Tree-of-Code: A Tree-Structured Exploring Framework for End-to-End Code Generation and Execution in Complex Task Handling

Anonymous ACL submission

Abstract

Solving complex reasoning tasks is a key real-world application of agents. Thanks to the pre-training of Large Language Models (LLMs) on code data, recent approaches like CodeAct successfully use code as LLM agents' action, achieving good results. However, CodeAct greedily generates the next action's code block by relying on fragmented thoughts, resulting in inconsistency and instability. Moreover, CodeAct lacks action-related ground-truth (GT), making its supervision signals and termination conditions questionable in multi-turn interactions. To address these issues, we first introduce a simple yet effective end-to-end code generation paradigm, CodeProgram, which leverages code's systematic logic to align with global reasoning and enable cohesive problem-solving. Then, we propose Tree-of-Code (ToC), which self-grows CodeProgram nodes based on the executable nature of the code and enables self-supervision in a GT-free scenario. Experimental results on two datasets using ten popular zero-shot LLMs show ToC remarkably boosts accuracy by nearly 20% over CodeAct with less than 1/4 turns. Several LLMs even perform better on one-turn CodeProgram than on multi-turn CodeAct. To further investigate the trade-off between efficacy and efficiency, we test different ToC tree sizes and exploration mechanisms. We also highlight the potential of ToC's end-to-end data generation for supervised and reinforced fine-tuning.

1 Introduction

Large language models (LLMs) significantly improve agents' ability to leverage external tools. (Chen et al., 2023b; Hong et al., 2023; Paul, 2024). Effectively and efficiently handling complex real-world problems (Blount and Clarke, 1994), especially those requiring multiple tools and calls (Li et al., 2023b; Wang et al., 2024), has become a key focus across industry and academia. Currently, the widely used paradigm, ReAct, (Yao et al., 2022), combines reasoning with action strategies, allowing for actions to be performed incrementally and adjusted based on environmental feedback.

The application of code generation techniques to complex task planning and execution has garnered significant attention (Holt et al., 2024; Wen et al., 2024b; Xu et al., 2024), particularly with the emergence of CodeAct (Wang et al., 2024) approaches. CodeAct moves the interaction unit from ReAct's individual tool calls to generating code blocks with local reasoning, while uniquely using code logic and libraries. Rather than JSON (Qin et al., 2023) or text (Park et al., 2023), it treats code as action, utilizing LLM's pre-trained coding skills for efficient handling of complex tasks.

However, each turn of CodeAct is based on individual actions rather than the entire program. It can not autonomously reason and generate complete code in one turn. Instead, it follows a step-by-step generation and interaction process. This is akin to the brain's control of motor actions, processing tasks iteratively and incrementally, with the basal ganglia supporting step-granularity execution and the cerebellum refining and sequencing for the complete complex motor task (Baladron et al., 2023).

On the one hand, the fragmented and stalled thinking in code can hinder a thorough understanding of the logical chains embedded within it, potentially leading to redundant code (Wang et al., 2023; Guo et al., 2024). Repeatedly integrating all prior thoughts causes context overload due to long histories, making it more prone to accumulating significant model hallucinations (Ji et al., 2023).

On the other hand, solving complex problems can have multiple solutions (Mialon et al., 2023), such as calling tools in different sequences. LLMs also tend to explore randomly, resulting in varied solutions, making it difficult to set a standard answer for each step. Long action sequences over multiple turns only provide sparse rewards (Xu et al., 2023). When using these trajectories as training data for supervised fine-tuning (SFT), they can only be combined into an overall program response as one sample (Wang et al., 2024). Applying reinforcement learning is challenging due to the lack of necessary process supervision (Zelikman et al., 2024), leading to fundamental issues.

Therefore, we are considering how to use task-level feedback as a supervision signal for each turn. Additionally, how can we design a framework that incorporates reflection and refinement based on environmental feedback to create real multi-turn trajectory data?

Here, we are inspired by the Dorsolateral Prefrontal Cortex (DPC), which is involved in high-level global cognitive circuits (Kaller et al., 2015), to simulate how programmers form systematic representations and utilizations of code. We first propose an end-to-end code reasoning and generation paradigm, dubbed CodeProgram. In this way, the final answer can serve as a direct evaluation metric.

To utilize environmental feedback from code execution, we further develop a framework called Tree-of-Code (ToC). In this method, task-level CodePrograms serve as nodes, forming a self-expanding exploration tree driven by the verifiability of code execution. This differs from "Tree-of-Thoughts" (ToT) (Yao et al., 2024a), which enhances "Chain-of-Thought" (CoT) (Wei et al., 2022) by exploring varied approaches within the same solution. In contrast, we generate diverse solutions through random settings, with each node representing a complete solution. This concept is akin to a Code "Random Forest" (Rigatti, 2017). Experimental results on two types of multi-turn, multi-tool, complex task datasets, using ten LLMs, demonstrate that ToC outperforms CodeAct in both accuracy and efficiency. The core contributions of this paper are summarized as follows:

1. We introduce CodeProgram, an end-to-end paradigm designed to continuously generate complete code solutions using necessary tools. The purpose and its advantages are analyzed.

2. We present Tree-of-Code (ToC), which self-grows CodeProgram nodes by leveraging the executable nature of the code. By exploring random settings, ToC enhances performance in solving complex tasks through ensembling.

3. Empirical validation on two complex task datasets, M3ToolEval and API-Bank, demonstrates significant improvements in both problem-solving performance and efficiency.

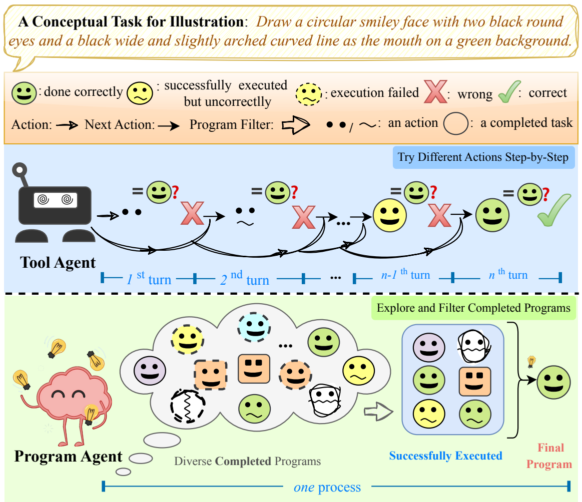


Figure 1: Illustration of our design motivation.

2 Design Motivation

In industry, complex tasks requiring multiple tools and function calls, are typically driven by open-ended user queries. This creates two key challenges: (1) For zero-shot queries, it is challenging to pre-obtain task-level ground-truth (GT). Manual annotation or assigning different rewards to responses is required for subsequent SFT (Chung et al., 2024) or reinforced fine-tuning (ReFT) (Luong et al., 2024). Moreover, without GT, the termination criteria become unclear. (2) Multi-turn interactions lack a fixed trajectory, making it difficult to define the process supervised signals (Luo et al., 2024). Current methods often rely on a 'judge' model to evaluate whether the user's needs from the task query are met at each step (Chen et al., 2024; Li et al., 2024a). However, each evaluation demands strong analytical and reasoning skills from the model, making it costly and time-consuming. Existing methods deliberately avoid these challenges by assuming GT is known, matching task-level GT with action-related outcomes at each step. The process stops if they match, or continues until the step limit is reached. The tool agent in Figure 1 illustrates this.

We aim to explore whether it is possible to develop a method that can self-provide supervision and termination criteria at each step while approximating GT in a GT-free scenario.

Unlike cerebellum-controlled step-by-step motor tasks, we learn from the DPC to treat each turn as a complete task. By iteratively approaching the feasible region, we collect a batch of feasible solutions and then consider the optimal one. Inspired by it, we represent the program agent in Figure 1 to simulate the cognitive process of senior programmers. When working on a project, they start with global planning, complete the entire code, and iteratively debug until no errors are reported, leveraging the systematic logic and environmental feedback inherent to the code project.

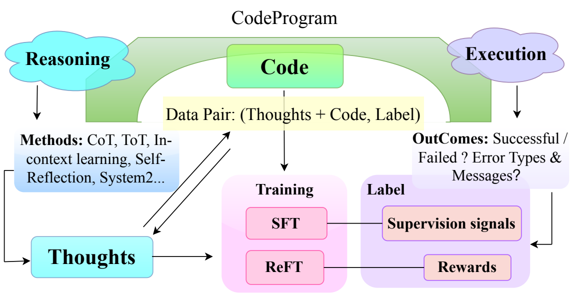


Figure 2: Illustration of CodeProgram.

3 CodeProgram

A block of code in CodeAct's actions corresponds to the eyes or mouths of the smiley face in Figure 1, when faces represent complex tasks. We propose CodeProgram, which can draw the complete face in one turn. Figure 2 illustrates how it works. Specifically, code serves as a bridge, aligning with natural language reasoning and connecting it to execution outcomes in the environment. By decoupling the reasoning process from code execution, we achieve flexibility while ensuring consistency.

3.1 Code as Reasoning

Code generation plays a crucial role in the concept of "code-as-reasoning," where the process of writing code itself reflects a reasoning process.

Global reasoning is required to guide complete code generation in a single end-to-end process. This enables the seamless integration of various reasoning methods for large language models (LLMs), such as prompt engineering (Chen et al., 2023a), Chain-of-Thoughts (CoT) (Wei et al., 2022), Tree-of Thoughts (ToT) (Yao et al., 2024a), in-context learning (Kojima et al., 2022), self-reflection (Zhang et al., 2024), and System2 reasoning (Frankish, 2010; OpenAI, 2024b; Yao et al., 2024b). Furthermore, longer chains of thought have consistently been shown to improve task performance (Wei et al., 2022; Zelikman et al., 2024).

Building on this foundation of global reasoning, we write the root prompt based on previous work (Wang et al., 2024) to guide the generation of step-by-step CoT thoughts and the corresponding complete code. LLMs are prompted to first analyze and break down the problem, generate reasoning-based thoughts for solving it, and then produce the complete code that reflects and executes that reasoning. The thoughts and codes are enclosed using the "*<thought>-</thought>*" and "*<execute>-</execute>*" tags, respectively. The root prompt is shown in Appendix A.

3.2 Two Helper Tools

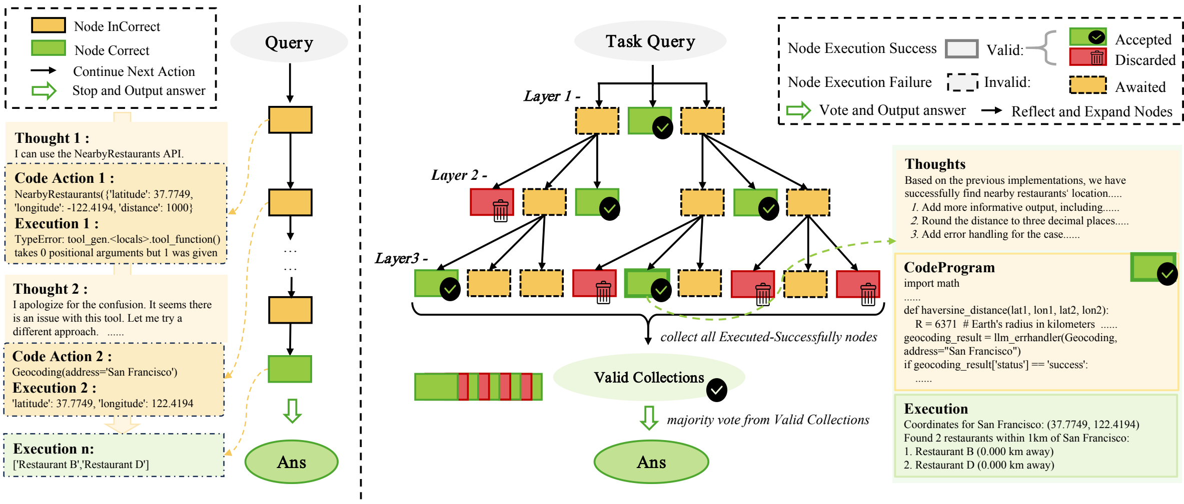
CodeProgram struggles when LLMs must rely on tool outputs to determine the next steps. For example, we can only provide the final summary answer based on the outputs of the tools; in web browsing tasks, the next action is determined only after the page content is viewed. To maintain end-to-end flow, we introduce two additional functions that call the LLM into our code: a general **res_handler**, which defines a prompt to generate results that meet the prompt requirements for final summarization, and a **next_action** for web tasks, which decides the next action from a given set of possible browsing actions based on the page content. The tool descriptions and functions are shown in Appendix B. They help better understand the semantic relationships between tools, ensuring a smooth and cohesive sequence of tool calls during code generation.

3.3 Execution Outcome can be Label

The code solution is task-level, and its execution outcome is a self-provided annotation that can be directly used as labels. Specifically, it includes "successfully executed or failed", 0/1 supervision signals for SFT, and various rich comments (such as specific results or error messages) that can be quantified as rewards for ReFT, as long as we repeat the CodeProgram in different settings multiple times to receive feedback. In this way, the code acts as a verifier. This idea inspires us to build a single-layer multi-node Tree-of-Code (ToC).

Besides, thanks to the task-level granularity, the code's execution outcomes align with the task query and thought-code output, which helps generate valuable training data as the thoughts, code, and execution-based labels are strictly aligned.

In summary, CodeProgram is an annotation-free, end-to-end generation approach well-suited for producing large-scale training data, increasing efficiency, and improving task-solving performance.



(a) CodeAct

(b) Tree-of-Code (ToC)

Figure 3: An Overview of **CodeAct** and **ToC**. (a) CodeAct regards code as action with step-by-step reasoning. (b) ToC applies execution-based reflection in the tree structure, where each node generates end-to-end code with global planning as its thoughts. At each layer, different nodes are executed in parallel; if executed successfully, they are collected for voting. Yellow boxes with dashed borders indicate invalid nodes that fail execution. Both red and green boxes represent valid nodes: red boxes are discarded through LLM voting, while green boxes are finally accepted. The query is "Find nearby restaurants within 1km of San Francisco" from API-Bank level-3 dataset.

4 Tree-of-Code Method

Following the design motivation in Figure 1, we need to collect all successfully executed solutions and identify the one closest to the GT. CodeProgram introduced in Section 3 has allowed us to achieve two key goals: (1) to directly reflect on and refine the task-level code, and (2) to use its execution results as terminal criteria. We now propose an execution-based, self-growing, and self-filtering tree, with CodeProgram as its nodes.

4.1 Overview of Tree-of-Code

We represent the ToC framework as $\boldsymbol{T} = (\boldsymbol{N}, \boldsymbol{S})$, where $\boldsymbol{N}$ denotes a set of nodes (N), and $\mathbf{S}$ represents the stems (unidirectional arrows in Figure 3) , modeling the reflection reasoning process of LLMs when expanding the nodes. The overview of ToC and how it works is illustrated in Figure 3. Let L denote the max depth, l the layer index, M the expanded layer's max width, m the node index, $l \in \{1, \ldots, L\}$, $m \in \{1, \ldots, M\}$. We use T for the thoughts of the N, C for code, and E for its execution result. The next-layer N is denoted as:

$$N_{(l+1)\text{-}m} = S_{l \to (l+1)}(f, \sum_{j=0}^{l}(T_{j\text{-}m} + C_{j\text{-}m} + E_{j\text{-}m}))$$

where f represents the basic information of the task, such as the user's query, and all tool descriptions. The sum $\sum_{j=0}^{l}$ indicates that each reflection reasoning process for generating the next node relies on the thoughts, code, and execution results from all ancestor nodes in the history. The node index is fixed for simplicity in the formula.

4.2 Tree Expansion

We initialize from a root node and recursively expand the tree. The expansion process follows: (1) The breadth-first search (BFS) strategy is applied, with each parent node branching into M child nodes. (2) Whether the node continues to grow depends solely on the evaluation of its own execution state (success or failure). For each N_l,

$$\begin{cases} \textit{stop and collect}, & \textit{if } E_l \neq \textit{None or error}, \\ \textit{grow } \boldsymbol{N}_{(l+1)}, & \textit{otherwise}. \end{cases}$$

(3) Expansion continues until all child nodes stop or the maximum depth (L) of 3 is reached.

Execution-based Reflection. We can not guarantee that the end-to-end code solution will be correct on the first attempt. Treating task-level execution errors as continuation signals, we propose execution-based reflection, which enables LLMs to self-reflect, identify errors, refine thoughts, and improve code through prompting, significantly enhancing problem-solving. The prompt for reflection is shown in Appendix A.2.1.

As long as execution fails, self-reflection continues iteratively, generating new nodes. This allows the branch to grow as a data sample, with

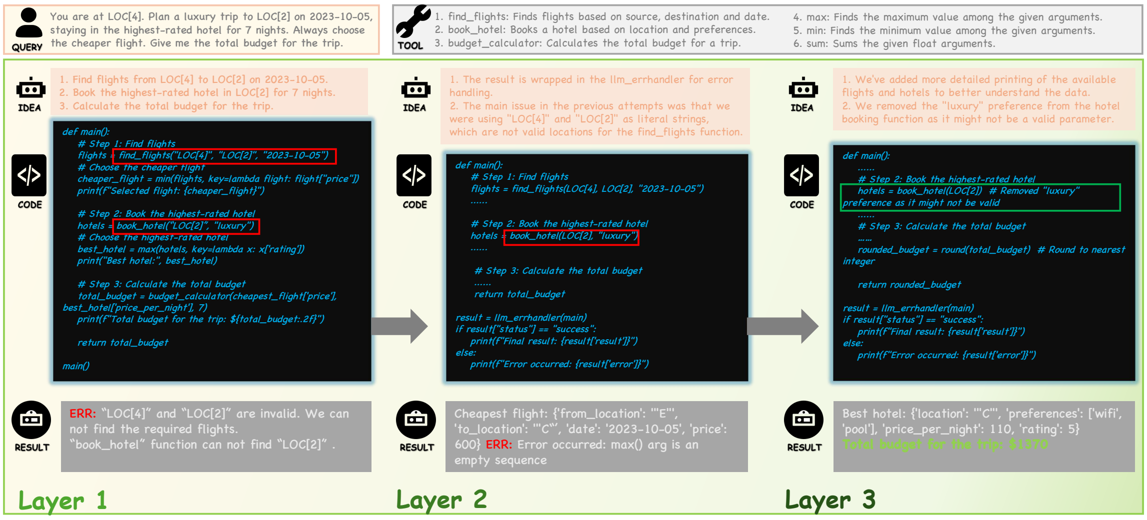


Figure 4: Illustrative example of a branch of ToC. We demonstrated the process of a node expanding into deeper levels. Based on the user query, tool descriptions, and previous execution outcomes, ToC first thinks about how to do it and then writes the end-to-end code. The example is selected from M3ToolEval dataset.

each node in the trajectory providing supervision signals. Since these supervision signals are inherently embedded within the CodeProgram node, the growth process is self-driven. Therefore, the whole tree is end-to-end generated. Figure 4 illustrates an example of a branch of ToC.

Additionally, our flexible tree-structured framework allows for the integration of any reflection method in end-to-end code generation.

Exploration strategy. Generating code in a single pass presents two main limitations on diversity:

- 1) Limited strategy: It easily leads to cognitive narrowing, where the fundamental reasoning mechanism remains unchanged.
- 2) Limited robustness: If an error occurs, the only option for the user is to re-run the whole process, without any proactive adjustments, which leads to inefficiencies.

Research (Renze and Guven, 2024) has shown that performance benefits from diverse perspectives of error identification, which encourages models to generate multiple solutions.

To enhance the diversity of ToC, we introduce randomness into the expansion process by varying LLMs and prompts, inspired by the random forest (Rigatti, 2017). At the system level, different LLMs are randomly explored from our LLM list, which will be introduced in Section 5.1, with a consistent temperature setting of 0.1. At the instruction level, prompts are randomly selected from a diverse pool, designed through self-evolution and human crafting phases. In the first phase, we used our ten LLMs to create ten various prompts based on the prompt evolution with the root prompt (see Appendix A). The prompt for prompt evolution requires maintaining consistent core content with the root prompt while encouraging orthogonal or divergent expressions. Then, we manually selected six distinct prompts, randomly applying one or more of the following modifications: (1) Adding more detailed usage examples (beyond just printing "Hello world") to three prompts; (2) Adjusting their format by adding line breaks and indentation; (3) Randomly rearranging components, such as the reflection part, usage examples, role instructions, tool descriptions, and the chat history.

4.3 Final Result Generator

Once valid outputs from successfully executed nodes are collected, the same LLM makes the final decision by performing a majority vote and summarization to determine the most likely answer — this corresponds to the green node in Figure 3. Other valid responses (red nodes) are discarded.

5 Experiment and Analysis

5.1 Setup

Datasets. Following CodeAct, our evaluation is based on M3ToolEval[1] (M3) (Wang et al., 2024) and the test set of API-Bank[2] (Li et al., 2023b). M3

[1] https://github.com/xingyaoww/code-act/tree/main/scripts/eval/m3tooleval

[2] https://huggingface.co/datasets/liminghao1630/API-Bank/tree/main

Model	M3ToolEval				API-Bank level-3			
	ReAct	**CodeAct**	**CodeProgram**	**ToC (1-3)**	**ReAct**	**CodeAct**	**CodeProgram**	**ToC (1-3)**
claude-instant-1	28.0% (8.7)	18.0% (8.9)	30.5% (1)	35.3% (1)	0.0% (10.0)	2.0% (10.0)	6.0% (1)	18.0% (1)
claude-2	40.2% (8.2)	54.9% (7.2)	57.3% (1)	59.8% (1)	0.0% (10.0)	20.0% (8.9)	8.0% (1)	18.0% (1)
claude-3-haiku	24.4% (9.0)	9.8% (9.4)	29.3% (1)	31.7% (1)	10.0% (9.4)	0.0% (10.0)	6.0% (1)	8.0% (1)
claude-3-5-sonnet	48.8% (7.7)	73.2% (5.7)	**73.2%** (1)	**82.9%** (1)	14.0% (9.3)	32.0% (7.8)	**48.0%** (1)	**52.0%** (1)
gpt-3.5-turbo-1106	18.3% (8.9)	25.6% (8.6)	12.2% (1)	17.1% (1)	14.0% (9.2)	2.0% (9.9)	4.0% (1)	8.0% (1)
gpt-4-1106-preview	**54.9%** (7.5)	**75.6%** (5.4)	72.0% (1)	73.2% (1)	**18.0%** (8.2)	30.0% (8.2)	34.0% (1)	38.0% (1)
gpt-4o-mini-2024-07-18	32.9% (8.4)	47.6% (7.0)	31.7% (1)	42.7% (1)	10.0% (9.6)	16.0% (9.5)	14.0% (1)	20.0% (1)
gpt-4o-2024-08-06	35.4% (8.5)	56.1% (6.7)	51.2% (1)	62.2% (1)	14.0% (9.4)	**36.0%** (7.8)	28.0% (1)	32.0% (1)
qwen2.5-72b-instruct	50.0% (7.9)	70.7% (5.6)	51.2% (1)	59.8% (1)	2.0% (9.9)	30.0% (8.2)	24.0% (1)	32.0% (1)
deepseek-chat	47.6% (7.6)	62.2% (5.9)	40.2% (1)	52.4% (1)	0.0% (9.8)	24.0% (8.6)	22.0% (1)	26.0% (1)
Avg.	**38.05% (8.24)**	**49.37% (7.04)**	**43.53% (1)**	**50.98% (1)**	**8.2% (9.48)**	**19.2% (8.89)**	**19.4% (1)**	**24.4% (1)**

Table 1: Detailed performance comparison of different models under ReAct, CodeAct, CodeProgram, and one-layer, three-node Tree-of-Code (ToC) on the M3ToolEval and API-Bank level-3 datasets. The correctness is reported, with the average number of turns in parentheses. The noteworthy point is that since the model is fixed, the ToC mechanism at this time represents the ToC mechanism without model exploration (w/o model exploration).

consists of 82 tasks utilizing 100 tools in code/JSON/txt action space respectively across 5 types of scenarios, including DNA sequencer, message decoder, trade calculator, travel itinerary planning, and web browsing. API-Bank contains 314 tool-use dialogues and 73 API tools, including level-1, 2, 3. Unlike CodeAct, which evaluates only on level-1, we focus directly on the 50 most challenging level-3 tasks, on which nearly all non-GPT4 models score 0%, according to the original paper.

For M3, we add the "next_action" tool, which is a customized function introduced in Section 3.2. For API-Bank, which only supports JSON format, we make the following modifications to adapt it for code interaction: (1) functionalize all API tools, (2) add output examples to each function description (shown in Figure 7). We include all tool signatures in the prompt context and let LLMs inherently search and select tools, instead of using ToolSearch API, deemed the least essential in (Li et al., 2023b). (3) determine correctness by matching the response to the expected final output through conditional keywords, not by API call matching.

Models. We include the following ten models in our model pool for evaluation: the GPT family from OpenAI (Achiam et al., 2023; Bubeck et al., 2023; OpenAI, 2024a), including gpt-3.5-turbo-1106, gpt-4o-mini-2024-07-18, gpt-4o-2024-08-06, and gpt-4-1106-preview checkpoints, excels in generation capabilities. From the Anthropic's Claude family (Anthropic, 2023, 2024), we select claude-instant-1, claude-2, claude-3-haiku-20240307, and claude-3-5-sonnet-20240620 known for their code generation and problem-solving capabilities. Besides, we incorporate open-sourced deepseek-chat from DeepSeek (Guo et al., 2024) and qwen2.5-72b-instruct from Alibaba (Bai et al., 2023).

Baselines. ReAct (Yao et al., 2022) combines reasoning and action in a dynamic, step-by-step interaction, providing a flexible approach to task-solving by adjusting action strategies based on environmental feedback. CodeAct (Wang et al., 2024) replaces the JSON in ReAct with a block of code as the LLM agent's action, enabling multi-turn interactions and effectively expanding the action space for solving complex real-world problems.

Metrics. The evaluation includes accuracy and averaged turns. Accuracy represents the percentage of complex tasks that are correctly solved. Each LLM-generated code is considered one turn. For parallel generation, the number of threads counts as turns in terms of resource usage, but when considering time, multiple parallel generations count as one turn. We use the latter approach.

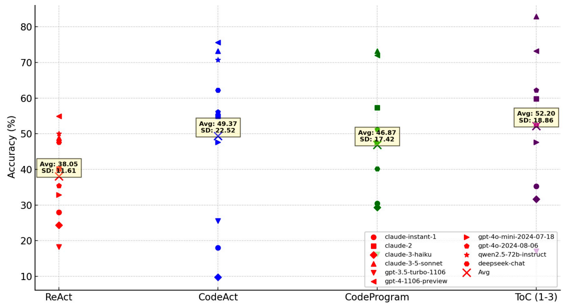


Figure 6: Performance of 10 LLMs on ReAct, CodeAct, CodeProgram, and 1-3 ToC for the M3 dataset is visualized, with average and standard deviation reported.

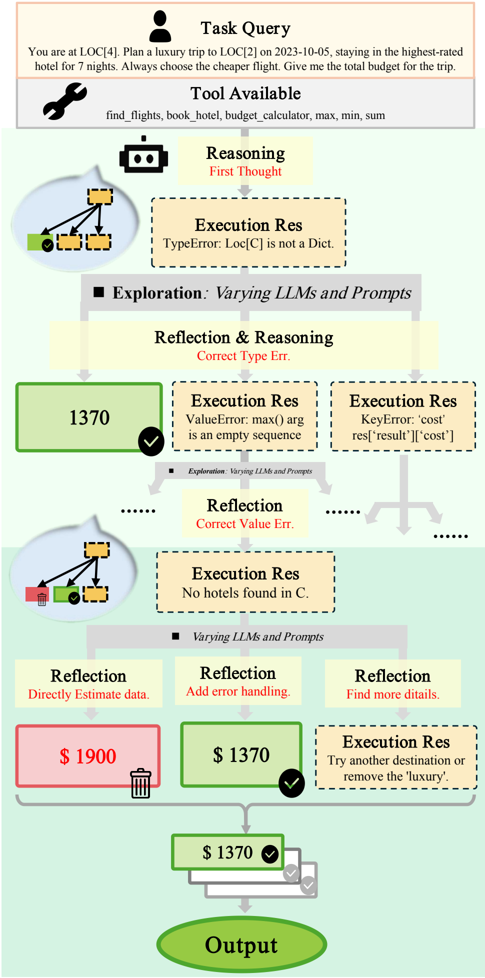


Figure 5: A detailed example illustrating ToC's execution-based reflection and expansion.

5.2 One-turn vs. Multi-turn

CodeProgram enables global planning and complete solutions in a single turn by leveraging code's ability to handle long logic chains, aligning with global reasoning in language, and defining clear tool inputs and outputs. With a significant advantage in the number of turn, Table 1 and Figure 6 demonstrate that the performance of some models even surpasses the multi-turn CodeAct, particularly for the Claude series models. We grow CodeProgram into a single-layer, three-node (1-3) Tree-of-Code (ToC). The prompt for each node is randomly sampled from our prompt pool. Compared to CodeProgram, the simple 1-3 tree-structured ToC with random prompts significantly boosts performance. The average performance of 1-3 ToC already surpasses CodeAct, highlighting the power of prompt randomness. Upon reviewing the generated code, we observe that LLMs often produce modular code for each step or add comments before modules, even when not explicitly required.

We highlight the best-performing models in bold. Experimental results show that the top models differ between the CodeAct and ToC, and even within CodeAct, performance varies by dataset. For M3, gpt-4 performs best, while for API-Bank level-3, gpt-4o excels, likely because API-Bank level-3 emphasizes tool usage over scenario understanding, with simpler problem expressions. For ToC, claude-3-5-sonnet stands out due to its strong prompt-following ability, which is key for aligning reasoning with code and tool selection.

Example of the Function Signature in API-Bank level-3

```
UserMoviePreferences():
    description: "API for retrieving user preferences for
movie recommendations. Here is an example of the output:
result = {'api_name': 'UserMoviePreferences', 'input':
{'user_name': 'John'}, 'output': {'preferences': ['Action',
'Comedy', 'Drama']}, 'exception': None}"
    input_parameters:{
        'user_name': {'type': 'str', 'description': 'Name of
the user.'},
    }
    output_parameters:{
        'preferences': {'type': 'list', 'description': 'List
of movie preferences.'},
    }
```

Figure 7: Example of the function signature in level-3.

5.3 ToC vs. CodeAct and ReAct

We primarily compare the ToC framework, which is comprised of CodeProgram nodes, with the CodeAct and ReAct framework, which are comprised of steps, using the M3 and the level-3 datasets. For ToC, we randomly sample the LLM and prompt from the LLM list and prompt pool, respectively, at each node exploration. For CodeAct and ReAct, we report the average results across all models used in this paper. Table 2 shows that ToC consistently achieves superior performance and demonstrates a significant advantage with fewer interaction steps, highlighting its efficiency in managing complex tool-use scenarios.

Mechanism	M3ToolEval		API-Bank level-3	
	Avg Turns	Correct	Avg Turns	Correct
ReAct	8.2	38.1%	9.5	8.2%
CodeAct	7.0	49.4%	8.9	19.2%
Tree-of-Code	1.7	67.1%↑	2.1	38.0% ↑

Table 2: Performance comparison of CodeAct and our ToC in terms of averaged turns and accuracy on M3 and API-Bank level-3 tasks. Note: all numerical results presented in this paper are rounded to one decimal place.

5.4 Analysis and Ablation Studies

Varying tree sizes. We test the performance of the top model, claude-3-5-sonnet, on different tree sizes to evaluate the trade-off between efficacy

Layer / Node Per Layer	1	2	3
1	73.2% (1)	75.6% (1)	82.9% (1)
2	73.2% (1.4)	76.8% (1.4)	84.1% (1.5)
3	74.4% (1.8)	79.3% (1.7)	84.1% (1.6)

Table 3: The performance of varying tree sizes.

and efficiency. Table 3 shows impressive results: with proper prompts and no additional training, the model achieves 84.1% accuracy (3-3) on the M3, 10.9% higher than 73.2% (1-1).

Visualization results for ReAct, CodeAct, and 3-3 ToC on the M3 dataset (Figure 8) show that ToC achieves near-perfect accuracy on all tasks except the web browsing task.

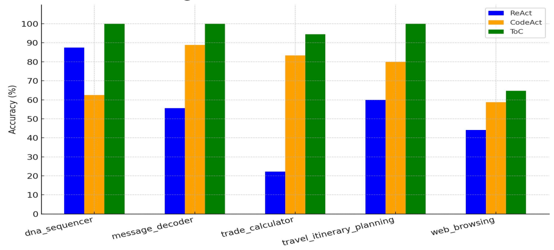


Figure 8: Comparison across five tasks in the M3.

Prompt exploration. The ablation results in Table 4 demonstrate the effectiveness of prompt exploration. By comparing the random model with the fixed model, prompt exploration proves more crucial in low-diversity scenarios.

Mechanism	M3ToolEval	
	Avg Turns	Correct
Random Model ($\Delta = 3.7\%$)		
ToC	1.7	67.1%
ToC w/o prompt exploration	1.9	63.4% $\downarrow$
Fixed Model (the best) ($\Delta = 8.5\%$)		
ToC w/o model exploration	1.6	84.1%
ToC w/o model+prompt exploration	1.8	75.6% $\downarrow\downarrow$

Table 4: Results of the ablation study.

6 Discussion

Why we do not try a search tree? We initially explored using evolutionary algorithms to merge nodes from different branches for the next generation, aiming to reduce the search space. However, after extensive testing, we found this approach impractical and hard to implement. We analyzed error cases from ToC and found that a single branch can easily fall into specific errors. Since different branches follow distinct reasoning paths, they struggle to learn from each other. Even with reflection and random exploration, LLMs are prone to early-stage errors that disrupt subsequent reasoning (An et al., 2023; Bao et al., 2024; Tong et al., 2024). This issue mirrors the human cognitive challenge of breaking out of a "bistable state" — like the famous "duck-rabbit illusion" or Rubin's vase (Hancock, 2013). In such states, a person may generate multiple parallel thoughts, but once one is chosen, it is difficult to switch to another without external intervention (Andreev et al., 2020). This highlights the need for introducing random exploration and multiple nodes at each growth step, as these strategies help overcome cognitive bottlenecks.

7 Related Work

LLM Code Generation. Chain of Codes framework (Li et al., 2023a) expands the range of reasoning tasks that LLMs can solve by "thinking in code." Similarly, CodePlan (Wen et al., 2024a) utilizes pseudocode to guide structured reasoning. Additionally, the Structured Chain-of-Thought Prompting (SCoT) technique (Esfahani et al., 2024) has highlighted the potential of structured prompts in this domain. Recent works combining code with agents have primarily focused on task completion within programming-related domains, such as software development (Qian et al., 2024; Wang et al., 2023), programming assistance (Islam et al., 2024; Wen et al., 2024c), and scientific problems (Chen et al., 2022; Gao et al., 2023; Hong et al., 2024). Few methods (Wang et al., 2024) treat code as a scalable language format to call multiple tools for solving complex real-world tasks.

Recently, we found a new work, CodeTree (Li et al., 2024b), which uses a tree structure to explore the search space of code generation tasks. Unlike our approach, it focuses on multi-agent searching rather than an end-to-end self-growing tree. Additionally, it was released three months later than the initial submission of our work.

8 Conclusion

In this paper, we introduced the Tree-of-Code (ToC) method, which combines execution-based reflection with end-to-end thought-code generation for complex task handling. With efficient model integration and prompt exploration, ToC significantly outperformed the baselines on two complex task datasets, boosting both efficiency and task-solving capabilities. The ToC framework opens up exciting possibilities for advancing human-like global cognition, inspiring further exploration of tree structures in end-to-end code generation, particularly for complex multi-tool tasks, data generation, and reinforced fine-tuning.

Limitations

Additional engineering effort

We transform multi-turn interactions into a single-turn complete program containing a series of tool calls with great efficiency. However, this approach requires more detailed usage instructions for the tools, especially for actions with limited semantic information (e.g., webpage clicking and scrolling). Therefore, additional engineering effort would be required to implement our end-to-end code generation approach.

Limited reasoning scope for Program

We emphasize that our method operates at the granularity of code "program" rather than "action". However, it is limited in fully open-ended scenarios requiring step-by-step exploration, such as a robot navigating an unfamiliar environment, or in handling tasks with extremely long sequences beyond the capabilities of current reasoning methods, like generating an entire paper. In such cases, it cannot provide a complete final solution. For larger and more complex system programs in the future, our method may serve as a "subprogram" within the overall solution, similar to a single agent's role in multi-agent systems.

Opportunities for Reflection Refinement

While our framework provides a solid foundation inspired by human problem-solving, it uses a basic reflection mechanism, relying on execution feedback alone. Whether tracking full execution history or selectively summarizing with LLMs offers better performance remains an open question. Future research could explore enhanced search strategies or adaptive pruning methods to handle more complex real-world tasks.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2023. Learning from mistakes makes llm better reasoner. *arXiv preprint arXiv:2310.20689*.

Andrey V. Andreev, Nikita S. Frolov, Natalija A. Alexandrova, and Marija A. Chaban. 2020. Control of dynamics of bistable neural network by an external pulse.

Anthropic. 2023. Introducing claude. `https://www.anthropic.com/index/introducing-claude`. Accessed: 2023-10-20.

AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 1.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Javier Baladron, Julien Vitay, Torsten Fietzek, and Fred Henrik Hamker. 2023. The contribution of the basal ganglia and cerebellum to motor learning: A neuro-computational approach. *PLOS Computational Biology*, 19.

Guangsheng Bao, Hongbo Zhang, Linyi Yang, Cunxiang Wang, and Yue Zhang. 2024. Llms with chain-of-thought are non-causal reasoners. *arXiv preprint arXiv:2402.16048*.

GN Blount and S Clarke. 1994. Artificial intelligence and design automation systems. *Journal of Engeering Design*, 5(4):299–314.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2023a. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023b. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*.

Mouxiang Chen, Hao Tian, Zhongxi Liu, Xiaoxue Ren, and Jianling Sun. 2024. Jumpcoder: Go beyond autoregressive coder via online modification. In *Annual Meeting of the Association for Computational Linguistics*.

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

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53.

Ali Mohammadi Esfahani, Nafiseh Kahani, and Samuel A Ajila. 2024. Understanding defects in generated codes by language models. *arXiv preprint arXiv:2408.13372*.

Keith Frankish. 2010. Dual-process and dual-system theories of reasoning. *Philosophy Compass*, 5(10):914–926.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Roeland Hancock. 2013. Attentional processing in bistable perception is influenced by genetic effects associated with sinistrality. *Cognitive Science*, 35.

Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2024. L2mac: Large language model automatic computer for extensive code generation. In *The Twelfth International Conference on Learning Representations*.

Sirui Hong, Yizhang Lin, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Lingyao Zhang, Mingchen Zhuge, et al. 2024. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Christoph P. Kaller, Marco Reisert, Michael L. Katzev, Roza M. Umarova, Irina Mader, Jürgen Hennig, Cornelius Weiller, and Lena Köstering. 2015. Predicting planning performance from structural connectivity between left and right mid-dorsolateral prefrontal cortex: moderating effects of age during postadolescence and midadulthood. *Cerebral cortex*, 25 4:869–83.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023a. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*.

Jierui Li, Hung Le, Yinbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024a. Codetree: Agent-guided tree search for code generation with large language models.

Jierui Li, Hung Le, Yinbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024b. Codetree: Agent-guided tree search for code generation with large language models.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. 2024. Improve mathematical reasoning in language models by automated process supervision. *ArXiv*, abs/2406.06592.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. 2024. Reft: Reasoning with reinforced fine-tuning. *ArXiv*, abs/2401.08967.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. *ArXiv*, abs/2311.12983.

OpenAI. 2024a. Hello gpt-4o. `https://openai.com/index/hello-gpt-4o/`. Accessed: 2024-05-18.

OpenAI. 2024b. Openai o1 system card. `https://cdn.openai.com/o1-system-card-20240917.pdf`. Accessed: 2024-09-12.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*.

Swarna Kamal Paul. 2024. Continually learning planning agent for large environments guided by llms. In *2024 IEEE Conference on Artificial Intelligence (CAI)*, pages 377–382. IEEE.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.

Yujia Qin, Shi Liang, Yining Ye, Kunlun Zhu, Lan Yan, Ya-Ting Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Marc H. Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ArXiv*, abs/2307.16789.

Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.

Steven J Rigatti. 2017. Random forest. *Journal of Insurance Medicine*, 47(1):31–39.

Yongqi Tong, Dawei Li, Sizhe Wang, Yujia Wang, Fei Teng, and Jingbo Shang. 2024. Can llms learn from previous mistakes? investigating llms' errors to boost for reasoning. *arXiv preprint arXiv:2403.20046*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2023. Leti: Learning to generate from textual interactions. *arXiv preprint arXiv:2305.10314*.

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

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. 2024a. Codeplan: Unlocking reasoning potential in large langauge models by scaling code-form planning. *arXiv preprint arXiv:2409.12452*.

Jiaxin Wen, Ruiqi Zhong, Pei Ke, Zhihong Shao, Hongning Wang, and Minlie Huang. 2024b. Learning task decomposition to assist humans in competitive programming. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Jiaxin Wen, Ruiqi Zhong, Pei Ke, Zhihong Shao, Hongning Wang, and Minlie Huang. 2024c. Learning task decomposition to assist humans in competitive programming. *arXiv preprint arXiv:2406.04604*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Pei Xu, Junge Zhang, and Kaiqi Huang. 2023. Exploration via joint policy diversity for sparse-reward multi-agent tasks. In *International Joint Conference on Artificial Intelligence*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Wenlin Yao, Haitao Mi, and Dong Yu. 2024b. Hdflow: Enhancing llm complex problem-solving with hybrid thinking and dynamic workflows. *arXiv preprint arXiv:2409.17433*.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. 2024. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*.

Wenqi Zhang, Yongliang Shen, Linjuan Wu, Qiuying Peng, Jun Wang, Yueting Zhuang, and Weiming Lu. 2024. Self-contrast: Better reflection through inconsistent solving perspectives. *arXiv preprint arXiv:2401.02009*.

A Prompt

A.1 Root Prompt

```
You are a helpful assistant assigned with the task of problem-solving.
To achieve this, you will be using an interactive coding environment equipped with a variety of tool
    functions to assist you throughout the process.\n\n
At each turn, you should first provide your step-by-step thinking for solving the task, for example: <
    thought> I need to print "hello world!"</thought>.
After that, you can Interact with a Python programming environment and receive the corresponding output.
Your code should be enclosed using "<execute>" tag, for example: <execute> print("Hello World!") </
    execute>.\n\n
You can use the following functions:\n{toolset_descs}\n.
Ensure the code matches the fn_signature and input-output formats for proper execution.\n
Here's the chat history for your reference:\n{chat_history}\n\n
History End:\n
User's Query:\n{query}\nYour Thought And Code:\n
```

A.2 Additional Prompt

A.2.1 Reflection Prompt

```
Based on the provided chat history, reflect on the code and its execution. Identify potential issues or
    areas for optimization and provide specific suggestions to refine and improve the code. Consider
    edge cases, efficiency, and clarity in your reflections.
```

A.2.2 The Prompt for Prompt Evolution

```
In order to guide the diversity of results and enhance the performance through ensemble methods, we need
    to increase the diversity of prompts. We diversify the current prompt while maintaining consistency
    in core content, aiming for orthogonal expressions or prompts that lead to different directions and
    divergent thinking.
```

A.2.3 The Prompt Sample from Prompt Pool for API-Bank

```
Note:
The outputs produced by the tool will be formatted like a JSON dictionary.
For example, 'result = {{'api_name': 'QueryMeeting', 'input': {{'user_name': 'John'}}, 'output': {{'
    meetings': [{{'meeting_id': 1, 'meeting_name': 'Meeting with the client', 'meeting_time':
    '2021-01-01 10:00:00', 'meeting_location': 'Room 1', 'meeting_attendees': ['John', 'Mary', 'Peter
    ']}}, {{'meeting_id': 2, 'meeting_name': 'Meeting about the new project', 'meeting_time':
    '2021-01-02 10:00:00', 'meeting_location': 'Room 2', 'meeting_attendees': ['John', 'Mary', 'Peter
    ']}}]}}, 'exception': None}}'
Ensure that the code strictly adheres to the function descriptions and the input-output format provided.
Navigate through the 'output' key correctly to retrieve results.
If you encounter any unfamiliar formats, first print the structure to ensure proper handling in the
    future.
Consistently focus on the user's request and attempt to produce the complete solution without needing
    multiple steps.
```

B Helper tools

B.1 ResHandler

B.1.1 ResHandler Tool Description

```
llm_reshandler():
    name="llm_reshandler",
    description='Define a prompt to generate results that meet the prompt requirements. Note that you
        need to define the requirements for the generated results in the prompt. input: prompt (str):
        The input prompt for the large language model, defining the task requirements for the generated
        results. Common tasks include summarization, stylistic writing, translation, question answering,
         etc. output: completion (str): The inference result generated by the large model, typically a
        summary, writing output, translation result, or answer that meets the requirements.',
    function=llm_reshandler,
    fn_signature='llm_reshandler(prompt: str) -> str')
```

B.1.2 ResHandler Tool Function

```
from some_model_API import llm_playground

def llm_reshandler(prompt):
    result_str = ""
    result = llm_playground(prompt[:3000], stream=False)
    for item in result:
        result_str += item
    return result_str
```

B.2 NextAction for Web Task

B.2.1 NextAction Tool Description

```
next_action():
    name="next_action",
    description='Examine the results of the view function to determine if it can answer the user's
        original question, and decide what to do next. The next possible actions include click_url(URL),
        scroll_down(), scroll_up(), go_to_previous_page() and end(), which represent clicking a link,
        scrolling down, scrolling up, go to previous page and end() means you have found the answer page
        , respectively. If next action is end(), it means that relevant information to user query is
        found, you should summarize string result based on llm_prediction. click_url(URL), scroll_down()
        , scroll_up(), go_to_previous_page() can be directly called, and URL should be Clickable url',
    function=next_action,
    fn_signature="next_action(query: str, current_page_content: str) -> str")
```

B.2.2 NextAction Tool Description

```
from some_model_API import llm_playground

def next_action(query, current_page_content):
    prompt = f"You are viewing a web page, the content is: {current_page_content}, you should make
        decision on the next step. given user query {query}, you have the following options. \n1. end():
        it means current user query can be answered by web page content. \n2. click_url(URL): it means
        current user query should be checked by clicking one of the urls shown on the current webpage
        for more details. specify the detailed url into URL field. \n3. scroll_up(): it means current
        page content is not enough to answer user query, you should scroll up current page to check the
        answer of the query. \n4. scroll_down(): it means current page content is not enough to answer
        user query, you should scroll down current page to check the answer. go_to_previous_page(): it
        means that current page does not have relevant information to current query or does not have any
        clickable urls, you should go back to previous viewed page to search more information."
    result_str = ""
    result = llm_playground(prompt[:max_length])

    for item in result:
        result_str += item
print("================next_action_debug===================")
    print("query")
    print(query)
    print("current page content")
    print(current_page_content)
    print("status")
    print(result_str)
print("==========================================")

    if "scroll_up()" in result_str:
        return "scroll_up()"
    elif "scroll_down()" in result_str:
        return "scroll_down()"
    elif "click_url" in result_str:
        import re
        pattern = r"click_url\('.*'\)"
        match = re.search(pattern, result_str)
        if match:
            return match.group()
    elif "go_to_previous_page" in result_str:
        return "go_to_previous_page()"
    elif "end()" in result_str:
```

```
        return "end()"

    return "end()"
```