most performant model, has recorded pass rates exceeding 85% on the well-known HumanEval benchmark (Chen et al., 2021). Despite their strengths, the closed-source nature sparks accessibility and privacy concerns (Wu et al., 2023). In response, there is a trend of adopting knowledge distillation (Xu et al., 2024) to transfer the advanced code generation ability from the proprietary LLMs to open-source counterparts, thereby enhancing their capabilities while ensuring broader availability and owner autonomy.

Given that accessing the model weights of proprietary LLMs is infeasible, the knowledge distillation pipeline is considered as a process where the teacher models synthesize supervised data, primarily consisting of instruction-response pairs (Liu et al., 2024). Student models are subsequently trained on this data, enabling the transfer of capabilities from the teacher models. For example, Chaudhary (2023) employs the self-instruct method (Wang et al., 2022) to prompt the teacher model to generate new coding instructions based on predefined seed tasks. Similarly, OSS-Instruct (Wei

*Corresponding Authors.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 1143–1166
November 12-16, 2024

et al., 2023) utilizes a variety of code snippets sourced from GitHub to inspire GPT-3.5 to produce novel coding instructions. Likewise, Code Evol-Instruct (Luo et al., 2024) employs iterative prompting to progressively elevate the complexity of code instructions provided by teacher models. Each of these methods has proven effective in distilling coding knowledge from teacher models.

Despite these advancements, there remains an unresolved challenge in enhancing the quality of code response distillation within the data synthesis process. In this setting, code responses serve as labels that teach the student models. Previous works have shown that higher-quality responses can lead to more effective distillation (Zhou et al., 2023; Mukherjee et al., 2023). However, current methods (Chaudhary, 2023; Wei et al., 2023; Luo et al., 2024) tend to rely solely on teacher models for direct response distillation. As shown in Figure 1, this approach is limited by the capabilities of the teacher models, making it difficult to produce accurate responses for complex tasks. The issue becomes even more challenging with methods like Code Evol-Instruct, which deliberately amplify the complexity of instructions. Consequently, relying on direct distillation can result in lower-quality responses, ultimately affecting the performance of the student models (Wang et al., 2024).

A straightforward yet costly solution to guarantee response quality is to hire human annotators to craft the unit tests for each response. These tests could then be used in an execution-based strategy to validate answers. However, this method is financially prohibitive because it requires the recruitment of annotators with extensive programming expertise. Alternatively, depending on the teacher model to automatically generate unit tests for self-repair (Chen et al., 2023a; Olausson et al., 2023; Chen et al., 2023c) introduces the same concern of response quality, providing no certainty regarding the correctness of the code repair.

To address the challenge of distilling high-quality code responses from teacher models, we introduce a novel framework named **A**daptive **M**odular **R**esponse **Evol**ution (**AMR-Evol**). In Figure 1, the example reveals that the direct response distillation can somewhat capture the essential concepts required for solving coding tasks; however, it often deviates from the specific requirements and incorporates logical errors. Motivated by this observation, **AMR-Evol** leverages the outputs of direct distillation as seed data and employs a two-stage process—namely, modular decomposition and adaptive response evolution—to gradually refine the distilled code responses. By intricately refining the process of response distillation, our framework elicits better knowledge distillation of the student models.

In the first stage of our **AMR-Evol**, we adopt the idea from modular programming (Dijkstra, 1967) to manage the complexity of distilling code responses. By utilizing direct responses as the seeds, this method breaks down the coding task into smaller, more manageable sub-modules. This strategy shifts the focus of the teacher models towards solving these sub-modules step-by-step rather than generates a complete solution in a single attempt, whose effectiveness has been verified in recent Chain-of-X works (Wei et al., 2022; Le et al., 2023; Xia et al., 2024).

Additionally, while coding tasks may vary significantly in objectives, the modular components need to construct their solutions frequently exhibit commonalities, or can even be identical (Parnas, 1972). Hence, our adaptive response evolution stage leverages an auxiliary functional module database to store all validated modules for reuse. During response generation, this process utilizes the modules formulated in the decomposition stage to retrieve suitable, pre-validated modules from the database. These related modules serve as in-context examples, aiding the adaptive refinement of responses, thus reducing our sole reliance on teacher models. As evolution progresses, any newly created modules that differ from those in the database are added after a verification process by the teacher model.

We apply our **AMR-Evol** framework to different student models and select the most representative coding benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and EvalPlus (Liu et al., 2023), for evaluation. The results reveal that our **AMR-Evol** framework consistently surpasses other response distillation methods, namely direct response distillation, chain-of-thought distillation, and response repairing. These results affirm the superiority of our approach in improving knowledge distillation for LLMs in code generation. Moreover, by integrating our **AMR-Evol** with Code Evol-Instruct, one of the SOTA instruction construction methods, our models achieve better performance than the open-source alternatives trained on a comparable data scale. Specifically, we observed an improvement of more than +3.0 on HumanEval-Plus and +1.0 on MBPP-Plus.

2 Related Work

LLMs and Code Generation. Recently, LLMs have showcased significant achievements across a vast array of tasks. Leading tech firms have made substantial progress in developing highly advanced close-source LLMs, including OpenAI's GPT4 (OpenAI, 2023), Google's PaLM (Chowdhery et al., 2022; Anil et al., 2023b) and Gemini (Anil et al., 2023a), as well as Anthropic's Claude (Anthropic, 2023). On the other side, the AI community has also seen the launch of several open-source LLMs, with model weights becoming publicly available. MistralAI has contributed the Mistral-Series (Jiang et al., 2023). Google has released UL2-20B (Tay et al., 2022) and Gemma (Mesnard et al., 2024). Tsinghua University introduced GLM-130B (Zeng et al., 2022) and MiniCPM (Hu et al., 2024), while Meta has made available OPT (Zhang et al., 2022) and LLaMA1&2&3 (Touvron et al., 2023a,b; Meta, 2024). Furthermore, Allen AI has introduced the wholly open-sourced LLM, OLMo (Groeneveld et al., 2024), and Microsoft has released Phi-series (Gunasekar et al., 2023; Li et al., 2023b). Although a gap remains between the open-source models and their closed-source counterparts, this gap is gradually narrowing.

In parallel, recent research efforts have been directed towards leveraging LLMs for code-related tasks to address the understanding and generation of code. OpenAI has unveiled Codex (Chen et al., 2021), Google has proposed CodeGemma (Google, 2024), and Salesforce has introduced CodeGen-Series (Nijkamp et al., 2023b,a), and CodeT5&Plus (Wang et al., 2021, 2023). Contributions from Tsinghua University include CodeGeeX (Zheng et al., 2023), and the BigCode Project has developed StarCoder1&2 (Li et al., 2023a; Lozhkov et al., 2024). Meta has also made its mark with the CodeLlama (Rozière et al., 2023), while DeepSeek has open-sourced the DeepSeekCoder (Guo et al., 2024). These initiatives underscore the growing interest in employing powerful base LLMs for code generation. Our work introduces a novel method for more effectively distilling code knowledge from closed-source models to these open-source base models, thereby enhancing the coding performance.

Knowledge Distillation for Code Generation. To enhance the capabilities of open-source LLMs for code generation, recent works have adopted the knowledge distillation paradigm, utilizing closed-source LLMs as teachers for supervised data synthesis (Chen et al., 2023b; Zheng et al., 2024; Li et al., 2024; Yuan et al., 2024). For example, Chaudhary (2023) employs the self-instruct method (Wang et al., 2022) to generate training data, while Magicoder (Wei et al., 2023) generates training content using code snippets from GitHub. WizardCoder (Luo et al., 2024), on another hand, introduces the Code Evol-Instruct approach to progressively increase the complexity of coding tasks. Despite these advancements, a common limitation among these efforts is their primary focus on the creation of code instructions, often overlooking the criticality of enhancing code response distillation. Our research takes an orthogonal path by concentrating on the refinement of code response distillation, offering a novel perspective compared to previous works.

3 Method

As depicted in Figure 2, we introduce our novel framework, **AMR-Evol**, aimed at improving code response distillation to elicit better performance of the student models. In this section, we will provide a detailed discussion of our framework's pipeline.

3.1 Direct Response Distillation

In the knowledge distillation framework, the foremost goal is enabling the student model $\mathcal{M}_s$ to assimilate the strategies deployed by the teacher model $\mathcal{M}_t$ in tackling code generation tasks. Utilizing approaches like Code Evol-Instruct facilitates the generation of an extensive dataset of code instructions $\{I\}$ by the teacher model. Subsequently, the direct response distillation method employs the teacher model to process these task instructions to produce the corresponding code responses R_d, resulting in a paired dataset, $\mathcal{D}_{direct} = \{(I, R_d)\}$. Then, the student model $\mathcal{M}_s$ learns from this dataset through supervised fine-tuning.

3.2 Adaptive Modular Response Evolution

As discussed in Section 1, direct responses $\mathcal{D}_{direct}$ to complex instructions can result in suboptimal quality, which in turn impacts the performance of the student model $\mathcal{M}_s$. While these responses often include logical errors or may not fully align with the precise requirements of the tasks, they generally remain close to correct and capture the essential concepts needed for task solution. To address this,

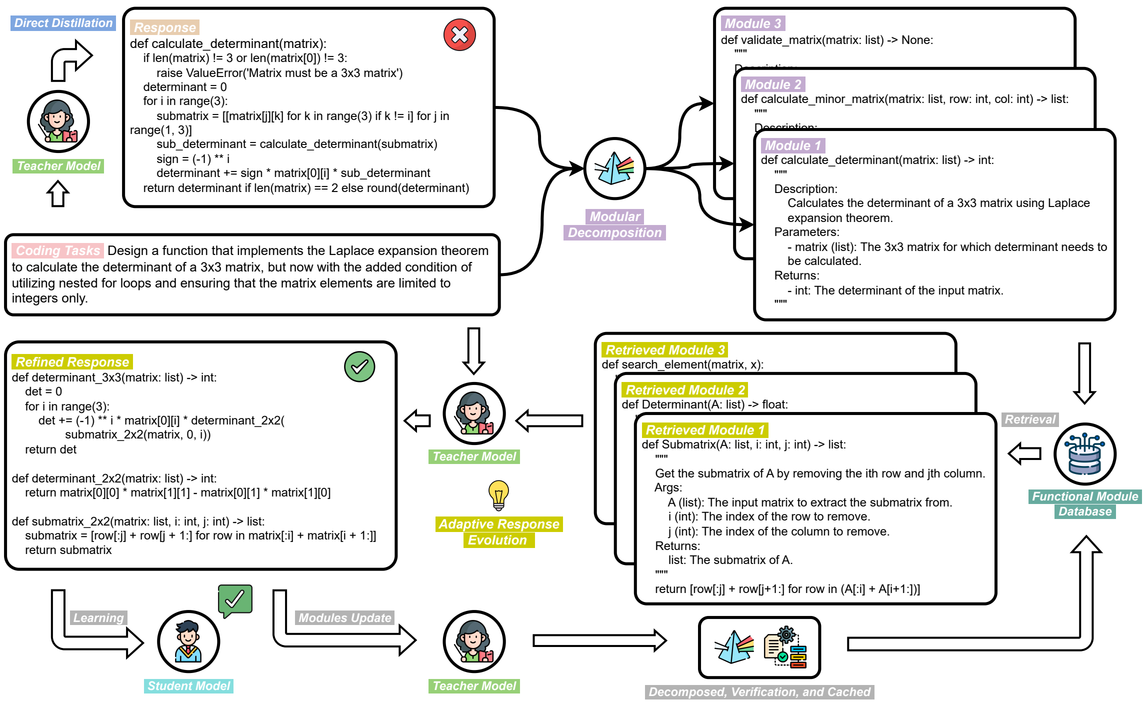


Figure 2: Our **Adaptive Modular Response Evolution (AMR-Evol)** framework with modular decomposition and adaptive response evolution elicits better response distillation for LLMs in code generation.

our **AMR-Evol** framework capitalizes on these direct response distillations as a starting point. It incorporates a two-stage method—modular decomposition and adaptive response evolution—for an automated refinement process that improves the quality of responses, thereby enhancing the efficacy of distillation.

Modular Decomposition (MD). In the first stage of our framework, we employ the principle of modular programming (Dijkstra, 1967) to tackle the complexity inherent in distilling code responses. Our method utilizes direct responses R_d as a starting point, guiding the teacher model $\mathcal{M}_t$ in breaking down the given code instructions into a series of smaller, well-defined sub-modular functions. We represent this process mathematically as follows:

$$\{F_1^m, F_2^m, \ldots, F_n^m\} \leftarrow \mathcal{M}_t\left(I, R_d\right), \tag{1}$$

where each function module F_i^m is conceptualized to fulfill a distinct subset of requirements stipulated by the code instruction I. This decomposition breaks down complex instructions into a series of easier and more manageable sub-modules, enabling the teacher model to tackle each one with less difficulty. This results in a more effective response distillation process.

Adaptive Response Evolution (ARE). In the second stage, we observe that while coding instructions may greatly differ, the sub-modules needed for assembling the final solution often share similarities or can even be identical (Parnas, 1972). Leveraging this insight, we establish an auxiliary functional module database $\{F_i^v\}$, which archives all validated modules for future reuse. This database acts as a repository, enabling the retrieval of previously validated sub-modules to foster the creation of new code responses.

Building upon the modular decomposition achieved in the first stage, $\{F_1^m, F_2^m, \ldots, F_n^m\}$, we initially convert both the newly decomposed and previously archived functional modules into dense vector representations through a sentence embeddings model $\mathcal{M}_r$:

$$V_{f_i^{(\cdot)}} \leftarrow \mathcal{M}_r\left(F_i^{(\cdot)}\right), \tag{2}$$

where $V_{f_i^{(\cdot)}}$ denotes the dense representation of any given functional module $F_i^{(\cdot)}$. Then, to facilitate the retrieval of the most suitable archived module for each new sub-module, we apply:

$$\text{Sim}\left(F_i^m, F_j^v\right) \leftarrow \text{CosineSimilarity}\left(V_{f_i^m}, V_{f_j^v}\right), \tag{3}$$

where $\text{Sim}\left(F_i^m, F_j^v\right)$ calculates the similarity between the dense representations of two modules using cosine similarity. The archived modules that exhibit the highest similarity are then used as additional in-context contents, assisting the teacher model in refining the final code responses:

$$R_{amr} \leftarrow \mathcal{M}_t\left(I, \{F_i^m\}, \{F_i^v\}\right), \tag{4}$$

where R_{amr} represents the refined code responses. These responses, alongside the original instruction I, compile an evolved dataset aimed at optimizing the knowledge distillation process.

As the process evolves, our framework identifies new modules within R_{amr} that exhibit notable differences from those currently in the database—judged by the cosine similarity between the new modules and existing ones. Modules that are distinct undergo a rigorous verification stage prior to their integration into the database. This critical stage harnesses the capabilities of the teacher model for generating unit tests tailored to the functionalities of the specific modules. This procedure not only assesses the functional correctness of the new modules but also ensures that they meet the predefined quality standards, thereby streamlining the process of enriching the module database with reliable and effective components.

Functional Module Database. The functional module database is pivotal within our **AMR-Evol** framework. We begin by compiling a collection of seed functions that have been validated. Leveraging the self-instruct method (Wang et al., 2022), we prompt our teacher models to generate a diverse range of function modules. Following this, we adopt a strategy similar to CodeT (Chen et al., 2023a), instructing the teacher models to produce unit tests that verify the functionality of these modules. Only the functions that pass these unit tests are included in our dataset. Through this stringent process, we construct a seed functional module database that becomes a fundamental component of our framework.

3.3 Knowledge Distillation

Upon completing the data synthesis process with the help of teacher models, we acquire a dataset that consists of paired instructions and responses, $\mathcal{D}_{amr} = \{(I, R_{amr})\}$. This dataset equips the student model $\mathcal{M}_s$ for the task of knowledge distillation, where it is trained to use I as input with the goal of generating responses R_{amr} that closely resemble those produced by the teacher model. The training follows an auto-regressive learning objective, formalized as follows:

$$\mathcal{L}(\theta) = -\sum_{(I,R_{amr})\in\mathcal{D}_{amr}} \log P(R_{amr}|I;\theta), \tag{5}$$

where $\mathcal{L}(\theta)$ denotes the loss function minimized during training, and θ signifies the parameters of the student model $\mathcal{M}_s$. This objective encourages the student model to accurately predict the next token in the response sequence, given the instruction I and the current state of the generated response.

4 Experiment

4.1 Setup

Baselines. Within our evaluation framework, we compare the performance of our framework against several baselines in code response distillation. The first of these, referred to as **direct**, utilizes teacher models to distill code responses in a straightforward manner, as detailed in Section 3.1. The second baseline employs the Chain-of-Thought (**CoT**) prompting method for distilling responses (Hsieh et al., 2023). This approach is analogous to the few-shot CoT method (Wei et al., 2022), in which the teacher model first provides a step-by-step explanation leading up to the formulated response. Our third baseline, **AnsRepair**, draws inspiration from previous works (Chen et al., 2023a; Olausson et al., 2023; Chen et al., 2023d), where the teacher models are utilized to generate unit tests. These tests serve to evaluate the correctness of the generated responses. If the responses fail these tests, the teacher models are subsequently invoked to make the necessary corrections. More details about baseline methods are included in the Appendix A.

Datasets and Benchmarks. Our framework focuses on distilling responses and necessitates a dataset of instructions. To this end, we utilize a subset of the training set from the MBPP as our seed data. This is then expanded using the self-instruct method with the teacher model to generate around 10k instructions. With these newly derived instructions, we employ a process akin to the Code Evol-Instruct to iteratively synthesize a spectrum of complex coding instructions across three distinct levels of complexity. This variety allows us to assess our framework's efficacy in handling complex instructions. More data construction and decontamination details can be found in the Appendix B.

Method	HE	HE-Plus	MBPP	MBPP-Plus
Complexity Level 1				
Direct	54.9	46.3	65.9	54.1
CoT	52.4	45.7	65.7	53.4
AnsRepair	53.7	45.1	63.2	52.1
AMR-Evol	58.5	49.4	68.7	58.1
Δ	+3.6	+3.1	+2.8	+4.0
Complexity Level 2				
Direct	53.7	46.3	64.4	52.6
CoT	54.9	46.3	65.7	53.9
AnsRepair	56.1	47.6	63.4	52.9
AMR-Evol	56.1	47.6	68.7	56.6
Δ	+0.0	+0.0	+3.0	+2.7
Complexity Level 3				
Direct	52.4	45.7	65.2	53.9
CoT	52.4	43.9	65.7	53.9
AnsRepair	55.5	47.6	65.4	53.1
AMR-Evol	56.1	49.4	67.7	56.4
Δ	+0.6	+1.8	+2.0	+2.5

Table 1: Comparison of various response distillation methods for code generation, utilizing `deepseek-coder-6.7b-base` as the student model.

Method	HE	HE-Plus	MBPP	MBPP-Plus
Complexity Level 1				
Direct	36.6	31.1	54.4	44.1
CoT	36.0	31.1	55.1	45.6
AnsRepair	35.4	29.3	56.4	45.4
AMR-Evol	37.8	32.3	57.4	45.6
Δ	+1.2	+1.2	+1.0	+0.0
Complexity Level 2				
Direct	37.2	31.1	55.4	44.6
CoT	36.0	31.1	54.6	45.6
AnsRepair	35.4	29.3	56.6	45.9
AMR-Evol	39.6	32.3	59.4	47.6
Δ	+2.4	+1.2	+2.8	+1.7
Complexity Level 3				
Direct	36.0	30.5	56.4	45.6
CoT	37.2	30.5	55.6	46.4
AnsRepair	37.2	29.3	55.6	44.9
AMR-Evol	39.0	32.9	59.1	46.9
Δ	+1.8	+2.4	+2.7	+0.5

Table 2: Comparison of various response distillation methods for code generation, utilizing `CodeLlama-7b-hf` as the student model.

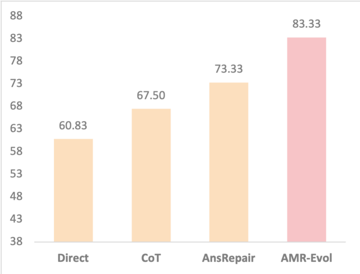
For performance evaluation, we utilize the well-known coding benchmark, namely HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and EvalPlus (Liu et al., 2023). HumanEval contains 164 coding problems with an average of 9.6 test cases per problem. MBPP includes 399 coding problems, each with three automated test cases. EvalPlus extends the number of test cases for both HumanEval and MBPP, resulting in enhanced versions named HumanEval-Plus and MBPP-Plus. Following EvalPlus, we report our method's effectiveness in terms of pass rates using greedy decoding, which helps minimize the impact of any randomness in the results. More details are included in the Appendix C.

Implementation Details. For all experiments, we employ OpenAI's close-sourced LLM, `gpt-3.5-turbo-1106` as our teacher model and choose two popular open-sourced code LLMs, `deepseek-ai/deepseek-coder-6.7b-base` (Guo et al., 2024) and `meta-llama/CodeLlama-7b-hf` (Rozière et al., 2023) as our student models. For the dense embeddings, we adopt one of the SOTA embeddings models, `Alibaba-NLP/gte-large-en-v1.5` (Li et al., 2023c) as our representation model. The supervised knowledge distillation phases of all experiments are conducted with 200 training steps, 3 epochs, a sequence length of 2048 and the AdamW optimizer (Loshchilov and Hutter, 2019). For further training details and prompting designes, please refer to the Appendix D.

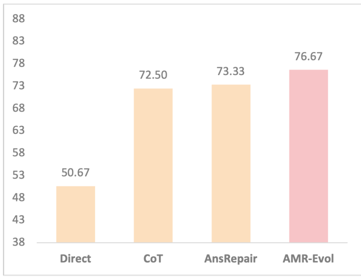
4.2 Main Results

In Table 1, our **AMR-Evol** consistently outperforms various response distillation methods for code generation, when adopt the `deepseek-coder-6.7b-base` as the student model. Specifically, at Complexity Level 1, **AMR-Evol** exhibited superior results, with improvements ranging between +2.8 to +4.0 across all tasks. Our method maintained this lead in Complexity Level 2, with the most substantial gains in MBPP and MBPP-Plus, at +3.0 and +2.7, respectively. Notably, even at the highest complexity (Level 3), the method continued to show incremental enhancements, most prominently a +2.5 increase in MBPP-Plus. The performance exhibits **AMR-Evol's** consistent proficiency in eliciting better code knowledge distillation across varying degrees of complexity.

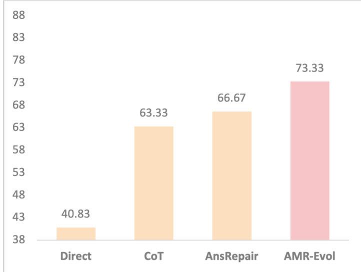
When utilizing `CodeLlama-7b-hf` as the student model, Table 2 reveals that the performance patterns of **AMR-Evol** closely paralleled its efficacy with the previous model. Albeit with modest improvements at Complexity Level 1, **AMR-Evol** showed more enhancement in higher complexity scenarios. At Complexity Level 2, our method achieves increases of +2.4 on HE and +2.8 on



(a) Complex 1.



(b) Complex 2.



(c) Complex 3.

Figure 3: Manual evaluation of the accuracy of various code response distillation methods across 120 randomly selected samples from each complexity level.

MBPP. The upward trend persisted through Complexity Level 3, as the method underscored its robustness with increases such as +2.4 on HE-Plus and +2.7 on MBPP. These results solidify **AMR-Evol** as an effective method for code knowledge distillation, adaptable to various instruction complexity levels.

4.3 Analysis

Quality Comparison. Our experimental findings illustrate the effectiveness of our **AMR-Evol** in enhancing the knowledge distillation. To further validate the efficacy of **AMR-Evol** in producing better instruction fine-tuning data, we conducted a manual evaluation. We randomly selected the sample sets of 120 coding problems for each levels of complexity. Given that all samples are coding challenges, their responses can be definitively classified as either correct or incorrect. Two experienced programmers were engaged to review and label the code responses generated by various methods as suitable or not. The manual assessment results, depicted in Figure 3, reveal that although no method attained complete perfect, **AMR-Evol** demonstrated consistently superior performance compared to all other baseline methods across all complexity levels. In Appendix E, we also include some examples of responses generated by different methods to qualitatively compare their quality.

Ablation. In Table 3, we present an ablation study meticulously designed to identify the individual contributions of modular decomposition (MD) and adaptive response evolution (ARE) to the efficacy of our framework. First, we remove the MD stage in our framework by adopting direct response to retrieve the related function modules for ARE. This led to a performance drop, underscoring its crucial role in our framework. Specifically, the omission of MD typically results in the recall of only one function module based on the direct response. However, while direct responses address more complex or larger coding tasks, function modules target tasks with finer granularity. This difference creates a gap, making it challenging for the retrieved function modules to effectively contribute to refining the direct responses.

Method	**HE**	**HE-Plus**	**MBPP**	**MBPP-Plus**
Complexity Level 1				
AMR-Evol	58.5	49.4	68.7	58.1
w/o MD	57.9	49.4	67.4	55.9
w/o ARE	56.1	48.8	69.4	57.1
Complexity Level 2				
AMR-Evol	56.1	47.6	68.7	56.6
w/o MD	54.9	46.3	67.7	54.4
w/o ARE	54.9	47.0	67.4	55.9
Complexity Level 3				
AMR-Evol	56.1	49.4	67.7	56.4
w/o MD	54.3	47.6	66.4	53.6
w/o ARE	53.0	47.0	67.4	54.6

Table 3: Ablation studies by removing modular decomposition (MD) or adaptive response evolution (ARE) in our framework.

Subsequently, we exclude the ARE stage, which also resulted in a performance decline, highlighting its vital role in the framework. Without ARE, the generation of responses is solely reliant on the modular decomposition output, lacking the improvements that come from in-context learning with related function modules. This places the entire responsibility for refining responses on the inherent capabilities of the teacher model. This analysis strongly reinforces the indispensable nature of both MD and ARE within our framework. In Appendix F, we also present examples to showcase the output of the MD stage and the top-1 function modules retrieved from the database.

Model	Size	#SFT Ins	HE	HE-Plus	MBPP	MBPP-Plus
Proprietary models						
GPT4	-	-	85.4	81.7	83.0	70.7
GPT3.5	-	-	72.6	65.9	81.7	69.4
Gemini Pro	-	-	63.4	55.5	72.9	57.9
Base model: deepseek-ai/deepseek-coder-6.7b-base						
†**DeepSeekCoder-Instruct**	6.7B	>1M	73.8	70.1	72.7	63.4
MagiCoder-DS	6.7B	75k	63.4	57.3	75.2	61.9
‡**WaveCoder-DS**	6.7B	20k	66.5	57.9	73.7	60.4
DeepSeekCoder-AMR-Evol	6.7B	50k	68.9	61.0	74.4	62.9
Base model: meta-llama/CodeLlama-7b-Python-hf						
†**CodeLlama-Instruct**	7B	80k	32.9	26.8	59.1	45.6
WizardCoder-CL	7B	78k	55.5	48.2	64.9	53.9
MagiCoder-CL	7B	75k	54.3	48.8	63.7	51.9
CodeLlama-AMR-Evol	7B	50k	59.1	51.8	64.7	55.4

†: Official instruction models. Responses are distilled from unknown, humans or themselves.
‡: Responses are distilled from GPT4.

Table 4: Comparison of our fine-tuned models against both publicly available academic Code LLMs, similarly scaled in terms of SFT data and based on the same student models as ours, and the official instruction-based LLMs. We either download the model weights or utilize the APIs for performance reproduction.

4.4 Comparing with Open Code LLMs

To delve deeper into the efficacy of our framework, we have incorporated **AMR-Evol** with one of the SOTA instruction construction methods, Code Evol-Instruct, to expand our SFT data set. We have generated around 50k instructions using this approach and employed **AMR-Evol** to distill code responses from the teacher models (GPT3.5). Subsequently, we used `deepseek-coder-6.7b-base` and `CodeLlama-7b-Python-hf` as our two student models for training. For a relative fair comparison, we compare our fine-tuned student models against publicly available academic Code LLMs, which are trained with a similar scale of SFT data and employ the same base models as ours. This includes MagiCoder-DS/CL (Wei et al., 2023), WaveCoder-DS (Yu et al., 2023), and WizardCoder-CL (Luo et al., 2024). We also compare against official instruction models, namely DeepSeek-Coder-Instruct and CodeLlama-Instruct, to showcase performance gaps. For more discussions about baseline selection and SFT details, please refer to the Appendix G.

Table 4 showcases the exceptional performance of DeepSeekCoder-AMR-Evol across all tasks. When compared to MagiCoder-DS, trained with 75k SFT data, and WaveCoder-DS, distilled from GPT4, the AMR-Evol version notably stands out by demonstrating substantial performance gains: +2.4 on HE, +3.2 on HE-Plus, and +1.0 on MBPP-Plus. Even when compared to the official instruction model, which is trained with more than 20 times as much data, our model achieves comparable performance on MBPP and MBPP-Plus. Similarly, the CodeLlama-AMR-Evol variant exhibits superior performance in most tasks, with performance improvements of +3.6 on HE, +3.0 on HE-Plus, and +1.5 on MBPP-Plus, respectively. Moreover, our model significantly outperforms CodeLlama-Instruct, which is an official model from Meta. In addition, the Pass@k sampling results, presented in Appendix G, Table 8, also evident the better performance of our models.

Model	CC Val	CC Test	APPS
DS-Instruct	7.69	6.67	11.67
MagiCoder-DS	8.55	12.73	13.00
DS-AMR-Evol	10.26	12.73	14.22

Table 5: Comparing different models on the harder code generation tasks, CodeContest (CC) (Li et al., 2022) and APPS (Hendrycks et al., 2021). DS-Instruct = DeepSeekCoder-Instruct. DS-AMR-Evol is our model.

Since HumanEval and MBPP cover basic coding tasks, we've gone further to evaluate different models on advanced coding challenges, specifically CodeContest (Li et al., 2022) and APPS (Hendrycks et al., 2021). All models generate the answers with greedy decoding. As seen in Table 5, our model not only performs better overall but also beats the official instruction model, despite it being trained on much more data than ours.

5 Conclusion

In this study, we present a novel framework, **AMR-Evol**, that leverages a two-stage approach—namely, modular decomposition and adaptive response evolution—to enhance code response distillation from teacher models, thereby improving knowledge distillation in code generation. Our experiments across three well-known coding benchmarks, HumanEval, MBPP, and EvalPlus, demonstrate the effectiveness of our method.

Acknowledgement

This work is partially supported by National Natural Science Foundation of China Young Scientists Fund(No. 62206233) and Hong Kong RGC ECS (No. 22200722).

Limitation

Our framework has room for enhancement in several aspects:

- First, despite Figure 3 showcasing our method's capacity to improve the accuracy of code response distillation, achieving 100% accuracy remains unattainable. While our approach does alleviate this concern to some extent, the risk of delivering low-quality responses that could potentially mislead the student models cannot be entirely eliminated. Future endeavors could explore the integration of tools, such as compilers, to further refine the quality of the responses.

- Second, our framework's enhanced capability for code knowledge distillation is accompanied by a requirement for multi-stage generation, leading to increased costs in leveraging the teacher models. This cost-performance trade-off has been discussed in Appendix H, where we conclude that the benefits in performance outweigh the incremental costs incurred.

- Third, the design of our method is narrowly focused on code knowledge distillation, limiting its broader application across general domains. The foundation of our framework in modular programming principles presents considerable obstacles in adapting its method for use in non-coding areas.

References

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. 2023. Claude: A family of large language models. https://www.anthropic.com/claude.

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. *arXiv preprint arXiv:2309.16609*.

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 *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Hailin Chen, Amrita Saha, Steven C. H. Hoi, and Shafiq Joty. 2023b. Personalised distillation: Empowering open-sourced llms with adaptive learning for code generation. *CoRR*, abs/2310.18628.

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.

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023d. 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. 2022. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311.

Edsger W. Dijkstra. 1967. The structure of the "the"-multiprogramming system. In *Proceedings of the First Symposium on Operating Systems Principles, SOSP 1967, Gatlinburg, Tennesse, USA, 1967*. ACM.

Google. 2024. Codegemma: Open code models based on gemma. `https://storage.googleapis.com/deepmind-media/gemma/codegemma_report.pdf`.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. 2024. Olmo: Accelerating the science of language models. *CoRR*, abs/2402.00838.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need. *CoRR*, abs/2306.11644.

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. 2021. 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*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 8003–8017. Association for Computational Linguistics.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, Xinrong Zhang, Zhen Leng Thai, Kai Zhang, Chongyi Wang, Yuan Yao, Chenyang Zhao, Jie Zhou, Jie Cai, Zhongwu Zhai, Ning Ding, Chao Jia, Guoyang Zeng, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *CoRR*, abs/2404.06395.

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.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *CoRR*, abs/2310.08992.

Kaixin Li, Qisheng Hu, Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Qizhe Xie, and Junxian He. 2024. Instructcoder: Instruction tuning large language models for code editing. *Preprint*, arXiv:2310.20329.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need II: phi-1.5 technical report. *CoRR*, abs/2309.05463.

Yujia Li, David H. 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. *CoRR*, abs/2203.07814.

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

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. *CoRR*, abs/2305.01210.

Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, and Andrew M. Dai. 2024. Best practices and lessons learned on synthetic data for language models. *CoRR*, abs/2404.07503.

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 J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 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. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Cristian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, and et al. 2024. Gemma: Open models based on gemini research and technology. *CoRR*, abs/2403.08295.

Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive learning from complex explanation traces of GPT-4. *CoRR*, abs/2306.02707.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT self-repair for code generation. *CoRR*, abs/2306.09896.

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

David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. 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.

Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Xavier Garcia, Dara Bahri, Tal Schuster, Huaixiu Steven Zheng, Neil Houlsby, and Donald Metzler. 2022. Unifying language learning paradigms. *CoRR*, abs/2205.05131.

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.

Jiahao Wang, Bolin Zhang, Qianlong Du, Jiajun Zhang, and Dianhui Chu. 2024. A survey on data selection for LLM instruction tuning. *CoRR*, abs/2402.05123.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.

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, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. 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*.

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

Xiaodong Wu, Ran Duan, and Jianbing Ni. 2023. Unveiling security, privacy, and ethical concerns of chatgpt. *CoRR*, abs/2307.14192.

Yu Xia, Rui Wang, Xu Liu, Mingyan Li, Tong Yu, Xiang Chen, Julian McAuley, and Shuai Li. 2024. Beyond chain-of-thought: A survey of chain-of-x paradigms for llms.

Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A survey on knowledge distillation of large language models. *CoRR*, abs/2402.13116.

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.

Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, Zhenghao Liu, Bowen Zhou, Hao Peng, Zhiyuan Liu, and Maosong Sun. 2024. Advancing llm reasoning generalists with preference trees. *Preprint*, arXiv:2404.02078.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. 2022. GLM-130B: an open bilingual pre-trained model. *CoRR*, abs/2210.02414.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

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.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: less is more for alignment. 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*.

A Baselines

To ensure a fair comparison, we incorporate three distinct response distillation methods as our baselines. The first method is **direct** distillation. As outlined in Section 3.1, this approach involves using the teacher model to directly produce responses based on the provided code instructions. The prompt used is as follows:

Prompt for Direct Distillation

System: You are a professional coder. Your answer must include Python code in Markdown format.

User: {instruction}

The second method involves response distillation utilizing the Chain-of-Thought (**CoT**) approach. We adopt the method from the few-shot CoT (Wei et al., 2022), prompting the teacher model to produce the responses. To minimize costs, we opt to include a single example in our prompt:

Prompt for CoT Distillation

System: You are a professional coder. You will be given a Python Question. Your objective is to develop an accurate solution to the Python Question. Begin by step-by-step think about your approach to solve this question, then proceed to generate your final code response in Markdown format.

One-Shot Example
Python Question:
{one-shot-example-question}

Correct Solution:
{one-shot-example-solution}

User: ## New Task
Python Question:
{question}

Correct Solution:

The third baseline, **AnsRepair**, incorporates self-repair techniques (Chen et al., 2023a; Olausson et al., 2023). This method employs the teacher model to generate unit test functions for each sample, enabling the model to verify the correctness of its own answers. The employed prompt is as follows:

Prompt for Test Function Generation

System: You are a professional coder. You will be given a Python Question and its possible code solution. Your objective is to provide a test function to test whether the code solution is correct or not. Your response should be in Markdown format.

One-Shot Example
Python Question:
{one-shot-example-question}

Possible Code Solution:
{one-shot-example-solution}

Tests Function:
{one-shot-example-tests}

User: ## New Task
Python Question:
{question}

Possible Code Solution:
{answer}

Tests Function:

Upon obtaining the test functions for each sample, we execute these tests to assess the output's correctness. Should the output fail to meet the criteria set by the test functions, we prompt the teacher

Data	Source	Number
Seed	MBPP-Train	332
Self-Instruct	Seed	10k
Complex 1	Self-Instruct	9.8k
Complex 2	Complex 1	9.7k
Complex 3	Complex 2	9.7k

Table 6: Statistics of Our Instruction Dataset.

Data	#Question	#Avg. Tests
HumanEval	164	9.6
HumanEval-Plus	164	x80
MBPP	399	3
MBPP-Plus	399	x35

Table 7: Statistics of Our Benchmarks.

model to regenerate the output. The prompt used for this process is as follows:

Prompt for AnsRepair Distillation

System: You are a professional coder. You will be given a Python Question and its wrong solution. You need to provide the correct solution for the Python Question in Markdown format.

One-Shot Example
Python Question:
{one-shot-example-question}

Wrong Solution:
{one-shot-example-wrong-answer}

Correct Solution:
{one-shot-example-correct-answer}

User: ## New Task
Python Question:
{question}

Wrong Solution:
{answer}

Correct Solution:

B Datasets

Our framework concentrates on distilling responses and requires a dataset of instructions for this purpose. As indicated in Table 6, we enumerate the quantity of instructions used in our experiments. We initiate our process with the MBPP training set (task-ids 601-974) as a seed dataset, which enhances our ability to generate Python code effectively. To prevent any overlap with the EvalPlus test data, we are diligent in omitting any samples that coincide with the test set, thereby narrowing our training set to 332 unique MBPP tasks. We then utilize this filtered seed data and apply the self-instruction method to construct instructions. Subsequently, we employ the Code Evol-Instruct method to iteratively generate instructions of varying complexity across three distinct levels.

To ensure decontamination of our datasets, we invoke a method akin to the work of Code Evol-Instruct (Luo et al., 2024) for data filtering. This involves employing the `gte-large-en-v1.5` model to treat each test set sample as a query, which retrieves the top five most similar samples from the training data. Subsequently, these pairs are evaluated by GPT4 in a binary classification task to decide whether a match exists. Detected matches lead to the exclusion of those specific training samples to eliminate potential data leakage.

Prompt for Modular Decomposition

System: You will be presented with a Python coding question along with a potential solution. Your task is to deconstruct the given solution into smaller, manageable modules. Each module should be clearly defined with specific function names, detailed input/output specifications, and concise function descriptions. Do NOT repeat the functions in the One-Shot Example.

One-Shot Example
Python Question:
{one-shot-example-question}

Potential Solution:
{one-shot-example-solution}

RESPONSE:
{one-shot-example-modules}

User: ## New Task
Python Question:
{question}

Potential Solution:
{answer}

RESPONSE:

C Benchmark

Table 7 details the quantity of questions along with the average number of unit tests per question across all the benchmarks utilized in our study. The license of HumanEval is MIT.[1] The license of MBPP is cc-by-4.0.[2] The license of EvalPlus is Apache-2.0.[3]

[1] https://huggingface.co/datasets/openai/openai_humaneval
[2] https://huggingface.co/datasets/google-research-datasets/mbpp
[3] https://github.com/evalplus/evalplus

D Implementation Details

Our **AMR-Evol** framework encompasses a two-stage process. In the first stage, Modular Decomposition is applied to break down the code instructions into multiple sub-modules, using the direct responses as the initial seed data. The prompt utilized for this stage is demonstrated above. During the second stage, Adaptive Response Evolution refines these decomposed sub-modules, utilizing the retrieved modules to develop the final answer. The corresponding prompt for this stage is as follows:

Prompt for Adaptive Response Evolution

System: You are a professional coder. You will be given a Python Question and a selection of relevant, modularized functions intended to inspire your approach. Your objective is to develop a more refined and accurate solution to the Python Question. Your response should pretend that you have never seen the Relevant Functions.

One-Shot Example
Python Question:
{one-shot-example-question}

Relevant Functions:
{one-shot-example-similar-functions}

Correct Solution:
{one-shot-example-solution}

User: ## New Task
Python Question:
{question}

Relevant Functions:
{similar-functions}

Correct Solution:

For all instruction construction processes, we set the temperature to 0.7 and the sequence length to 2048. For all response distillation processes, the temperature is fixed at 0.0, and the sequence length is set to 3000. We train the models for 200 steps across 3 epochs with a sequence length of 2048, employing the AdamW optimizer, BF16 precision, and DeepSpeed Zero-2 (Rasley et al., 2020). The training is conducted on 4 A800 GPUs.

E Qualitative Comparison

Table 10 11 12 display distilled responses obtained through various methods. It is evident from the comparison that our framework facilitates the generation of better responses for code knowledge distillation.

F Modular Decomposed and Retrieval Examples

Table 13 14 15 showcase the modular decomposed (MD) and retrieved top-1 (Recall) examples.

G Comparing with Open Code LLMs

To compare with other Open Code LLMs, we integrate our AMR-Evol framework with Code Evol-Instruct to continually expand our SFT dataset. We also employ the same data decontamination method to prevent data leakage. We have generated approximately 50k training samples. Subsequently, we fine-tuned our models using settings similar to those detailed in Appendix D. Given the larger volume of data, we opted to increase the number of training steps to 400.

To obtain a relative fair comparison, we only include the open code LLMs which are trained with a similar scale of SFT data and employ the same base models as ours, including MagiCoder-DS/CL, WaveCoder-DS, and WizardCoder-CL. We also compare against official instruction-based models, namely DeepSeekCoder-Instruct and CodeLlama-Instrut. However, these official models are trained with more than 20 times data than ours, which lead to unfair comparison. We only want to showcase the performance gaps.

Models with a higher parameter count have been excluded from our comparison, such as DeepSeekCoder-Instruct-33B, WizardCoder-33B-v1.1, Codestral-22B-v0.1,[4], CodeLlama-Instruct-34B, and Starcoder2-15b-Instruct.[5] These models considerably exceed the size of our own, rendering a direct comparison unfair. Additionally, models that primarily derive their learning from GPT4 are excluded, including MagiCoder-S-DS, WaveCoder-DS-Ultra, and OpenCodeInterpreter (Zheng et al.,

[4] https://huggingface.co/mistralai/Codestral-22B-v0.1

[5] https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1

Model	HE-Plus (Pass@1)	HE-Plus (Pass@10)	MBPP-Plus (Pass@1)	MBPP-Plus (Pass@10)
MagiCoder-DS	56.0	72.5	61.7	68.5
WaveCoder-DS	56.6	63.2	57.6	63.0
DS-AMR-Evol	59.1	75.2	61.3	70.7

Table 8: Results of pass@k(%) on HE-Plus, MBPP-Plus. We follow the previous works (Chen et al., 2021) to generate n=200 samples to estimate the pass@k scores our models with the same set of hyper-parameters: temperate=0.2, and top_p=0.95. DS-AMR-Evol is our model.

Teacher	HE-Plus	MBPP-Plus
GPT3.5-Turbo	61.0	62.9
Llama-3-70B	62.2	63.2

Table 9: Adopting open-source model, Llama-3-70B-Instruct, as our teacher model.

2024). As our teacher model is based on GPT-3.5, a direct comparison with these GPT4-based models would not be equitable. Non-academic models, such as CodeQwen (Bai et al., 2023), are also excluded since the methods behind their construction are not disclosed.

In Table 4, all models employ greedy decoding to generate answers for each question. To present additional results and align with some previous studies (Chen et al., 2021; Luo et al., 2024), we also display results obtained through sampling in Table 8. The temperature is set to 0.2, and the number of samples is fixed at 200. Following the method of prior work (Chen et al., 2021), we calculate the pass@1 and pass@10 scores. It is also evident that our models outperform the baseline models.

H Data Synthesis Cost Trade-off

Differing from direct distillation, our framework necessitates multi-stage response distillation, which increases the cost of using the API of the teacher model (around 4 times). However, Table 1 and 2 showcase that our method can outperformance the direct distillation over all tasks and different student models. In addition, we adopt the `gpt-3.5-turbo-1106` as our teacher model, whose API price is low. Therefore, we conclude that the benefits in performance outweigh the incremental costs incurred.

I Adopting Open-Source LLMs as Teachers

While our work primarily focuses on distilling the code generation ability from closed-source models, we also include an additional experiment using the open-source model, Llama-3-70B-Instruct, as our teacher model. Table 9 shows that our method is also effective when using the open-source model as the teacher.

J Broader Impact

Our research presents a novel framework for transferring code knowledge from closed-source LLMs to open-source LLMs. This framework is designed to generate code responses for various coding instructions during the data synthesis process. While our approach has been shown to improve response quality, as illustrated in Figure 3, it does not guarantee absolute correctness. Consequently, data generated through our method may still contain errors. It is essential to filter out these erroneous samples before deploying our approach in real-world applications to mitigate the risk of misuse.

K Manual Evaluation

In Figure 4, we present the interface used by human annotators to determine whether a given response is an appropriate answer for the coding tasks under evaluation, as shown in Figure 3. The annotators are the authors of this paper, possessing expertise in programming.

L Use Of AI Assistants

The AI assistant, GPT4-Turbo, is used solely for refining the writing of our paper.

<table>
<tr><td>Ins.</td><td>Design a function that implements the Laplace expansion theorem to calculate the determinant of a 3x3 matrix, but now with the added condition of utilizing nested for loops and ensuring that the matrix elements are limited to integers only.</td></tr>
<tr><td>Direct</td><td>

```
def calculate_determinant(matrix):
    if len(matrix) != 3 or len(matrix[0]) != 3:
        raise ValueError('Matrix must be a 3x3 matrix')
    determinant = 0
    for i in range(3):
        submatrix = [[matrix[j][k] for k in range(3) if k != i] for j in
    range(1, 3)]
        sub_determinant = calculate_determinant(submatrix)
        sign = (-1) ** i
        determinant += sign * matrix[0][i] * sub_determinant
    return determinant if len(matrix) == 2 else round(determinant)
```

(Wrong)</td></tr>
<tr><td>CoT</td><td>

```
def determinant_3x3(matrix):
    if len(matrix) != 3 or not all((len(row) == 3 for row in matrix)):
        raise ValueError('Input matrix must be a 3x3 matrix')
    determinant = 0
    for i in range(3):
        determinant += matrix[0][i] * matrix[1][(i + 1) % 3] * matrix[2][(i
     + 2) % 3]
        determinant -= matrix[0][i] * matrix[1][(i + 2) % 3] * matrix[2][(i
     + 1) % 3]
    return determinant
```

(Correct)</td></tr>
<tr><td>Ans-Repair</td><td>

```
def calculate_determinant(matrix):
    if len(matrix) != 3 or len(matrix[0]) != 3:
        raise ValueError('Matrix must be a 3x3 matrix')
    determinant = 0
    for i in range(3):
        submatrix = [[matrix[j][k] for k in range(3) if k != 0] for j in
    range(3) if j != i]
        sub_determinant = calculate_determinant(submatrix)
        sign = (-1) ** i
        determinant += sign * matrix[0][i] * sub_determinant
    return determinant if len(matrix) == 2 else round(determinant)
```

(Wrong)</td></tr>
<tr><td>AMR-Evol</td><td>

```
def determinant_3x3(matrix: list) -> int:
    det = 0
    for i in range(3):
        det += (-1) ** i * matrix[0][i] * determinant_2x2(submatrix_2x2(
    matrix, 0, i))
    return det

def determinant_2x2(matrix: list) -> int:
    return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]

def submatrix_2x2(matrix: list, i: int, j: int) -> list:
    submatrix = [row[:j] + row[j + 1:] for row in matrix[:i] + matrix[i +
    1:]]
    return submatrix
```

(Correct)</td></tr>
</table>

Table 10: Examples of different response distillation methods.

<table>
<tr><td>Ins.</td><td>Develop a algorithm to simulate multiple rounds of a custom coin flip game with distinct scoring mechanisms for each outcome. The algorithm should efficiently manage a substantial number of players and monitor their scores in real-time, while prioritizing minimal memory usage and computational complexity.</td></tr>
<tr><td>Direct</td><td>

```python
import random
def simulate_round():
    return random.choice(['heads', 'tails'])
def simulate_game(num_players, num_rounds):
    players = {i: 0 for i in range(1, num_players + 1)}
    for _ in range(num_rounds):
        outcome = simulate_round()
        for player in players:
            players[player] = scoring_mechanisms[outcome](players[player])
        print('Scores after round', _ + 1, ':', players)
    return players
```

</td></tr>
<tr><td>CoT</td><td>

```python
import random
def coin_flip_game(players):
    scores = {}
    for player in players:
        scores[player] = 0
    rounds = 10
    for _ in range(rounds):
        for player in players:
            outcome = random.choice(['heads', 'tails'])
            if outcome == 'heads':
                scores[player] += heads_score
            else:
                scores[player] += tails_score
    return scores
```

</td></tr>
<tr><td>Ans-Repair</td><td>

```python
import random
def simulate_round():
    return random.choice(['heads', 'tails'])

def simulate_game(num_players, num_rounds):
    players = {i: 0 for i in range(1, num_players + 1)}
    scoring_mechanisms = {'heads': lambda x: x + 1, 'tails': lambda x: max
    (0, x - 1)}
    for _ in range(num_rounds):
        outcome = simulate_round()
        for player in players:
            players[player] = scoring_mechanisms[outcome](players[player])
        print('Scores after round', _ + 1, ':', players)
    return players
```

</td></tr>
<tr><td>AMR-Evol</td><td>

```python
import random
class CoinFlipGame:
    def __init__(self):
        self.scores = {}

    def coin_flip(self, num_flips=10):
        heads_count = sum((random.randint(0, 1) for _ in range(num_flips)))
        if heads_count < num_flips / 2:
            return 'You lost the game!'
        else:
            return 'You won the game!'

    def update_scores(self, player, amount):
        if player not in self.scores:
            self.scores[player] = 0
        self.scores[player] += amount

    def get_scores(self):
        return self.scores
```

</td></tr>
</table>

Table 11: Examples of different response distillation methods. Our AMR-Evol leads more suitable response.

<table>
<tr><td>Ins.</td><td>Craft a program tasked with computing the Fibonacci sequence up to a user-defined boundary, ensuring that every generated number is not only part of the Fibonacci sequence but also a prime number.</td></tr>
<tr><td>Direct</td><td>

```
def is_prime(n):
    ...

def fibonacci_prime_sequence(boundary):
    fib_sequence = [0, 1]
    while fib_sequence[-1] <= boundary:
        next_fib = fib_sequence[-1] + fib_sequence[-2]
        if is_prime(next_fib):
            fib_sequence.append(next_fib)
        else:
            fib_sequence.append(next_fib + 1)
    return fib_sequence[:-1] # Wrong
```

</td></tr>
<tr><td>CoT</td><td>

```
def is_prime(num):
    ...

def fibonacci_prime_sequence(boundary):
    fib_sequence = [1, 1]
    while fib_sequence[-1] + fib_sequence[-2] <= boundary:
        next_fib = fib_sequence[-1] + fib_sequence[-2]
        if is_prime(next_fib):
            fib_sequence.append(next_fib)
        else:
            fib_sequence.append(next_fib)
    return fib_sequence # Wrong
```

</td></tr>
<tr><td>Ans-Repair</td><td>

```
def is_prime(num):
    ...

def fibonacci_prime_sequence(boundary):
    def fibonacci(n):
        if n <= 0: return 0
        elif n == 1: return 1
        else: return fibonacci(n - 1) + fibonacci(n - 2)
    fib_sequence, i = [], 0
    while fibonacci(i) <= boundary:
        next_fib = fibonacci(i)
        if is_prime(next_fib):
            fib_sequence.append(next_fib)
        i += 1
    return fib_sequence # Correct
```

</td></tr>
<tr><td>AMR-Evol</td><td>

```
from typing import List
def is_prime(num: int) -> bool:
    ...

def generate_fibonacci_sequence(boundary: int) -> List[int]:
    fibonacci_sequence = [0, 1]
    while True:
        next_num = fibonacci_sequence[-1] + fibonacci_sequence[-2]
        if next_num <= boundary:
            fibonacci_sequence.append(next_num)
        else:
            break
    return fibonacci_sequence

def find_fibonacci_primes(boundary: int) -> List[int]:
    fibonacci_sequence = generate_fibonacci_sequence(boundary)
    prime_fibonacci_numbers = [num for num in fibonacci_sequence if
    is_prime(num)]
    return prime_fibonacci_numbers # Correct
```

</td></tr>
</table>

Table 12: Examples of different response distillation methods. The `is_prime` has been omitted to save space.

Ins.	Craft a program tasked with computing the Fibonacci sequence up to a user-defined boundary, ensuring that every generated number is not only part of the Fibonacci sequence but also a prime number.
Direct	See Table 10

MD	<pre>def validate_matrix(matrix: list) -> None:
	    """
	    Description:
	        Validates if the input matrix is a 3x3 matrix.
	    ...
	    """
	
	def calculate_minor_matrix(matrix: list, row: int, col: int) -> list:
	    """
	    Description:
	        Calculates the minor matrix by removing the specified row and
	    column from the input matrix.
	    ...
	    """
	
	def calculate_determinant(matrix: list) -> int:
	    """
	    Description:
	        Calculates the determinant of a 3x3 matrix using Laplace expansion
	    theorem.
	    ...
	    """</pre>

Recall	<pre>def search_element(matrix, x):
	    """
	    Search for a given element in a sorted matrix.
	    ...
	    """
	    # Start from the top right corner
	    i = 0
	    j = len(matrix[0]) - 1
	    while (i < len(matrix) and j >= 0):
	        if (matrix[i][j] == x):
	            return True
	
	        if (matrix[i][j] > x):
	            j -= 1
	        else:
	            i += 1
	    return False
	
	def Submatrix(A: list, i: int, j: int) -> list:
	    """
	    Get the submatrix of A by removing the i-th row and j-th column.
	    ...
	    """
	    return [row[:j] + row[j+1:] for row in (A[:i] + A[i+1:])]
	
	def Determinant(A: list) -> int:
	    """
	    Calculate the determinant of the provided matrix A.
	    ...
	    """
	    if len(A) == 1:
	        return A[0][0]
	    if len(A) == 2:
	        return A[0][0]*A[1][1] - A[0][1]*A[1][0]
	    det = 0
	    for j in range(len(A)):
	        det += (-1) ** j * A[0][j] * Determinant(Submatrix(A, 0, j))
	    return det</pre>

Table 13: Examples of the modular decomposed (MD) functions and the retrieved top-1 (Recall) functions. We omit some function descriptions to save space.

Ins.	Develop a algorithm to simulate multiple rounds of a custom coin flip game with distinct scoring mechanisms for each outcome. The algorithm should efficiently manage a substantial number of players and monitor their scores in real-time, while prioritizing minimal memory usage and computational complexity.
Direct	See Table 11

MD

```python
def simulate_coin_flip() -> str:
    """
    Description:
        Simulates a single coin flip and returns the outcome.
    ...
    """

def update_player_scores(players: dict, outcome: str, scoring_mechanisms:
    dict) -> None:
    """
    Description:
        Updates the scores of all players based on the outcome of the coin
    flip.
    ...
    """

def simulate_multiple_rounds(num_players: int, num_rounds: int) -> dict:
    """
    Description:
        Simulates multiple rounds of the game for a given number of players
    .
    ...
    """
```

Recall

```python
import random
def coin_flip():
    """Simulate a game of coin flip by flipping a coin 10 times and
    determining the outcome based on the number of heads. ..."""
    result = 0
    for x in range(10):
        n = random.randint(0, 1)
        if n == 0:
            result += 1
    if result < 5:
        return "You lost the game!"
    else:
        return "You won the game!"

def score_transactions(transactions):
    """Calculate the total amount of transactions for each sender and store
     the scores in a dictionary. ..."""
    scores = {}
    for transaction in transactions:
        if transaction['sender'] not in scores:
            scores[transaction['sender']] = 0
        scores[transaction['sender']] += transaction['amount']
    return scores

def determine_winner(scores: list) -> str:
    """Determine the winner of a match based on the scores provided. ..."""
    team_names = [item[0] for item in scores]
    point_diffs = [abs(item[1] - item[2]) for item in scores]
    max_point_diff_idx = point_diffs.index(max(point_diffs))
    if scores[max_point_diff_idx][1] > scores[max_point_diff_idx][2]:
        return team_names[max_point_diff_idx]
    else: return 'Draw'
```

Table 14: Examples of the modular decomposed (MD) functions and the retrieved top-1 (Recall) functions. We omit some function descriptions to save space.

<table>
<tr><td>Ins.</td><td>Craft a program tasked with computing the Fibonacci sequence up to a user-defined boundary, ensuring that every generated number is not only part of the Fibonacci sequence but also a prime number.</td></tr>
<tr><td>Direct</td><td>See Table 12</td></tr>
<tr><td>MD</td><td>

```python
def is_prime(n: int) -> bool:
    """
    Description:
        Checks if a number is a prime number.
    Parameters:
        - n (int): The number to be checked for primality.
    Returns:
        - bool: True if the number is prime, False otherwise.
    """

def fibonacci_prime_sequence(boundary: int) -> List[int]:
    """
    Description:
        Generates a Fibonacci sequence of prime numbers up to a user-
    defined boundary.
    Parameters:
        - boundary (int): The upper limit for the Fibonacci sequence.
    Returns:
        - List[int]: A list of prime numbers within the Fibonacci sequence
    up to the boundary.
    """
```

</td></tr>
<tr><td>Recall</td><td>

```python
def is_prime(num: int) -> bool:
    """
    Check if the provided number is a prime number.
    Args:
        num (int): The number to check for primality.
    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if num <= 1:
        return False

    for i in range(2, num):
        if (num % i) == 0:
            return False
    return True

def prime_sequence(start: int, end: int) -> List[int]:
    """
    Generate a list of prime numbers within the specified range from start
    to end (inclusive).
    Args:
        start (int): The starting value of the range.
        end (int): The ending value of the range.
    Returns:
        List[int]: A list of prime numbers within the specified range.
    """
    prime_list = []
    for num in range(start, end + 1):
        if num > 1:
            for i in range(2, num):
                if (num % i) == 0:
                    break
            else:
                prime_list.append(num)
    return prime_list
```

</td></tr>
</table>

Table 15: Examples of the modular decomposed (MD) functions and the retrieved top-1 (Recall) functions.

Create a Python function that generates a random 10-digit number within a specified range without using

direct

```
import time

def generate_divisible_random_number(start_range, end_range):
    start_time = int(time.time())
    seed = start_time % 1000000
    a = 1664525
    c = 1013904223
    m = 2 ** 32
    random_num = (a * seed + c) % m
    random_num = start_range + random_num % (end_range - start_range)
    while random_num % 3 != 0:
        random_num = (random_num + 1) % (end_range - start_range) + start_range
    return random_num
```

Correct ☐

cot

```
import time

def generate_divisible_by_3():
    start_range = 10 ** 9
    end_range = 10 ** 10 - 1
    while True:
        random_seed = int(time.time() * 1000 % (end_range - start_range + 1) + start_range)
        if random_seed % 3 == 0:
            return random_seed
```

Correct ☐

repair

```
def generate_divisible_random_number(start_range, end_range):
    num = int(str(time.time()).replace('.', '')[-10:])
    while num < start_range or num > end_range or num % 3 != 0:
        num = (num * 1664525 + 1013904223) % 2 ** 32
    return num
```

Correct ☐

evol

```
def generate_divisible_by_3(start: int, end: int) -> int:
    """
    Generate a random 10-digit number within the specified range that is divisible by 3.

    Args:
        start (int): The lower limit of the range.
        end (int): The upper limit of the range.

    Returns:
        int: A random 10-digit number within the specified range that is divisible by 3.
    """
    while True:
        random_num = int(''.join((str(random.randint(start, end)) for _ in range(10))))
        if random_num % 3 == 0:
            return random_num
```

Correct ☐

Previous | Next | Export Modified JSON

Figure 4: Screenshot of the interface for the human annotators to annotate whether the responses are suitable or not.