MoTCoder: Elevating Large Language Models with Module-of-Thought

Anonymous authors
Paper under double-blind review

Abstract

Large Language Models (LLMs) have showcased impressive capabilities in handling straightforward programming tasks. However, their performance tends to falter when confronted with more challenging programming problems. We observe that conventional models often generate solutions as monolithic code blocks, restricting their effectiveness in tackling intricate questions. To overcome this limitation, we present Module-of-Thought Coder (MoTCoder). We introduce a framework for MoT instruction tuning, designed to promote the decomposition of tasks into logical sub-tasks and sub-modules. Our investigations reveal that, through the cultivation and utilization of sub-modules, MoTCoder significantly improves both the modularity and correctness of the generated solutions, leading to substantial *pass@1* improvements of 5.8% on APPS and 5.9% on CodeContests. MoTCoder also achieved significant improvements in self-correction capabilities, surpassing the current SOTA by 3.3%. Additionally, we provide an analysis of between problem complexity and optimal module decomposition and evaluate the maintainability index, confirming that the code generated by MoTCoder is easier to understand and modify, which can be beneficial for long-term code maintenance and evolution. Our code will be released.

1 Introduction

Developing systems that can generate executable and functionally correct computer programs has long been sought after in artificial intelligence (Manna & Waldinger, 1971). Recently, Large Language Models (LLMs) (Brown et al., 2020; OpenAI, 2023a; Chowdhery et al., 2022; Anil et al., 2023; Hoffmann et al., 2022; Rae et al., 2021; Zeng et al., 2022; Touvron et al., 2023; Zhang et al., 2022) have showcased remarkable success in many problem domains beyond natural language processing, and is poised as a promising approach to tackle code modeling and generation (as well as other coding related tasks) (Roziere et al., 2023; Black et al., 2021; Chen et al., 2021). Through instruction fine-tuning (Li et al., 2023b; Luo et al., 2023b; Wang et al., 2023b; Li et al., 2022; Nijkamp et al., 2023; Zheng et al., 2023; Fried et al., 2022; Chen et al., 2021; Wang et al., 2021), LLMs have achieved impressive performance on code generation benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021).

Yet, when confronted with intricate coding problems such as APPS (Hendrycks et al., 2021b) and CodeContests (Li et al., 2022), Current models struggle to match seasoned developers (Hendrycks et al., 2021b; Li et al., 2022; Shinn et al., 2023). The main culprit is their overly simplistic generation approach. Current models produce the code solution as a single monolithic block in an attempt to solve the problem in one shot. While feasible for simple tasks, this approach becomes increasingly inefficient to tackle a complex task which entails solving multiple sub-tasks. In contrast, adept developers often devise modularized solutions by breaking down the original problem into more approachable components that can be individually and more efficiently solved.

Following this intuition, recent works (Jiang et al., 2023a; Le et al., 2023) propose iterative code inference for code generation. They, however, come with added inference costs and offer only marginal performance gains. In this work, we propose to more efficiently improve the modularization capability of coding LLMs using Module-of-Thought (MoT) instruction fine-tuning. Our approach guides LLMs to break down their solution into modular components, each representing an abstract function dedicated to a logical sub-task. To train the model to adhere to the MoT prompt, we generate instructional data using a process termed MoT Code Instruction Transformation. In this process, LLMs are instructed to outline necessary modules, generating only their function headers and docstrings that describe their intended usage. Subsequently, the instruction guides the model to implement these modules and eventually combine them into the final solution. After that, we fine-tune the LLM with our MoT instruction fune-tuning, resulting in our MoTCoder model.

Our experiments demonstrate that MoTCoder establishes new SOTA results on challenging code generation benchmarks such as APPS and CodeContests. Specifically, MoTCoder improves the *pass@1* performance by significant margins, exemplified by improvements over existing metrics by 5.8% on APPS and 5.9% on CodeContests, as illustrated in Fig. 1. MoTCoder also achieved significant improvements in self-correction capabilities, surpassing the current SOTA by 3.3%. Beyond the pass@k metric, we have conducted a detailed analysis and comprehensive evaluation of MoTCoder. Furthermore, we analyze the relationship between problem complexity and optimal module decomposition, confirming that finer modular decomposition is beneficial for enhancing model performance in complex problems. This helps explain why the MoT approach is more effective than traditional methods for complex programming challenges. We also perform a quantitative analysis on the time and memory usage across different problem scales, verifying that our approach can reduce memory consumption. Moreover, through the evaluation of the maintainability index, we confirm that the code generated by MoTCoder is easier to understand and modify, which can be beneficial for long-term code maintenance and evolution.

In summary, our contributions are threefold:

1. We propose a 2-step *Module-of-Thought Code Instruction Transformation* approach that instructs LLMs to generate modularized code solutions.
2. We develop Module-of-Thought Coder (MoTCoder), a new model that enhances the modularization capabilities of LLMs with *MoT Instruction Tuning*. It breaks down complex problems into sub-modules, and it has been validated to improve the model's maintainability index.
3. Our approach not only achieves SOTA performance on challenging programming tasks, including APPS and CodeContests, but also demonstrates strong self-repair capabilities.

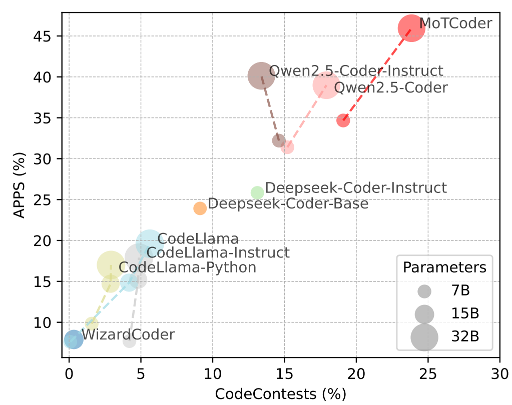


Figure 1: *Pass@1* results on CodeContests (x-axis) and APPS (y-axis). Comparison of our MoTCoder with previous SOTAs. Model size are indicated by scatter size.

2 Related Works

General LLMs. In recent times, Large Language Models (LLMs) have exhibited remarkable prowess across a wide array of tasks. Leading technology companies have made significant advancements in developing highly proficient closed-source LLMs, including OpenAI's GPT3 (Brown et al., 2020) and GPT4 (OpenAI, 2023a), Google's PaLM (Chowdhery et al., 2022; Anil et al., 2023), Bard, DeepMind's Chinchilla (Hoffmann et al., 2022), and Gopher (Rae et al., 2021), as well as Anthropic's Claude. The AI community has also observed the release of several open-source LLMs, where model weights are made publicly available. EleutherAI has contributed GPT-NeoX-20B (Black et al., 2022) and GPT-J-6B (Wang & Komatsuzaki, 2021). Google has released UL2-20B (Tay et al., 2022). Tsinghua University has introduced GLM-130B (Zeng et al., 2022). Meta has released OPT (Zhang et al., 2022) and LLaMA (Touvron et al., 2023). Recently, the Qwen series (Bai et al., 2023; Yang et al., 2024; Qwen et al., 2025) and DeepSeek series (DeepSeek-AI et al., 2024a;b; 2025) models have emerged as significant contributors, achieving SOTA performance across a variety of benchmarks.

Coding LLMs. Recent research has introduced a significant number of LLMs tailored for code-related tasks to address the challenges of code understanding and generation. Closed-source models include OpenAI's Codex (Chen et al., 2021) and Code-Davinci (Microsoft, 2023). Google has proposed PaLM-Coder (Chowdhery et al., 2022). These models excel on popular code completion benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). On the open-source front, Salesforce has introduced CodeGen (Nijkamp et al., 2023), CodeT5 (Wang et al., 2021), and CodeT5+ (Wang et al., 2023a). Tsinghua University has contributed CodeGeeX (Zheng et al., 2023), and the BigCode Project has developed StarCoder (Li et al., 2023b). Furthermore, the latest DeepSeek-Coder series (Guo et al., 2024; DeepSeek-AI et al., 2024b) and Qwen-Coder (Hui et al., 2024) models have set new standards by achieving SOTA results on multiple coding benchmarks.

General Instruction Tuning. In its early stages, the core aim of instruction fine-tuning was to amplify the cross-task generalization capabilities of Language Models (LMs). This was accomplished by subjecting LMs to fine-tuning using an extensive corpus of public Natural Language Processing (NLP) tasks. Pioneering this approach, T5 (Raffel et al., 2020) underwent training on a diverse set of supervised text-to-text tasks. Subsequent endeavors like FLAN (Wei et al., 2022a), ExT5 (Aribandi et al., 2022), T0 (Sanh et al., 2022), and UnifiedQA (Khashabi et al., 2020) broadened the spectrum of tasks, fortifying the overall generalization capability of LMs. Noteworthy contributions from ZeroPrompt (Xu et al., 2022) and FLAN-T5 (Chung et al., 2022) pushed boundaries by incorporating thousands of tasks into their training pipelines. OpenAI has taken an alternative route by enlisting human annotators to contribute an extensive corpus of human instructions, encompassing diverse formats and a broad spectrum of task types. Building upon this dataset, OpenAI trained its GPT-3 (Brown et al., 2020) model to create InstructGPT (Ouyang et al., 2022), which better aligns with users' inputs. This developmental trajectory has given rise to notable works such as ChatGPT. In the open-source realm, Alpaca (Taori et al., 2023) adopts the self-instruct method (Wang et al., 2022), leveraging ChatGPT to generate data for training. Vicuna (Chiang et al., 2023) utilizes user-shared conversations collected from ShareGPT.com to train its models. Introducing the Evol-Instruct method, WizardLM (Xu et al., 2023) involves evolving existing instruction data to generate more intricate and diverse datasets.

Chain-of-Thought Instruction Tuning. Contrary to general methods, recent research (Luo et al., 2023b; Yue et al., 2023; Chen et al., 2022b; Gunasekar et al., 2023; Haluptzok et al., 2023) employs instruction tuning in various domains such as common-sense reasoning (West et al., 2022), text-summarization (Sclar et al., 2022), and mathematical reasoning (Luo et al., 2023a; Yue et al., 2023). It's also applied in tool use (Patil et al., 2023), coding (Luo et al., 2023b), and universal reasoning (Li et al., 2023a; Zelikman et al., 2022). Among them, (Yue et al., 2023) offers a diverse math problem corpus with annotations similar to our module-of-thought, using chain-of-thought or program-of-thought (Chen et al., 2022b). Gunasekar et al. (2023) suggests pre-training models on artificially created programming textbooks from GPT3.5. In a similar vein, (Haluptzok et al., 2023) generates coding puzzles and their solutions using language models.

Prompting Techniques. The Chain of Thought (CoT) technique (Wei et al., 2022b) introduces an approach for language reasoning tasks by generating intermediate reasoning steps before providing the final answer. Subsequent approach least-to-most prompting (Zhou et al., 2022), simplifies a complex problem into a sequence of smaller sub-problems, solving them sequentially and incorporating the solution of each preceding sub-problem into the prompt for the next. Furthermore, PAL (Gao et al., 2022) and PoT (Chen et al., 2022a) use code generation to create intermediate reasoning steps. Similar methods are proposed for simple mathematical (Lewkowycz et al., 2022; Wu et al., 2022), commonsense (Sanh et al., 2022; Madaan et al., 2022), symbolic reasoning (Yao et al., 2023) and code generation problems (Jiang et al., 2023b; Le et al., 2023). However, these works can only plan code during generation. In comparison, our approach introduces a guided module-of-thought framework during training, making it more intrinsic. Our investigations reveal that, through the cultivation and utilization of sub-modules, MoTCoder significantly enhances both the modularity and correctness of the generated solutions.

3 METHODS

In this section, we first detail the module-of-thought instruction transformation in Sec. 3.1 and then introduce the module-of-thought instruction tuning strategy for our MoTCoder in Sec. 3.2.

Instruction

You are given three strings s1, s2, and s3. You have to perform the following operation on these three strings as many times as you want. In one operation you can choose one of these three strings such that its length is at least 2 and delete the rightmost character of it. Return the minimum number of operations you need to perform to make the three strings equal if there is a way to make them equal, otherwise, return -1.

Example 1:
Input: s1 = "abc", s2 = "abb", s3 = "ab"
Output: 2

Example 2:
Input: s1 = "dac", s2 = "bac", s3 = "cac"
Output: -1

Plain Code

```
if __name__ == '__main__':
    s = input()
    if s == s[::-1]:
        print(s)
    else:
        start, end = 0, 0
        for i in range(len(s)):
            left, right = i, i
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            if right - left - 1 > end - start:
                start = left + 1
                end = right - 1
            left, right = i, i + 1
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            if right - left - 1 > end - start:
                start = left + 1
                end = right - 1
        print(s[start:end+1])
```

Step 1: Define the sub-modules

```
def expandAroundCenter(s, left, right):
    """
    Expands around the center given by indices 'left' and 'right' while the characters at 'left' and 'right' are the same.
    Args:
    s (str): Input string.
    left (int): Left index of the expanding range.
    right (int): Right index of the expanding range.
    Returns:
    tuple: A tuple containing the updated left and right indices of the expanding range.
    """
```

```
def update_indices_if_longer(left, right, start, end):
    """
    Updates start and end indices if the current palindrome is longer.
    Args:
    left (int): Left index of the expanding range.
    right (int): Right index of the expanding range.
    start (int): Current start index of the longest palindrome.
    end (int): Current end index of the longest palindrome.
    Returns:
    tuple: A tuple containing the updated start and end indices of the longest palindrome.
    """
```

Step 2: Develop and integrate the sub-modules to form a solution.

```
def expandAroundCenter(
    s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return left, right

def update_indices_if_longer(
    left, right, start, end):
    if right - left - 1 > end - start:
        start = left + 1
        end = right - 1
    return start, end
```

```
def longestPalindrome(s):
    if s == s[::-1]:
        return s
    start, end = 0, 0
    for i in range(len(s)):
        left, right = expandAroundCenter(s, i, i)
        start, end = update_indices_if_longer(left, right, start, end)
        left, right = expandAroundCenter(s, i, i+1)
        start, end = update_indices_if_longer(left, right, start, end)
    return s[start:end+1]
```

Figure 2: Illustration of our 2-step Module-of-Thought Instruction Transformation: Plain code utilizes a single module to directly generate code. In contrast, our Module-of-Thought Instruction Transformation first instructs LLMs to outline necessary sub-modules, generating only their function headers and docstrings that describe their intended usage. Subsequently, the instructions guide the model in implementing these sub-modules and eventually combining them into a comprehensive final solution.

3.1 MoT Instruction Transformation

In this section, we first introduce the normal instruction, followed by our module-of-thought instruction and its assessment.

Normal Instruction. In general, a code sequence generated by a language model θ through the autoregressive sampling of tokens $\hat{o}_t$ is from the parameterized conditional distribution:

$$o_t \sim p_\theta(.|o_{1:t-1}, I), \tag{1}$$

where I represents the input instruction and o_t is the t-th token of the flattened output sequence.

Module-of-Thought Instruction. According to previous researches (Jain et al., 2023), straightforward and unambiguous, detailed instructions enhance the model's efficacy and precision in executing the desired tasks. Hence, for intricate data transformation tasks, decomposing the task into simpler, sequential steps yields better outcomes. Therefore, our proposed methodology aims to transform normal instructions into a sequential code generation process by leading the models through a two-step procedure, as illustrated in Fig. 2 .

1. **Sub-modules.** Initially, the models are instructed to outline the required sub-modules, generating only their function headers and docstrings describing their intended usage.

$$\hat{S}_i \sim p_\theta(.|\hat{S}_{1:i-1}, I), \tag{2}$$

   where $\hat{S}_i$ represents the i-th sub-module outlined by the model and I represents the input instruction.
2. **Final solution.** The subsequent instruction guides the model to implement these sub-modules and eventually combine them into a comprehensive final solution.

$$\hat{o}_t \sim p_\theta(.|\hat{o}_{1:t-1}, \{\hat{S}_i\}, I), \tag{3}$$

   where $\hat{o}_t$ is the t-th of the flattened output sequence.

The instruction is supplemented with a one-shot example, serving to prompt the model to adhere to the MoT instruction generation strategy. An illustration of the instruction prompt is presented in the appendix. The instruction encourages the model to decompose a program into sub-modules. This mirrors the methodology commonly employed by developers when addressing intricate coding tasks, where they systematically break down solutions into modular components.

Instruction Assessment. Throughout these transformations, we introduce guidelines for reviewing our transformed module-of-thought code. We identify the situations below as markers of instruction refinement failure:

1. The refined instruction diverges from the module-of-thought generation strategy, not adhering to the protocol of initial sub-module creation followed by the main code development.
2. At the sub-module creation phase, if no sub-modules are formed or if overarching code is developed instead.
3. During the main code development phase, the absence of main code creation or the emergence of multiple main code blocks indicates a problem.
4. The presence of test cases in the dataset that the transformed program fails to pass. This criterion ensures the transformed programs preserve functional equivalence with the original codes.

3.2 Module-of-Thought Instruction Tuning

MoT Dataset. Our queries are sourced from the training sets of APPS (Hendrycks et al., 2021b) and CodeContests (Li et al., 2022). There is overlap between the CodeContests training set and the APPS test set, so we performed detailed deduplication to prevent test data leakage. For each problem, we take at most 100 answers. We then use GPT4o to generate transformed instructions. We include two types of data: clean and Module-of-Thought (MoT). The goals for clean data are: 1. Optimize variable names to better reflect their purpose. 2. Add comments. 3. Follow the instructions and the meaning of the original code without changing its functionality. MOT data additionally uses functions if there are code segments that are functionally clear and reusable. All generated data are tested using input-output examples from the training set, and any data not passing the test are discarded. As a result, we have collected a total of 183K clean code data and 174K MoT data for our final training dataset. Detailed statistics of the data are shown in the Appendix.

MoT Instruction Tuning. Our MoTCoder underwent instruction tuning utilizing the SOTA coding model Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) across one epoch on our proposed MoT instruction dataset, and the best-performing model during the training process was selected. The model's maximum input length was 2048 tokens. We used a training batch size of 16 and an evaluation batch size of 4 per device, with gradient accumulation steps set to 4. The learning rate was initialized at 2×10^{-6}, with a warmup ratio of 0.03 steps to gradually adapt the learning rate, following a cosine learning rate scheduler for optimization. Additionally, our model was configured with the AdamW optimizer and utilized the WarmupLR scheduler to manage the learning rate adjustments effectively. To optimize memory and compute resources, we employed a third-stage zero optimization setting and enable communication overlap, contiguous gradients, and large sub group size of 1×10^9 to

Model	Size	Introductory	Interview	Competition	All
Qwen2.5-Coder-Instruct	7B	50.58	30.32	19.49	32.21
Normal Finetuning	7B	45.36	25.74	15.92	27.70
MoT Finetuning	7B	**54.26**	**32.63**	**21.18**	**34.67**

Table 1: APPs test results by *pass@1* (%) of the ablation experiment on training datasets comparing MoT and normal finetuning.

Model	Size	Introductory	Interview	Competition	All
CodeT5	770M	6.60	1.03	0.30	2.00
CodeRL+CodeT5	770M	7.08	1.86	0.75	2.69
text-davinci-002	-	-	-	-	7.48
Self-edit+text-davinci-002	-	-	-	-	7.94
GPT-2	0.1B	5.64	6.93	4.37	6.16
	1.5B	7.40	9.11	5.05	7.96
GPT-Neo	2.7B	14.68	9.85	6.54	10.15
GPT-3	175B	0.57	0.65	0.21	0.55
StarCoder	15B	7.25	6.89	4.08	6.40
WizardCoder	15B	26.04	4.21	0.81	7.90
CodeChain+WizardCoder	15B	26.29	7.49	3.75	10.50
Octocoder	16B	16.50	7.92	4.61	8.97
CodeLlama	7B	14.15	6.63	4.00	7.61
	13B	23.94	13.50	9.80	14.85
	34B	32.01	18.61	10.19	19.61
CodeLlama-Python	7B	18.83	8.62	4.47	9.83
	13B	26.40	13.44	6.86	14.72
	34B	26.45	16.61	8.77	17.01
CodeLlama-Instruct	7B	14.20	6.63	4.43	7.70
	13B	22.41	14.34	6.62	15.21
	34B	28.64	16.80	10.51	17.91
Deepseek-Coder-Base	6.7B	40.23	22.12	13.04	23.92
Deepseek-Coder-Instruct	6.7B	44.65	23.86	12.89	25.83
Qwen2.5-Coder	7B	51.33	29.37	17.54	31.40
	32B	61.00	37.47	21.50	38.98
Qwen2.5-Coder-Instruct	7B	50.58	30.32	19.49	32.21
	32B	60.72	38.60	24.11	40.13
MoTCoder	7B	54.26	32.63	21.18	34.67
	32B	**68.44**	**44.49**	**27.84**	**45.95**
GPT4o	-	78.53	55.57	31.46	55.34

Table 2: APPS test results by *pass@k* (%).

.

streamline the training process. In addition, the maximum live parameters, maximum reuse distance, and the parameter settings for gathering 16-bit weights during model saving were all set to 1×10^9.

4 Experiments

We demonstrate the efficacy of MoTCoder in tackling intricate code-generation tasks, rather than those that can be solved with just a few lines, as exemplified by HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Specifically, we focus on the following prominent benchmarks.

1. **APPS** (Hendrycks et al., 2021b) is a description-to-code generation benchmark from competitive programming platforms Codewars, AtCoder, Kattis, Codeforces, etc. Building upon prior research (Hendrycks et al., 2021b; Chen et al., 2021; Li et al., 2022), we conducted an assessment of the models using the passing rate metric *pass@k*. This metric is defined as the proportion of

Model	Size	Test	Validation	All
WizardCoder	15B	0.44	0.17	0.33
CodeLlama-Instruct	7B	5.57	2.26	4.20
	13B	5.52	3.80	4.81
	34B	5.57	3.87	4.86
CodeLlama-Python	7B	1.80	1.24	1.57
	13B	4.33	0.92	2.89
	34B	3.88	1.50	2.92
CodeLlama	7B	0.13	0.00	0.08
	13B	7.62	2.80	4.18
	34B	5.13	2.85	5.62
Deepseek-Coder-Base	6.7B	11.48	5.78	9.12
Deepseek-Coder-Instruct	6.7B	15.25	10.10	13.11
Qwen2.5-Coder	7B	16.41	13.49	15.20
	32B	20.41	14.41	17.92
Qwen2.5-Coder-Instruct	7B	15.45	13.42	14.61
	32B	12.67	14.40	13.39
MoTCoder	7B	20.77	16.72	19.09
	32B	**26.34**	**20.35**	**23.85**
GPT4o	-	29.76	30.42	30.03

Table 3: CodeContests test and validation (valid) results by *pass@k* (%).

Model	Size	Validation	Test	All
Qwen2.5-Coder-Instruct	7B	13.42	15.45	14.61
Qwen2.5-Coder-Instruct + Self-Reflection	7B	19.88	19.74	19.80
MoTCoder	7B	**16.72**	**20.77**	**19.09**
MoTCoder + Self-Reflection	7B	**21.63**	**24.10**	**23.08**

Table 4: Model performance comparing with self-reflection on CodeContests.

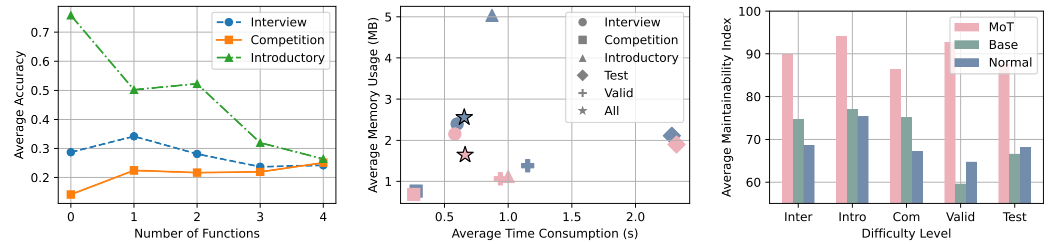
problems successfully solved by employing *k* generated programs for each problem. More details are in the appendix.

2. **CodeContests** (Li et al., 2022) is a competitive programming dataset sourced from Aizu, AtCoder, CodeChef, Codeforces, HackerEarth, etc. Building upon prior research (Hendrycks et al., 2021b; Chen et al., 2021; Li et al., 2022), we conducted an assessment of the models using the passing rate metric *pass@k*.

4.1 Ablation Experiments

In this section, we conduct ablation experiments to investigate the effects of MoT and normal finetuning. We use the Qwen2.5-Coder-7B-Instruct model as the base model. To control variables, we apply the same parameters and instructions during finetuning for both approaches. For ground truth, we use our constructed MoT code for the MoT finetuning, and standard code from the APPS and CodeContests training datasets for the normal finetuning. The results of these experiments are presented in Tab. 1. It is intriguing to note that, for a well-trained model like Qwen2.5-Coder-7B-Instruct, finetuning with uncurated normal data does not further enhance performance; rather, it leads to a decline, with scores dropping from 32.21 to 27.70 in terms of *pass@1* accuracy. In contrast, when performed with our MoT finetune, the model exhibits significant improvements across all levels, as evidenced by the enhanced scores. Overall, the application of MoT finetuning results in an improvement from 32.21% to 34.67% in the overall *pass@1* accuracy. This increase highlights the capacity of our MoT method to further augment the proficiency of a well-trained model.

Results on APPS. We conducted a comparison of our approach with existing large language model baselines on APPs (Hendrycks et al., 2021a). All outcomes are computed using raw predictions without being filtered by the test cases provided in the prompt. Our analysis includes a comparison



(a) (b) (c)

Figure 3: (a) Relationship between the number of functions and accuracy for solutions generated by MoTCoder across different difficulty levels in the APPS dataset. (b) Average time and memory consumption for the passed output for MoT (pink) and Normal (blue). (c) Average maintainability index for MoT finetuning, normal finetuning and baseline models. Valid and test are from CodeContests. Interview, introductory and competition are from APPS test set.

with open-sourced approaches such as CodeT5 (Wang et al., 2021), fine-tuned GPT-Neo (Hendrycks et al., 2021a), GPT-2 (Cheng et al., 2024), GPT-3 (Brown et al., 2020), one-shot StarCoder (Li et al., 2023b), WizardCoder (Luo et al., 2023b), CodeLlama series (Roziere et al., 2023), Deepseek-Coder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024). Additionally, we present results from SOTA closed-source model GPT4o (OpenAI, 2023b). As depicted in the results from the APPS shown in Tab. 2, we notice that the module-of-thought training approach leads to an enhancement in code generation capabilities compared with previous instruction finetuning models. Our MoTCoder exhibits improved performance across all difficulty levels and demonstrates more substantial gains in the interview and competition-level problems with more intricate solutions. To provide specifics, the *pass@1* performance of MoTCoder-6.7B surprisingly outperformed the closed-source model GPT-4 by an impressive margin of 12.61%. This outcome serves as compelling evidence of the efficacy of our approach in addressing competitive programming problems.

We also conduct a comparative analysis of our approach against previous LLM baselines with code-revision methods as well. Our included baselines contain Codex (Chen et al., 2021), CodeT5 (Wang et al., 2021), code-davinci, StarCoder (Li et al., 2023b), and WizardCoder (Luo et al., 2023b) and code-revision methods contain Self-edit (Zhang et al., 2023), CodeRL (Wang et al., 2021; Le et al., 2022), Self-repair (Olausson et al., 2023), and CodeChain (Le et al., 2023). The results presented in Tab. 2 illustrate that MoTCoder exhibits notable performance improvements. MoTCoder demonstrates superior performance across all categories compared to Qwen2.5-Coder-Instruct, achieving higher scores on the all of difficulty levels. MoTCoder-7B achieves an average score of 34.67%, surpassing Qwen2.5-Coder-Instruct-7B by 2.46%. MoTCoder-32B reaches an average score of 45.95%, surpassing the baseline model by 5.82%. This improvement highlights the effectiveness of MoTCoder's guided module-of-thought framework.

Results on CodeContests We conduct an evaluation of our approach on CodeContests (Li et al., 2022), benchmarking it against current coding models, including StarCoder (Li et al., 2023b), WizardCoder (Luo et al., 2023b), CodeLlama series models (Roziere et al., 2023), Deepseek-Coder (Guo et al., 2024), Qwen2.5-Coder (Hui et al., 2024). Furthermore, we present results from SOTA closed-source model GPT4o (OpenAI, 2023b). The results, as depicted in Tab. 3, reveal notable performance enhancements achieved by MoTCoder. Specifically, MoTCoder-7B achieves the performance of 19.09% compared to Qwen2.5-Coder (+3.89%), and MoTCoder-32B reaches 23.85% (+5.93%). These results demonstrate MoTCoder's superior capability in generating accurate code solutions.

Results on Self-Reflection To further explore our model's interactive and self-corrective capabilities, we constructed a multi-turn dialogue task. For cases in the CodeContests (Li et al., 2022) where the model did not succeed in passing all test cases, we prompted the model to self-reflect and regenerate the solutions. There is up to 5 rounds of reflection. The results in Tab. 4 demonstrate that our models achieved significant improvements in both code accuracy and self-correction capabilities. Specifically, the Qwen2.5-Coder-Instruct model with self-reflection showed an increase from 14.61% to 19.80% in the overall score. Moreover, the MoTCoder-7B model achieved comparable performance to Qwen2.5-Coder-Instruct without the need for reflection. With self-reflection, MoTCoder-7B improved from 19.09% to 23.08%, surpassing Qwen2.5-Coder-Instruct by 3.28%.

5 FURTHER ANALYSIS

Influence of modules on Accuracy. We conducted an analysis using the code generated by MoTCoder to evaluate the number of functions in the solution codes and their impact on accuracy across different difficulty levels. We focus on the relationship between problem complexity and optimal modular decomposition. The difficulty levels are categorized as introductory, interview, and competition, moving from easiest to hardest, as depicted in Fig. 3(a). The results reveal distinct trends across difficulty levels. For introductory problems, the accuracy generally declines as the number of functions increases, suggesting that simpler solutions benefit from minimal modularity. In contrast, the interview level maintains relatively stable accuracy across different function counts. Notably, for competition-level problems, there is an upward trend in accuracy with an increase in the number of functions.This observation indicates that while simpler problems are best addressed using fewer, singular modules, more complex problems benefit from being broken down into modular functions. This aligns with our intuition: for straightforward problems that can be solved with just a few lines of code, excessive modularization adds unnecessary complexity. Conversely, for challenging problems, decomposing them into submodules facilitates a more effective solution process, reflecting human strategies for tackling difficult issues.

Time and Memory Consumption. In Fig. 3(b), we analyze the average time and memory consumption for the generated MoT code and normal code, which are produced by the MoT finetuning and normal finetuning models, respectively. We collected data on samples from APPS and CodeContests where both MoT and normal code passed. It is evident that while the time consumption for the MoT model is comparable to that of the normal model, MoT finetuning consistently shows significantly lower memory consumption at all levels. This indicates that MoT finetuning is more memory efficient, as it is able to efficiently release unused memory by distinguishing between global variables and local variables that are only used within functions. Therefore, it is advantageous for scenarios with memory constraints.

Maintainability Analysis. In Fig. 3(c), we explore the maintainability metrics of the models. We compare the models after MoT finetuning and normal finetuning, as well as the baseline model Qwen2.5-Coder-7B-Instruct. We collected statistics on the generated passing code of these models on APPS and CodeContest, using the radon (Lacchia, 2014) tool to calculate their maintainability index. More details are in the Appendix. High maintainability index values generally indicate that the code is easier to maintain, usually due to lower complexity, fewer lines of code, and adequate documentation. The results show that for all levels, MoT code consistently demonstrates significantly higher maintainability compared to normal finetuning and the baseline. This suggests that MoT finetuning leads to code that is easier to understand and modify, which can be particularly beneficial in scenarios requiring long-term code maintenance and evolution.

6 CONCLUSION

This study highlights the limitations of Large Language Models (LLMs) in solving complex programming tasks due to their tendency to generate monolithic code blocks. In response, we developed Module-of-Thought Coder (MoTCoder), a framework that encourages the breakdown of tasks into manageable sub-tasks and sub-modules. Our results demonstrate that MoTCoder's approach significantly enhances the modularity and accuracy of solutions, as evidenced by considerable improvements in *pass@1* rates on both APPS and CodeContests benchmarks. Through the process of MoT instruction tuning, MoTCoder also achieved notable advancements in self-correction capabilities. Additionally, our analysis shows that MoTCoder improves the maintainability index of the generated code, thereby making it easier to comprehend and modify. We believe the introduction of MoT instruction tuning as a method to cultivate and leverage sub-modules paves the path for a promising direction for future research.

References

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. Palm 2 technical report. *CoRR*, abs/2305.10403, 2023. doi: 10.48550/arXiv.2305.10403. URL `https://doi.org/10.48550/arXiv.2305.10403`.

Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q. Tran, Dara Bahri, Jianmo Ni, Jai Prakash Gupta, Kai Hui, Sebastian Ruder, and Donald Metzler. Ext5: Towards extreme multi-task scaling for transfer learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL `https://openreview.net/forum?id=Vzh1BFUCiIX`.

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. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL `https://arxiv.org/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. Qwen technical report, 2023. URL `https://arxiv.org/abs/2309.16609`.

Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *URL https://doi. org/10.5281/zenodo*, 5297715, 2021.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model. *CoRR*, abs/2204.06745, 2022. doi: 10.48550/arXiv.2204.06745. URL `https://doi.org/10.48550/arXiv.2204.06745`.

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. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

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. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

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

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

Daixuan Cheng, Yuxian Gu, Shaohan Huang, Junyu Bi, Minlie Huang, and Furu Wei. Instruction pre-training: Language models are supervised multitask learners. 2024. URL https://arxiv.org/abs/2406.14491.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://vicuna.lmsys.org.

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. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *CoRR*, abs/2210.11416, 2022. doi: 10.48550/arXiv.2210.11416. URL https://doi.org/10.48550/arXiv.2210.11416.

DeepSeek-AI, :, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. Deepseek llm: Scaling open-source language models with longtermism, 2024a. URL https://arxiv.org/abs/2401.02954.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024b. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022. doi: 10.48550/arXiv.2204.05999. URL https://doi.org/10.48550/arXiv.2204.05999.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: program-aided language models. *CoRR*, abs/2211.10435, 2022.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

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. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL `https://arxiv.org/abs/2401.14196`.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. In *The Eleventh International Conference on Learning Representations*, 2023.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*, 2021a.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021b.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022. doi: 10.48550/arXiv.2203.15556. URL `https://doi.org/10.48550/arXiv.2203.15556`.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL `https://arxiv.org/abs/2409.12186`.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators, 2023.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models, 2023a.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 2023b.

Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. Unifiedqa: Crossing format boundaries with a single QA system. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pp. 1896–1907. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.171. URL `https://doi.org/10.18653/v1/2020.findings-emnlp.171`.

Michele Lacchia. Radon: A python tool that computes various metrics for python code, 2014. URL `https://radon.readthedocs.io/`. Version 5.1.0, accessed on [Access Date].

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

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

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In *NeurIPS*, 2022.

Liunian Harold Li, Jack Hessel, Youngjae Yu, Xiang Ren, Kai-Wei Chang, and Yejin Choi. Symbolic chain-of-thought distillation: Small models can also "think" step-by-step. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Toronto, Canada, July 2023a. Association for Computational Linguistics.

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

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. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022. doi: 10.48550/arXiv.2203.07814. URL `https://doi.org/10.48550/arXiv.2203.07814`.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*, 2023a.

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

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In *EMNLP*, pp. 1384–1403. Association for Computational Linguistics, 2022.

Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, mar 1971. ISSN 0001-0782. doi: 10.1145/362566.362568. URL `https://doi.org/10.1145/362566.362568`.

Microsoft. Azure openai service models. `https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/models`, 2023.

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

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023a. doi: 10.48550/arXiv.2303.08774. URL `https://doi.org/10.48550/arXiv.2303.08774`.

OpenAI. Gpt-4 technical report, 2023b.

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. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL `https://arxiv.org/abs/2412.15115`.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew J. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL `https://arxiv.org/abs/2112.11446`.

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

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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. 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, 2022. URL `https://openreview.net/forum?id=9Vrb9D0WI4`.

Melanie Sclar, Peter West, Sachin Kumar, Yulia Tsvetkov, and Yejin Choi. Referee: Reference-free sentence summarization with sharper controllability through symbolic knowledge distillation. *arXiv preprint arXiv:2210.13800*, 2022.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.

Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Xavier Garcia, Dara Bahri, Tal Schuster, Huaixiu Steven Zheng, Neil Houlsby, and Donald Metzler. Unifying language learning paradigms. *CoRR*, abs/2205.05131, 2022. doi: 10.48550/arXiv.2205.05131. URL `https://doi.org/10.48550/arXiv.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. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/arXiv.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

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

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP (1)*, pp. 8696–8708. Association for Computational Linguistics, 2021.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922, 2023a. doi: 10.48550/arXiv.2305.07922. URL https://doi.org/10.48550/arXiv.2305.07922.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023b.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 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, 2022a. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022b.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. Symbolic knowledge distillation: from general language models to commonsense models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4602–4625, Seattle, United States, July 2022. Association for Computational Linguistics.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *NeurIPS*, 2022.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

Hanwei Xu, Yujun Chen, Yulun Du, Nan Shao, Yanggang Wang, Haiyu Li, and Zhilin Yang. Zeroprompt: Scaling prompt-based pretraining to 1, 000 tasks improves zero-shot generalization. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pp. 4235–4252. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.findings-emnlp.312.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024. URL https://arxiv.org/abs/2407.10671.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*. OpenReview.net, 2023.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STar: Bootstrapping reasoning with reasoning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022.

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. GLM-130B: an open bilingual pre-trained model. *CoRR*, abs/2210.02414, 2022. doi: 10.48550/arXiv.2210.02414. URL `https://doi.org/10.48550/arXiv.2210.02414`.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*, 2023.

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. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022. doi: 10.48550/arXiv.2205.01068. URL `https://doi.org/10.48550/arXiv.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. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568, 2023. doi: 10.48550/arXiv.2303.17568. URL `https://doi.org/10.48550/arXiv.2303.17568`.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. *CoRR*, abs/2205.10625, 2022.

A Appendix

You may include other additional sections here.