SRA-MCTS: Self-driven Reasoning Augmentation with Monte Carlo Tree Search for Code Generation

Bin Xu[*], **Yiguan Lin**[*], **Yinghao Li**, **Yang Gao**[†]
School of Computer Science and Technology,
Beijing Institute of Technology, Beijing, China
{binxu,yglin,yhli,gyang}@bit.edu.cn

Abstract

Large language models exhibit remarkable performance in simple code generation tasks. However, they encounter significant challenges when addressing complex problems that require reasoning and question decomposition. To tackle this, we propose a self-driven reasoning augmentation process, SRA-MCTS, which incorporates Monte Carlo Tree Search (MCTS) for reasoning data generation. SRA-MCTS enables LLMs to self-generate intermediate reasoning steps and perform iterative self-evaluation, facilitating self-improvement. Specifically, it utilizes MCTS to produce diverse intermediate reasoning steps. During each iteration, MCTS generates a step and employs self-evaluation to guide the selection of subsequent branches, ultimately forming a sufficiently diverse reasoning path referred to as "thinking". This thinking guides the model in generating corresponding code, and both are combined as training data for supervised fine-tuning. Experimental results demonstrate that SRA-MCTS achieves consistent performance improvements across three model scales without additional supervisory assistance. Applied to the Meta-Llama-3.1-8B-Instruct model, it delivers an 11-point improvement on the MBPP-Complex dataset, underscoring the significant potential for model self-improvement. The code and data are available at https://github.com/DIRECT-BIT/SRA-MCTS.

1 Introduction

Large language models (LLMs) excel at generating code for simple questions, but their performance on more complex tasks remains unsatisfactory. [Luo *et al.*, 2024; Li *et al.*, 2023]. Recent studies have predominantly concentrated on enhancing the quality of training datasets to improve model performance. These improvements typically address two fundamental aspects of the training data.

On one hand, improving the quality of the questions enhances the model's understanding of the input. Some methods optimize the question portion of the dataset by leveraging more powerful models or summarizing existing open-source code, thereby enhancing both the depth and breadth of the questions [Ouyang *et al.*, 2022; Luo *et al.*, 2024; Wei *et al.*, 2023]. On the other hand, increasing the complexity of the answers helps guide the model to generate higher-quality codes. BRAINSTORM [Li *et al.*, 2023] introduces a brainstorming approach to generate and select diverse ideas related to a given question, thereby enhancing the model's reasoning capabilities. DolphCoder [Wang *et al.*, 2024b] utilizes GPT-3.5 with multiple prompt templates to obtain diverse outputs for multi-objective instruction fine-tuning. These methods all aim to leverage additional natural language text to guide LLMs in generating high-quality results.

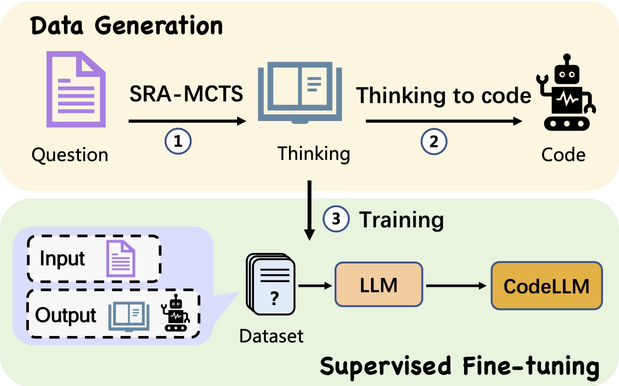

Figure 1: The overall workflow of our method, with data generation shown at the top and training at the bottom. SRA-MCTS guides the LLM to generate thinking, which is then used by the LLM as a part of the prompt to generate the corresponding code. The question, thinking, and code are organized as training data for supervised fine-tuning.

Furthermore, ScaleAI [Wang *et al.*, 2024a] has demonstrated that providing LLMs with correct solutions in natural language form can significantly enhance model performance, even when these solutions consist of incomplete plans with only a few dozen tokens. This highlights the potential of natural language solutions to guide and inspire LLMs to think in the right direction. Similar to ScaleAI, there have been some works exploring how to improve the intermediate reasoning steps to improve model performance. [Chen *et al.*, 2024; Li *et al.*, 2024b; Long *et al.*, 2024]. However, these studies lack explanations for the natural language solutions and lack an inherent mechanism for error correction, limiting the model's ability to effectively utilize natural language reasoning steps. Therefore, we focus on the intermediate natural language reasoning steps generated by the model, providing it with a reliable and diverse reasoning direction.

In this paper, we refer to the intermediate natural language reasoning steps generated by the model for code generation as "thinking". We propose a Self-driven Reasoning Augmentation method, leveraging Monte Carlo Tree Search (MCTS) [Coulom, 2006] to generate high-quality thinking, which we call SRA-MCTS as shown in Figure 1. Different from ReST-MCTS* [Zhang *et al.*, 2024a], which uses MCTS to generate complete answers, we apply MCTS to the more critical thinking portion. This provides the model with reliable reasoning, enhancing its ability to generate high-quality code. SRA-MCTS iteratively constructs the final reasoning path for a given question. Specifically, each iteration generates multiple potential next steps based on the current reasoning path. During each step selection, the model reflects on and scores the newly generated step, updating the scores of previously generated steps. This process allows the model to re-evaluate and reconsider unselected steps in subsequent iterations. This iterative process enables the model to engage in self-reflection, correcting potentially flawed reasoning steps. Finally, the sequence of reasoning steps generated over multiple iterations is combined into comprehensive thinking that guides the model in generating the final code. The data generated through SRA-MCTS will be used to fine-tune the model, ultimately resulting in a higher-performing code generation model. Our contributions are as follows:

- We propose a plug-and-play reasoning augmentation data generation method, SRA-MCTS. This method is simple and effective, as it enables the LLM to self-generate and self-evaluate data during the process. Then the generated data is used for model training, fostering the model's self-improvement.
- We propose a pipeline for the code generation domain, where input questions are processed by SRA-MCTS to generate diverse, high-quality thinking without additional supervision, thereby creating a positive feedback loop for continuous improvement.
- We perform a detailed analysis of the generated data, examining the impact of its quality on model performance. Additionally, by comparing the model's performance on the original benchmark and the derived, more complex benchmarks, we validate the effectiveness of the self-reasoning augmentation approach.

2 Related Work

Data Augmentation Evol-Instruct [Luo *et al.*, 2024] enhances CodeAlpaca [Chaudhary, 2023] by integrating heuristic rules, which increases the complexity and diversity of the seed instructions. OSS-Instruct [Wei *et al.*, 2023] leverages open-source code snippets to generate questions that better represent real-world distributions. CodeOcean [Yu *et al.*, 2023] utilizes GPT-3.5 and GPT-4 for question distillation and filtering, resulting in high-quality, diverse data. DolphCoder [Wang *et al.*, 2024b] improves LLM learning by generating step-by-step answers through multiple turns and pseudo code for problem-solving.

Self-generated Data Data generated by LLMs can effectively mitigate biases in understanding arising from variations in language style. ReCo [Li *et al.*, 2024a] utilizes LLMs to rewrite code from codebases, thereby reducing retrieval accuracy degradation caused by stylistic deviations during the retrieval process. ReST-MCTS* [Zhang *et al.*, 2024a] introduces a reinforced self-training approach that combines process reward guidance with MCTS to collect higher-quality reasoning trajectories and stepwise values, which are subsequently used to train the strategy and reward models.

Exploration and Thinking Methods Similar to how humans think deeply before answering difficult questions, OpenAI o1 [OpenAI,] employs a series of thought processes to solve problems. Through reinforcement learning, it refines its strategies, corrects errors, and streamlines complex steps. AFLOW [Zhang *et al.*, 2024b] employs MCTS to refine workflows through iterative code modifications and feedback. StepCoder [Dou *et al.*, 2024] utilizes parts of a standardized solution as a prompt, enabling LLMs to explore simple sequences and improve through feedback-based reinforcement learning.

Unlike previous works, our SRA-MCTS not only enhances the thinking process within the data, improving the model's reasoning and thinking capabilities, but also establishes a fully autonomous pipeline that includes self-generation, self-evaluation, and ultimately self-improvement.

3 Method

We propose a pipeline that leverages the model itself to enhance training data and augment reasoning ability, as shown in Figure 1, consisting of three stages: SRA-MCTS, Thinking to code, and Training. First, we use the LLM with SRA-MCTS to generate step-by-step natural language thinking for each question. Then, the LLM generates specific codes based on the question and the generated thinking. Finally, we combine the question, the thinking, and the code to create a fine-tuning dataset for training the LLM.

3.1 SRA-MCTS

We develop a self-driven reasoning augmentation method with MCTS, called SRA-MCTS, to generate natural language thinking for questions. It consists of four phases: **Selection**, **Expansion**, **Evaluation & Reflection**, and **Backpropagation**, as shown in Figure 2. These phases are carried out on a search tree composed of tree nodes and will be repeated multiple times, with each iteration generating a specific reasoning step.

A tree node has four attributes: 1) *State* represents the current state, including the question and the steps generated so far; 2) *Action* represents the next step generated based on the current state; 3) *Reward* represents the score generated by LLM based on state and action; 4) *Reflection* guides the generation of the next iteration. The initial search tree consists

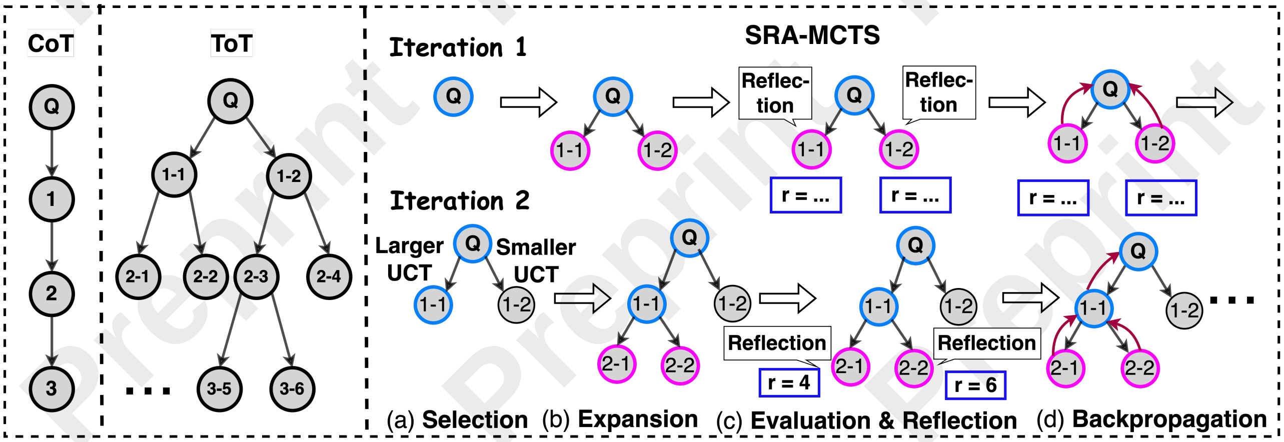

Figure 2: Self-driven reasoning augmentation process of SRA-MCTS. (a) Selection: A leaf node is selected to be expanded in the next phase. (b) Expansion: A single step is generated and assigned to the node. (c) Evaluation & Reflection: The step in the node is scored and an insight is generated as reflection. (d) Backpropagation: Reward scores are propagated back. In the notation "1-1" within a node, the first "1" indicates that it is the 1st step in the thinking, and the second "1" denotes the 1st variant for this step. The same logic applies to other nodes. The red nodes at the bottom represent the newly generated nodes, and all the ancestor nodes in the path from them to the root node, marked in Blue, are the previously selected nodes.

of only a root node, with its *state* attribute set to the input question, and the other three attributes are empty.

Selection

The purpose of the selection is to select one step from the model's currently generated steps for the next step generation. The selection phase involves selecting a leaf node N_d from the search tree, where d represents the depth of the leaf node. This node will be expanded in the expansion phase.

We use UCT [Kocsis and Szepesvári, 2006], which is calculated based on the node's *reward* value, to measure the value of each node being selected. The formula is as follows:

$$UCT = r + c \cdot \sqrt{\frac{\ln N}{n}} \quad (1)$$

where r is the *reward* value of the current node that is generated by the Evaluation & Reflection phase in the last iteration, c is the exploration constant set to 0.5, N denotes the number of times the parent node has been selected, and n represents the number of times the current node has been selected. UCT is designed to balance exploitation and exploration, represented by the terms on the left and right of the formula respectively. If the exploration term is removed, it degenerates into a greedy selection, which harms diversity.

With the selection phase starting from the root node, the node with the highest UCT value at the current depth is selected. Then, the process continues by choosing the child node with the highest UCT value among all the child nodes of the selected node. This process continues until a leaf node of the search tree is selected. Note that during the first iteration, only the root node which is also a leaf node exists, so the root node's UCT value does not need to be calculated.

Expansion

The purpose of the expansion is to generate the next step based on the existing steps. The expansion phase involves generating several new child nodes based on the selected node N_d, and assigning values to the *state* and *action* attributes of the new nodes.

The *state* attribute is designed to store the input question and steps that the LLM has generated up to the current node. Therefore, for child nodes that originate from the same parent node, their *state* values are identical and referred to S_{d+1}, as they share the input question and the same history of step. Specifically, S_{d+1} is formed by concatenating the *state* and *action* attributes of the node N_d:

$$S_{d+1} = concatenate(S_d, A_d) \quad (2)$$

where S_d and A_d represent the *state* and *action* of the selected node N_d, respectively.

The *action* attribute is designed to store the next step generated based on the current *state* and the *reflection* of the parent node. Since different methods can be used to generate different next steps, the *action* attribute can have multiple distinct values, leading to the creation of different child nodes. Each node represents a possible path that can be made from the current state, allowing for the exploration of various possibilities in the search tree.

To generate diversity in the steps, the expansion phase adopts the sample decoding strategy, where each next step is generated one by one. Each time a new step is generated, it represents a new node created from the selected node N_d. To prevent the LLM from generating duplicate next steps, we add the next steps that have already been generated to the input. This process can be represented by the following formula:

$$A^i_{d+1} = \begin{cases} LLM(S_{d+1}; R_d), & i = 1 \\ LLM(S_{d+1}; A^1_{d+1} \\ , ..., A^{i-1}_{d+1}; R_d), & i > 1 \end{cases} \quad (3)$$

where A^i_{d+1} represents the *action* value of the i-th child node at depth d+1, R_d represents the *reflection* attribute value of the selected node N_d, which is generated in the last iteration. Adding the already generated steps to the input helps the LLM explore different paths in the search space.

To manage computational costs, we set the number of expanded nodes to 3. If a duplicate step is detected, the generation process is repeated, with a maximum of 5 retries At this stage, code generation is not involved, only the natural language reasoning process is generated.

Evaluation & Reflection

The third phase consists of two parts: Evaluation and Reflection, which support the selection and expansion phases, respectively.

Evaluation The evaluation part assigns the *reward* attribute values to the new nodes generated in the expansion phase. These reward values will be then used in the selection phase to calculate the UCT values.

We use the LLM itself, which generates the steps during the expansion phase, as the evaluator to score the new nodes and explore their potential for self-improvement through self-evaluation. [1] The *reward* value of the i-th new node N^i_{d+1} is as follows:

$$r^i_{d+1} = LLM(S^i_{d+1}, A^i_{d+1}) \tag{4}$$

We use a progressive scoring method, making judgments sequentially from four aspects, the process is shown in Figure 3. The order of judgment is as follows: Single-step correctness, Solution coherence, Solution completeness, and Solution correctness. If the criteria for the current aspect are met, the corresponding score is given directly, and the other aspects are ignored.

Reflection The reflection part aims to guide the next generation direction for the current node, and assigns the *reflection* attribute values to the new nodes generated in the expansion phase. The *reflection* attribute is a brief thought generated by the LLM and will be used in the next iteration's expansion phase.

We use the reflection [Zhang *et al.*, 2024a] mechanism to allow the model to think before generating the next action, thus preventing performance degradation. The $reflection$ value of the i-th new node N^i_{d+1} is as follows:

$$R^i_{d+1} = LLM(S^i_{d+1}, A^i_{d+1}) \tag{5}$$

Additionally, the *reflection* value is used to assess whether all the generated steps have solved the question. If the generated steps from *state* and *action* at this node are deemed sufficient to solve the question, the *reflection* value will be an $< end >$ tag to indicate that the SRA-MCTS process has concluded. Otherwise, the LLM will generate a reflection for the next iteration's reasoning. The *reflection* value is shown as follows:

$$R = \begin{cases} < end >, & \text{question solved} \\ \text{short thought for next step}, & \text{question unsolved} \end{cases} \tag{6}$$

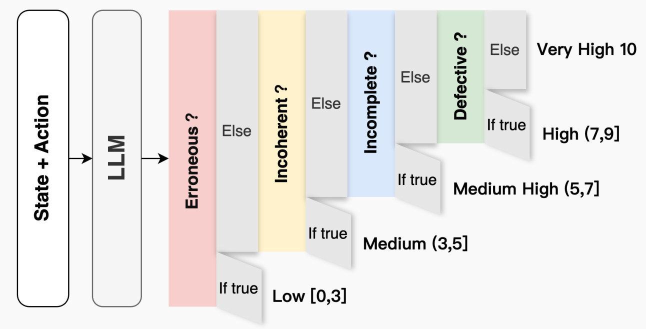

Figure 3: The progressive scoring method. The *state* and *action* of the node are used as inputs, and the judgment is made sequentially from left to right based on the four principles. If the current principle is satisfied, an integer score in the corresponding interval is output; otherwise, the next principle is evaluated. If all the principles are not met, the model will give the current input a full score of 10.

Backpropagation

Backpropagation aims to update the *reward* values of all parent nodes of the current node, making the reward values of the parent nodes more accurate, thereby affecting the generation of the thinking. For the new nodes generated during the expansion phase $\{N^1_{d+1}, N^2_{d+1}, ..., N^I_{d+1}\}$, they collectively update their parent node, which is the node selected in the selection phase N_d. Afterward, the parent node N_d continues to update its parent node N_{d-1}, and this process is repeated until the root node is updated. The update process of the parent node's *reward* value r_p is as follows:

$$\begin{aligned} r_\triangle &= \sum_{i=1}^{I}(v^i_c \ \cdot \ r^i_c) / \sum_{i=1}^{I} v^i_c \\ r_p &= \alpha \cdot r_p + (1-\alpha) \cdot r_\triangle \end{aligned} \tag{7}$$

where v^i_c represents the number of times the i-th child node is selected, r^i_c represents the *reward* score of the i-th child node, $r_\triangle$ represents the weighted increment of the parent node's *reward* value, and α is a hyperparameter to balance the original *reward* of the parent node and the increment from the child nodes.

The updated *reward* value of the parent node will influence the selection of child nodes in subsequent iterations. Specifically, in the next selection phase, SRA-MCTS will use the updated *reward* values to evaluate the relative merits of different nodes, thereby deciding which branch to explore. This dynamic reward adjustment helps gradually refine the strategy across the entire tree, making the search more efficient and progressively converging toward the optimal solution.

SRA-MCTS repeats the four phases above until the $<$ $end >$ tag is output, or the iteration limit is reached.

3.2 Thinking to Code

SRA-MCTS generates multiple steps based on the question, and these steps together form a thinking. In this stage, we have the LLM act as a function implementer, strictly converting the thinking into code.

[1] Note: The use of the model itself here is aimed at exploring the potential for end-to-end self-improvement. Larger open-source models and closed-source model APIs, which may have more accurate evaluation capabilities, can easily replace this component.

3.3 Training

After the first two stages, we collect the question, thinking, and code triples, with the question as the input and the latter two as the output, forming the training dataset. This dataset is then used for supervised fine-tuning.

4 Experiment

4.1 Training Data Generation

Question To validate whether SRA-MCTS can indeed generate higher-quality thinking, we need to collect questions with a certain degree of difficulty. Therefore, we select medium and hard-level questions from the LeetCode dataset [Greengerong, 2023], excluding easy-level questions. We retain only the question descriptions and discard the answers, allowing different methods to generate both thinking and code. To prevent any overlap between the training and test sets, we perform decontamination on the training data. Specifically, we conduct 10-gram level duplicate detection on each question in the dataset, setting the threshold at 0.3. If the similarity between a training question and any test set question exceeds this threshold, we remove the corresponding training data. As a result, we obtain a final training set of 1,819 unique questions.

Thinking and Code For thinking and code generation, we use SRA-MCTS to generate the corresponding thinking for the questions in the training set. Each step in the generation process is performed in a zero-shot manner. We prompt the model to generate each step according to a specified format, ensuring that the steps can be easily extracted. If the thinking contains code, we use a regular expression to remove it, leaving only the natural language reasoning content. This ensures that the reasoning process is kept separate from the code, enabling a clearer focus on the thought process behind the code generation. After generating the final thinking, we concatenate the question with its corresponding thinking and prompt the model to generate the final code.

4.2 Baseline

We choose gemma-2-2b-it [Team *et al.*, 2024], Meta-Llama-3.1-8B-Instruct [Dubey *et al.*, 2024], and Qwen2.5-14B-Instruct [Team, 2024] as our backbone models. The rationale behind this selection is their strong instruction-following capabilities and the fact that they have been pre-trained on extensive code data. These models demonstrate significant potential to activate proactive thinking for solving programming tasks through reasoning augmentation techniques. We directly use the backbone models to perform reasoning on the test set, which serves as the baseline performance score for comparison.

We apply the Chain-of-Thought (CoT) [Wei *et al.*, 2022] and Tree-of-Thought (ToT) [Yao *et al.*, 2023] reasoning methods to each model as baseline methods. For CoT, we use the conventional prompt, "Let's think step by step", to generate the thinking. For ToT, we use depth-first search to generate the thinking, setting the maximum depth to 4 and the branching factor to 3, which aligns with the number of nodes expanded during the expansion phase of SRA-MCTS. Similar to SRA-MCTS, the thinking generated by these methods is processed using regular expressions to remove any code content. Finally, the generated thinking is concatenated with the question and fed into the model to produce the final code.

4.3 Test Set

Benchmark We use commonly adopted benchmarks in the code generation field, including Human-Eval [Chen *et al.*, 2021] and MBPP [Austin *et al.*, 2021]. Additionally, we utilize Human-Eval+ and MBPP+ within the EvalPlus [Liu *et al.*, 2023] framework, where the test cases are several times larger than the original versions. As for the task type, Human-Eval involves completing a given function, while MBPP requires the model to generate the function from scratch.

Test Set Split To evaluate the model's reasoning ability across different difficulty levels, we categorize the test set questions into different difficulty levels. Specifically, we use GPT-4o [Hurst *et al.*, 2024] to classify the questions into easy, medium, and hard categories. All questions across the three difficulty levels form the Full split and the medium and hard questions are grouped into the Complex split.

Evaluation Metric In code generation field, pass@k is a widely used metric that measures the probability of generating a correct solution in at least one of k attempts [Chen *et al.*, 2021; Kulal *et al.*, 2019]. It reflects the model's success rate across multiple attempts, which is particularly valuable given the inherent uncertainty in the generation process.

4.4 Training Setup

In SRA-MCTS, the total iteration limit is set to 5, the generation temperature in the expansion phase is set to 0.9, with a top-p sampling value of 0.98, and the α value in the backpropagation phase is manually set to 0.5.

We use the models mentioned in the baseline as the foundation for training, setting the learning rate to 1e-4, training for 2 epochs, with a warmup ratio of 0.05 and a batch size of 4. For each model scale, we train using three different datasets: 1) Self-generated CoT data; 2) Self-generated ToT data; 3) SRA-MCTS generated data. We use Llama-factory [Zheng *et al.*, 2024] for LoRA [Hu *et al.*, 2022] fine-tuning, with the models trained on two RTX A6000 GPUs.

4.5 Main Results

SRA-MCTS demonstrates the best overall performance and excels in solving complex questions. As shown in Table 1, our method achieves the highest average performance improvement across all models, with a significantly larger gain on the Complex split of most benchmarks compared to the Full split, particularly with an increase of approximately 2-15 points on MBPP and MBPP+. However, for the Human-Eval-related benchmarks, all methods experience a decline in several metrics. While our method still shows improvement in some metrics, it is not obvious compared to the other baseline methods. We attribute this phenomenon to the differences between the task types in the training and test sets. As discussed in Section 4.3, tasks in the MBPP benchmark, like those in the training set, require the model to generate code from scratch based on a given question, whereas Human-Eval tasks ask the model to complete a function header provided

Method	**MBPP**		**MBPP+**			**Human-Eval**		**Human-Eval+**			**Average**
	pass@1		pass@1		pass@10	pass@1		pass@1		pass@10	
	Full	Complex	Full	Complex	Full	Full	Complex	Full	Complex	Full	**Increment**
					gemma-2-2b						
Instruct	34.42	11.32	43.39	13.64	48.41	39.76	25.42	33.23	18.64	51.22	-
CoT	34.90 (+0.48)	11.32 (+0.00)	43.70 (+0.31)	13.64 (+0.00)	47.90 (-0.51)	41.89 (+2.13)	25.42 (+0.00)	34.94 (+1.71)	20.34 (+1.70)	53.05 (+1.83)	+0.77
ToT	33.86 (-0.56)	9.43 (-1.89)	44.71 (+1.32)	9.09 (-4.55)	47.62 (-0.79)	40.18 (+0.42)	23.73 (-1.69)	32.68 (-0.55)	20.34 (+1.70)	45.12 (-6.10)	-1.27
SRA-MCTS	33.92 (-0.50)	**16.98 (+5.66)**	45.37 (+1.98)	**15.90 (+2.26)**	49.21 (+0.80)	40.73 (+0.97)	**25.42 (+0.00)**	34.88 (+1.65)	**20.34 (+1.70)**	49.39 (-1.83)	**+1.27**
					Meta-Llama-3.1-8B						
Instruct	51.94	33.96	45.37	29.55	74.60	62.74	**47.46**	58.90	**40.68**	67.68	-
CoT	52.94 (+1.00)	39.62 (+5.66)	60.50 (+15.13)	**40.91 (+11.36)**	74.60 (+0.00)	62.32 (-0.42)	28.81 (-18.65)	58.35 (-0.55)	22.03 (-18.65)	66.46 (-1.22)	-0.63
ToT	52.72 (+0.78)	32.08 (-1.88)	60.24 (+14.87)	31.82 (+2.27)	74.07 (-0.53)	62.26 (-0.48)	40.68 (-6.78)	57.44 (-1.46)	32.20 (-8.48)	63.41 (-4.27)	-0.60
SRA-MCTS	54.52 (+2.58)	**45.28 (+11.32)**	59.97 (+14.60)	38.64 (+9.09)	75.66 (+1.06)	62.19 (-0.55)	44.07 (-3.39)	57.87 (-1.03)	38.98 (-1.70)	68.29 (+0.61)	**+3.26**
					Qwen2.5-14B						
Instruct	56.42	56.60	61.48	52.27	70.37	80.37	69.49	76.52	61.02	90.95	-
CoT	58.12 (+1.70)	45.28 (-11.32)	63.97 (+2.49)	31.82 (-20.45)	70.37 (+0.00)	78.66 (-1.71)	61.02 (-8.47)	73.84 (-2.68)	54.24 (-6.78)	90.24 (-0.71)	-4.79
ToT	59.98 (+3.56)	54.72 (-1.88)	62.67 (+1.19)	45.45 (-6.82)	70.63 (+0.26)	75.61 (-4.76)	64.41 (-5.08)	73.84 (-2.68)	54.24 (-6.78)	90.24 (-0.71)	-2.37
SRA-MCTS	64.20 (+7.78)	**71.70 (+15.10)**	61.16 (-0.32)	**63.64 (+11.37)**	83.60 (+13.23)	85.37 (+5.00)	**69.49 (+0.00)**	75.00 (-1.52)	**61.02 (+0.00)**	91.46 (+0.51)	**+5.12**

Table 1: Main results of different methods. "Instruct" represents the method of directly using the backbone model for inference. "CoT", "ToT", and "SRA-MCTS" represent the methods of using the data generated by the corresponding methods for training. "Full" represents the entire dataset, and "Complex" represents the dataset consisting of the medium and hard categories classified by GPT-4o.

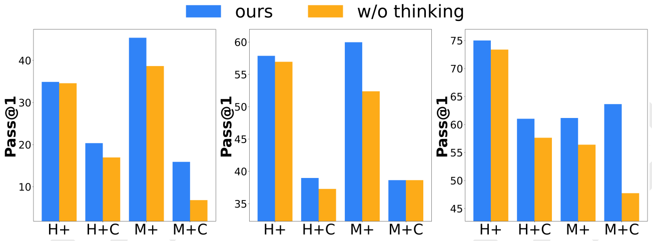

Figure 4: Comparison of the impact of "thinking" in training data across the three aforementioned models. "w/o thinking" indicates models trained without thinking processes. For clarity, H and M represent Human-Eval and MBPP, respectively; C denotes the Complex split. Left: our method; Right: control group without thinking.

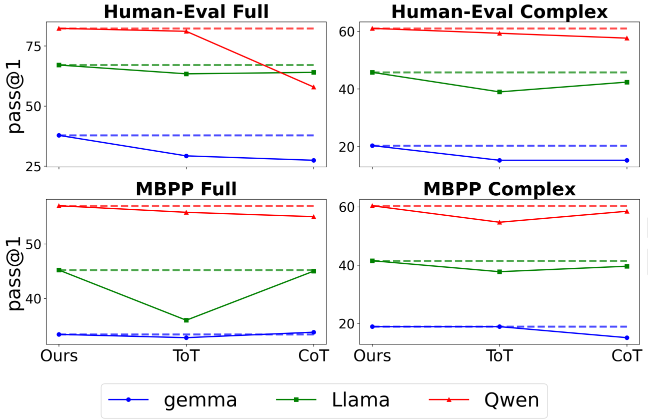

Figure 5: Comparison results of different thinking variants. The dashed line represents the performance of SRA-MCTS.

in the prompt. Due to the consistency in task style between MBPP and the training set, our method achieves more substantial improvements compared to other methods. This highlights the critical role that the style of training data plays in influencing downstream task performance.

The larger the model, the more pronounced the improvement from our method. As we observe in Table 1, the average performance improvement increases with the model size, and this trend is even more evident on the Complex split. We attribute this to the fact that larger models possess better generation and evaluation capabilities. With stronger generation and evaluation capabilities enabled by larger model scales, these models can produce higher-quality data and more effectively identify errors and redundancies in reasoning steps. SRA-MCTS effectively leverages these enhanced capabilities, leading to significant performance gains.

5 Analysis

5.1 Ablation Study of the Thinking

Effect of Existence We additionally train a model on a dataset without thinking, containing only the question and code, and compare it with the dataset that includes the question, thinking, and code triples. For a fair comparison, the code in the dataset without thinking is directly extracted from the training set generated by SRA-MCTS. As shown in Figure 4, nearly all cases without thinking perform worse than those with thinking, with this gap being more obvious on the Complex split. We believe this is due to simpler questions not requiring much reasoning from the model to provide the correct solution, allowing a code-only training set to solve these questions effectively. It is only when the question difficulty increases that the model benefits from higher-quality thinking to guide it toward generating the correct code.

Effect of Variants We extract the thinkings generated by CoT, ToT, and SRA-MCTS from their respective training sets and concatenate them with the question and ground truth code to create three new training sets, each differing only in the thinking content. We then retrain the backbone models on these new datasets, and the results are shown in Figure 5. Our method generates the most stable thinking among the three approaches. This is demonstrated by the fact that even when the code is replaced, our method maintains strong performance, whereas the other methods experience noticeable performance degradation. This indicates that the thinking

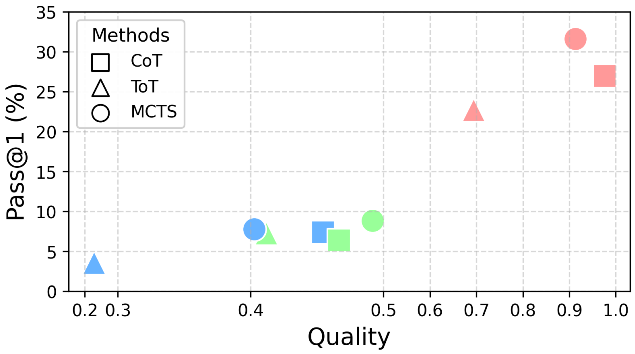

Figure 6: The relationship between the quality of thinking and the pass@1 rate of training set code. Red represents the Qwen model, green represents the Llama model, and blue represents the gemma model.

Model	Size	MBPP+	Human-Eval+	Average
CodeGen	2B	36.00	22.60	29.30
Codegemma	2B	**46.60**	20.70	33.65
SRA-MCTS (Ours)	2B	45.37	**34.88**	**40.13**
DolphCoder	7B	52.60	54.90	53.75
CodeLlama	7B	45.40	34.10	39.75
WizardCoder	7B	49.50	45.10	47.30
Magicoder-S-CL	7B	**60.10**	**67.70**	**63.90**
SRA-MCTS (Ours)	8B	59.97	57.87	58.92
DolphCoder	13B	54.10	57.90	56.00
CodeLlama	13B	50.90	36.60	43.75
WizardCoder	13B	54.20	50.60	52.40
StarCoder2-Instruct	15B	**61.20**	63.40	62.30
SRA-MCTS (Ours)	14B	61.16	**75.00**	**68.08**

Table 2: Comparison of the pass@1 results of our method and other competitive methods. The results of other methods are derived from the corresponding papers or the evaluation results of the EvalPlus framework.

generated by our method not only contributes to the final performance improvement but also provides robustness, leading to a more stable model.

Effect of Quality We use GPT-4o to evaluate the quality of the thinking, categorizing each instance into low, medium, and high quality. The ratios of medium and high-quality thinking in each training set are used to represent the quality score of the dataset. As shown in Figure 6, there is generally a linear relationship between the quality of the thinking in the training set and the model's performance, meaning that higher-quality thinking leads to better final performance. However, for our method, there are cases in the gemma and Qwen models where the thinking quality is relatively low, yet the performance is higher than expected. We hypothesize that, during the evaluation of thinking, the evaluator may overemphasize unnecessary overthinking steps, complicating the model's ability to generate code based on overly complex reasoning processes. This not only suggests that the evaluation of larger models may not always be reliable, but also highlights the superiority of our method.

Model	MBPP	MBPP+	Human-Eval	Human-Eval+
		gemma-2-2b		
Distillation	34.76	41.93	38.41	33.78
SRA-MCTS	**34.88** (+0.12)	**45.37** (+3.44)	**40.73** (+2.32)	**33.92** (+0.14)
		Meta-Llama-3.1-8B		
Distillation	56.03	58.09	59.45	51.10
SRA-MCTS	**57.87** (+1.84)	**59.97** (+1.88)	**62.19** (+2.74)	**54.52** (+3.42)
		Qwen2.5-14B		
Distillation	**71.20**	58.65	83.41	73.35
SRA-MCTS	64.20 (-7.00)	**61.16** (+2.51)	**85.37** (+1.96)	**75.00** (+1.65)

Table 3: The table presents the pass@1 results of the same model when trained on datasets generated by itself versus those distilled by an external model. In this comparison, the external model used is Meta-Llama-3-70B-Instruct.

5.2 Further analysis

Comparison with Competitive Methods. We compared our approach with other competitive methods using models of similar sizes, as shown in Table 2. For 2B and 13B models, our method not only excels in individual metrics but also achieves the best overall average performance. The only exception is the 8B Magicoder-S-CL method, where our approach falls slightly behind. We attribute this to the fact that our fine-tuning dataset is significantly smaller and less diverse compared to Magicoder. The Magicoder dataset is approximately 41 times larger than ours, and its larger scale and potential diversity likely contribute to its superior performance. This comparison underscores the efficiency of our approach. Despite using a small dataset, we are able to generate results that are of sufficiently high quality and diversity. This highlights the effectiveness of our method, which can achieve competitive results even with much more limited data.

Effectiveness of Self-generation. We use Meta-Llama-3-70B [Dubey *et al.*, 2024] as an external model to generate a training dataset, which is then used to train smaller models. We compare the performance of models trained with self-generated data versus those trained with data distilled from the larger model. The results in Table 3 demonstrate that our self-generation approach outperforms the data distillation method across all model sizes and nearly all benchmarks. Notably, SRA-MCTS shows substantial improvements on the Human-Eval and Human-Eval+ datasets. In contrast, the distillation method only outperforms on the MBPP benchmark when using a 14B model. This result strongly supports the feasibility of using small models to self-generate data for training, demonstrating that self-generated data can achieve competitive performance without relying on large-scale data distillation.

6 Conclusion

We propose a self-improvement pipeline where SRA-MCTS performs both self-generation and self-evaluation for given questions. The high-quality thinking text generated through this process serves to guide code generation, with both the thinking and the code subsequently used for fine-tuning. Experimental results demonstrate that SRA-MCTS outperforms both CoT and ToT on more complex tasks, achieving an 11-point improvement on the MBPP Complex split.

Acknowledgements

This work was supported by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92370110).

Contribution Statement

- Bin Xu: Developed the idea; conceived and designed the analysis; performed the analysis; wrote the paper.
- Yiguan Lin (Equal Contribution): Conceived and designed the analysis; performed the analysis; wrote the paper.
- Yinghao Li: Conceived and designed the analysis; performed the analysis; wrote the paper
- Yang Gao (Corresponding Author): Provided computing resources and funding.

References

[Austin *et al.*, 2021] 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.

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

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, 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.

[Chen *et al.*, 2024] Boyuan Chen, Zhuo Xu, Sean Kirmani, Brian Ichter, Dorsa Sadigh, Leonidas J. Guibas, and Fei Xia. Spatialvlm: Endowing vision-language models with spatial reasoning capabilities. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pages 14455–14465. IEEE, 2024.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[Dou *et al.*, 2024] Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4571–4585. Association for Computational Linguistics, 2024.

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

[Greengerong, 2023] Greengerong. Leetcode dataset, 2023.

[Hu *et al.*, 2022] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[Hurst *et al.*, 2024] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[Kulal *et al.*, 2019] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11883–11894, 2019.

[Li *et al.*, 2023] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. Think outside the code: Brainstorming boosts large language models in code generation. *CoRR*, abs/2305.10679, 2023.

[Li *et al.*, 2024a] Haochen Li, Xin Zhou, and Zhiqi Shen. Rewriting the code: A simple method for large language model augmented code search. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024,* *Bangkok, Thailand, August 11-16, 2024*, pages 1371–1389. Association for Computational Linguistics, 2024.

[Li *et al.*, 2024b] Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *CoRR*, abs/2409.09584, 2024.

[Liu *et al.*, 2023] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *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*, 2023.

[Long *et al.*, 2024] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. On llms-driven synthetic data generation, curation, and evaluation: A survey. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 11065–11082. Association for Computational Linguistics, 2024.

[Luo *et al.*, 2024] 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. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[OpenAI,] OpenAI. Learning to reason with large language models. https://openai.com/index/learning-to-reason-with-llms/. Accessed: 2024-10-31.

[Ouyang *et al.*, 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[Team *et al.*, 2024] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[Team, 2024] Qwen Team. Qwen2.5: A party of foundation models, September 2024.

[Wang *et al.*, 2024a] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves LLM search for code generation. *CoRR*, abs/2409.03733, 2024.

[Wang *et al.*, 2024b] Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4706–4721. Association for Computational Linguistics, 2024.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *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*, 2022.

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

[Yao *et al.*, 2023] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *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*, 2023.

[Yu *et al.*, 2023] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. *arXiv preprint arXiv:2312.14187*, 2023.

[Zhang *et al.*, 2024a] Dan Zhang, Sining Zhoubian, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: LLM self-training via process reward guided tree search. *CoRR*, abs/2406.03816, 2024.

[Zhang *et al.*, 2024b] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. *CoRR*, abs/2410.10762, 2024.

[Zheng *et al.*, 2024] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient finetuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics.