Tree-of-Code: A Self-Growing Tree Framework for End-to-End Code Generation and Execution in Complex Tasks

Ziyi Ni[1,2,*†], Yifan Li[4,*†], Ning Yang[1], Dou Shen[3], Pin Lv[1,‡], Daxiang Dong[3,‡],
[1]The Key Laboratory of Cognition and Decision Intelligence for Complex Systems,
Institute of Automation, Chinese Academy of Sciences
[2]School of Artificial Intelligence, University of Chinese Academy of Science
[3]Baidu, Inc. [4]Global Innovation Exchange Institution, Tsinghua University
‡ **Correspondence:** dongdaxiang@baidu.com, pin.lv@ia.ac.cn

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 accumulative hallucination. 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 propose Tree-of-Code (ToC), a self-growing framework that generates nodes through self-supervision, incorporating prompt and model exploration in a GT-free setting. Each node employs CodeProgram, an end-to-end code generation paradigm that aligns executable code logic with global reasoning. This approach uses task-level execution success as both node validity and stop-growing flags, bypassing process supervision to enable online applications. Experiments on two datasets with ten popular zero-shot LLMs show that ToC boosts accuracy by nearly 20% over CodeAct with fewer than 1/4 turns. To further investigate the trade-off between efficacy and efficiency, ablation studies on different ToC tree sizes and exploration mechanisms validate ToC's superiority.

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., 2024a; Xu et al., 2024b), 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 leveraging 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, CodeAct treats each turn as an individual action rather than addressing the entire program, following a step-by-step generation process. While this approach may seem explorative, it has four critical limitations: (I). CodeAct assumes that the ground truth (GT) is known and uses GT matching as a termination criterion, which is unrealistic and unfeasible. (II). Fragmented thinking is inefficient. For simple problems, stalled thinking is not only unnecessary but also disrupts the logical chains in the code (Wang et al., 2023; Guo et al., 2024). Moreover, as the number of turns increases, repeatedly integrating prior thoughts causes context overload, heightening model hallucinations (Ji et al., 2023), increasing computational cost. (III). CodeAct lacks exploration of diverse reasoning paths. While it supports multi-turn interactions, it follows a single reasoning process. In contrast, solving complex problems often has multiple solutions (Mialon et al., 2023), where different approaches can branch from different points, making it difficult to set a standard answer for each turn. (IV). Generated trajectory data is hard to reuse. When using these trajectories for supervised fine-tuning (SFT), they cannot be directly combined

[*]These authors contributed equally to this work.
[†]This work was conducted at Qianfan AppBuilder Group during the Baidu AI Cloud Summer Camp internship.

Findings of the Association for Computational Linguistics: ACL 2025, pages 9804–9819
July 27 - August 1, 2025

into a single program response (Wang et al., 2024). Reinforcement learning is also challenging due to the lack of process supervision (Zelikman et al., 2024), leading to fundamental issues.

Since defining and obtaining supervision signals for intermediate states is challenging, we propose using task-level feedback directly, treating task completion as a single step. We introduce CodeProgram, an end-to-end code reasoning and generation paradigm, as a 'turn,' where the only environmental supervision is execution success. To incorporate reflection and exploration, we design an outcome-driven refinement framework, Tree-of-Code (ToC), that enables multi-turn interactions with diverse solutions exploring the model and prompt pools as tree branches, where task-level CodePrograms serve as the nodes. The final output is determined by voting on the collected nodes, selected based on their successful execution. It's important to note that, in this paper, a 'turn' refers to a single action of code generation. For our CodeProgram, a turn involves completing the entire program, rather than just a single task step (as in CodeAct).

Although ToC's name and structure are similar to "Tree-of-Thoughts" (ToT) (Yao et al., 2024), their meanings fundamentally differ. Our concept might be closer to a Code "Random Forest" (Rigatti, 2017). While ToT enhances "Chain-of-Thought" (CoT) (Wei et al., 2022) by exploring different thoughts within the same solution, ToC explores multiple distinct program solutions. In other words, each node in ToC represents a complete solution, and the tree as a whole captures different iterative optimizations (depth) across a variety of complete solutions (breadth). The core contributions of this paper are summarized as follows:

1. We propose a self-growing Tree-of-Code (ToC) structure that automatically reflects and explores diverse, complete solution nodes without labeled data, facilitating complex tasks in multi-tool online scenarios.

2. Each node in ToC, called a CodeProgram, is generated end-to-end. We are the first to define process-level supervision at the task-outcome level using execution success.

3. Extensive experiments and ablation studies on two multi-tool, complex task datasets with ten models, demonstrate that ToC significantly enhances problem-solving accuracy and efficiency in real-world, zero-shot scenarios.

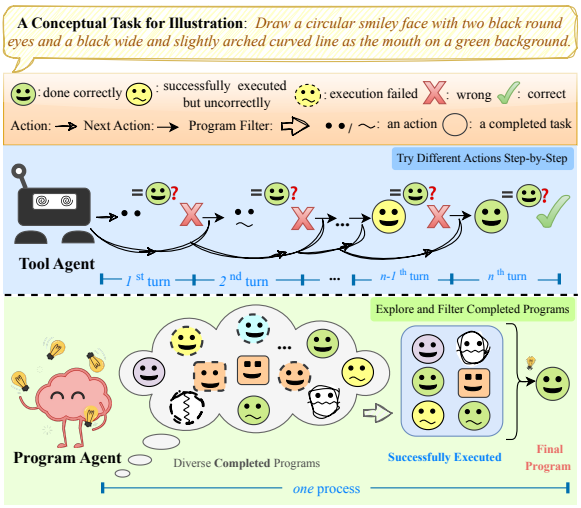

Figure 1: Illustration of our design motivation.

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 unrealistic to pre-obtain task-level ground-truth (GT), which is required for 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 standard trajectory, making it difficult to define the process supervised signals (Luo et al., 2024). Current methods often rely on 'LLM-as-judge' to evaluate whether the user's needs are met at each step (Chen et al., 2024; Li et al., 2024a). However, it would require an API call after every step to check progress, ultimately increasing both time and token costs. Besides, the evaluation without objective signals demands strong analytical and reasoning skills from LLMs. Existing methods deliberately avoid these challenges by assuming GT is known (Wang et al., 2024), matching task-level GT with action-related outcomes at each step, like the tool agent in Figure 1: the interaction turn stops only if they match, or continues until the step limit is reached.

Since intermediate states are absent, if possible, why not treat each complete end-to-end execution as an atomic state? By iteratively exploring feasible solutions through parallel executions, we first collect a batch of solutions, and then determine the optimal one, as shown by the program agent in Figure 1. This idea inspires our node outcome-driven reflection system specifically designed for multi-tool interaction in real-world environments. Our

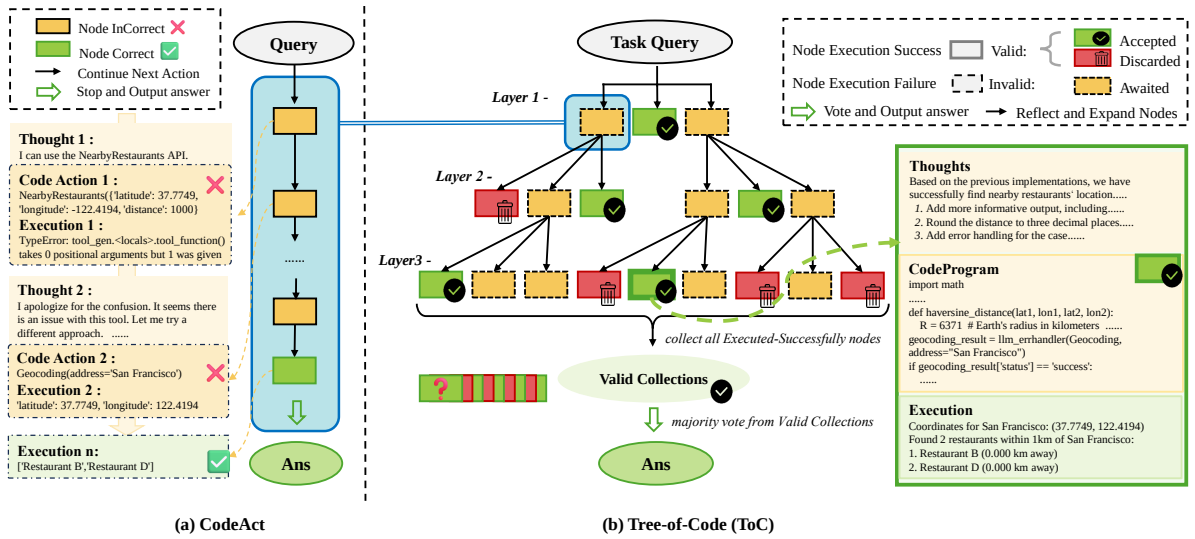

Figure 2: An Overview of **CodeAct** and **ToC**. (a) CodeAct regards code as action with step-by-step reasoning. (b) ToC applies execution-based reflection in the tree structure, where each node (CodeProgram) generates end-to-end code with global planning as its thoughts. At each layer, nodes are executed in parallel; if executed successfully, they are collected for voting. Note that the process supervision relies solely on the node's execution success or failure, rather than on the specific content executed (whether correct or incorrect), which would require pre-known labels. The query is "Find nearby restaurants within 1km of San Francisco" from API-Bank level-3 dataset.

key contribution is a self-growing framework enabling LLM agents to autonomously interact with code through zero-shot learning without GT supervision, whose implementation details will be subsequently presented.

3 Tree-of-Code Method

Following the design motivation, we need to collect all valid solutions and identify the one closest to the GT. By treating each tree node as a complete task-level solution and exploring different nodes for breadth while deepening through iterative refinement, we propose ToC (Tree-of-Code), an execution-based, self-growing, and self-filtering tree for handling real-world complex tasks.

3.1 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 2) , modeling the reflection reasoning process of LLMs when expanding the nodes. The overview of ToC and how it works is illustrated in Figure 2. 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.

3.2 Tree Node Generation

Unlike tool agents like CodeAct, which treat each intermediate action and environmental feedback as a step, each node in our ToC represents a complete task, effectively increasing the granularity of task handling at each layer.

In other words, a single tree node (one turn) is equivalent to multiple turns of CodeAct, with both being directly comparable and serving the same purpose (final response), significantly improving efficiency. We refer to this end-to-end code reasoning and generation paradigm as CodeProgram. Figure 4 illustrates how it works.

Specifically, the end-to-end code in CodeProgram serves as a bridge, aligning with natural language reasoning and execution outcomes in the environment. Besides, by decoupling the reasoning

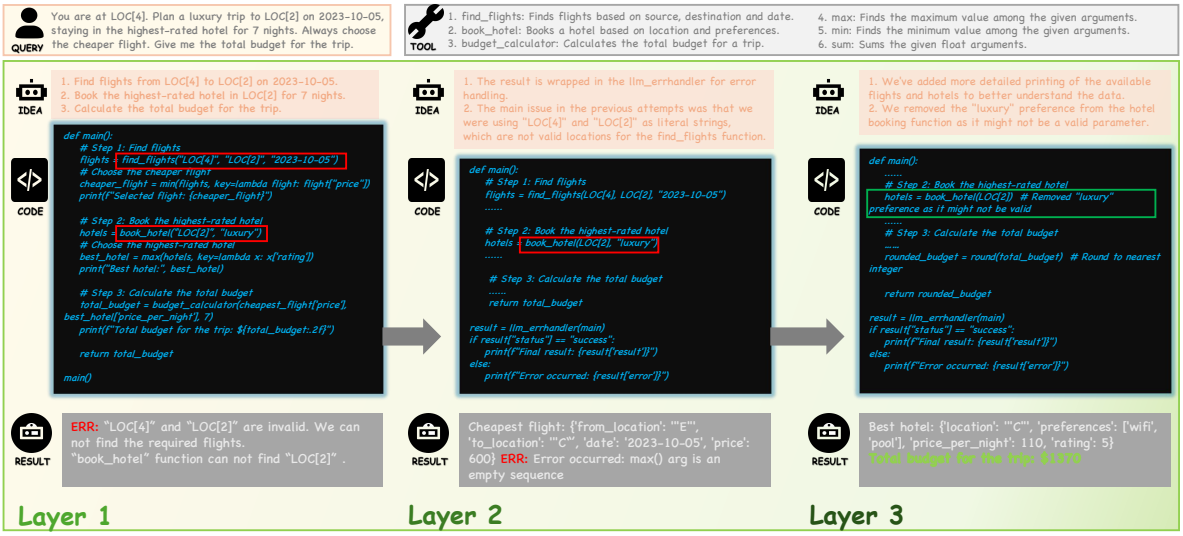

Figure 3: Illustrative example of a branch of ToC. We demonstrated the process of a node expanding into deeper levels. Based on the user query, tool descriptions, and previous execution outcomes, ToC first thinks about how to do it and then writes the end-to-end code. The example is selected from M3ToolEval dataset.

process from code execution, we achieve flexibility while ensuring consistency.

3.2.1 Code as Reasoning

On the one hand, CodeProgram leverages the concept of "code-as-reasoning" to generate code, where the process of writing code itself mirrors the reasoning process.

On the other hand, global reasoning is essential for guiding CodeProgram's complete code generation in a single end-to-end flow. This approach enables the seamless integration of various reasoning techniques 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., 2024), in-context learning (Kojima et al., 2022), self-reflection (Zhang et al., 2024), and System2 reasoning (Frankish, 2010; OpenAI, 2024b). Additionally, longer chains of thought have consistently been shown to enhance task performance (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

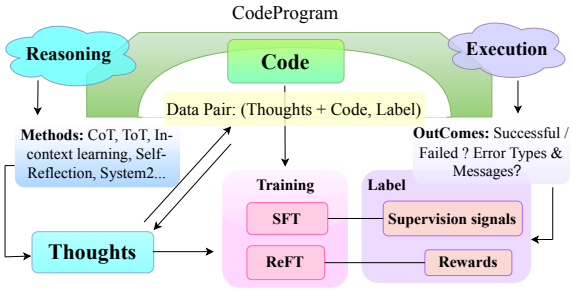

Figure 4: Illustration of the CodeProgram in ToC.

shown in Appendix A.

3.2.2 Two Helper Tools

CodeProgram struggles with environmental exploration when LLMs must rely on tool outputs to determine the next steps. For instance, in web browsing tasks, the next action can only be decided after viewing the page content, and a final summary answer can only be provided after considering all tool outputs. Thus, to maintain end-to-end flow, we introduce two functions: a general **res_handler**, which defines a prompt to generate results that meet the prompt requirements for final summarization, and a specific **next_action** for web tasks, which decides the next action from a given set of possible browsing actions based on the page content, visited URLs, and task query. Their tool descriptions and functions are shown in Appendix B.

They help better understand the semantic relationships between tools, ensuring a smooth, cohesive sequence of tool calls during code generation.

In the Appendix B.3, we also provide an example demonstrating how these helper tools work.

3.2.3 Execution Outcome as Process Label

The code solution is task-level, and its execution outcome is a self-provided annotation that can be directly used as labels. Note that we focus solely on task execution success, using a simple true/false label to filter feasible solutions and approximate more effective ones. This label is weak but available, simple, and useful—unlike pre-known GT or correctness judgments.

Benefiting from our end-to-end paradigm (a direct, complete task-level response to a single query), we can select "successfully executed" samples for SFT and use various rich comments (such as specific results or error messages) as rewards for ReFT by repeating the CodeProgram in different settings (i.e., multi-nodes). In this context, the code acts as a verifier. This verification-then-refinement concept also inspires the development of a multi-layer Tree-of-Code (ToC).

Thanks to task-level granularity, the code's execution outcomes align with both the task query and the thought-code output, enabling the generation of valuable data for potential future training.

3.3 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 $M = 3$ 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 one node 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. As long as execution fails, self-reflection continues iteratively, generating next-layer new nodes. The prompt for reflection is shown in Appendix A.2.1.

This also allows the branch to grow into deeper layers, where each node in the trajectory provides process supervision signals based on its outcome.

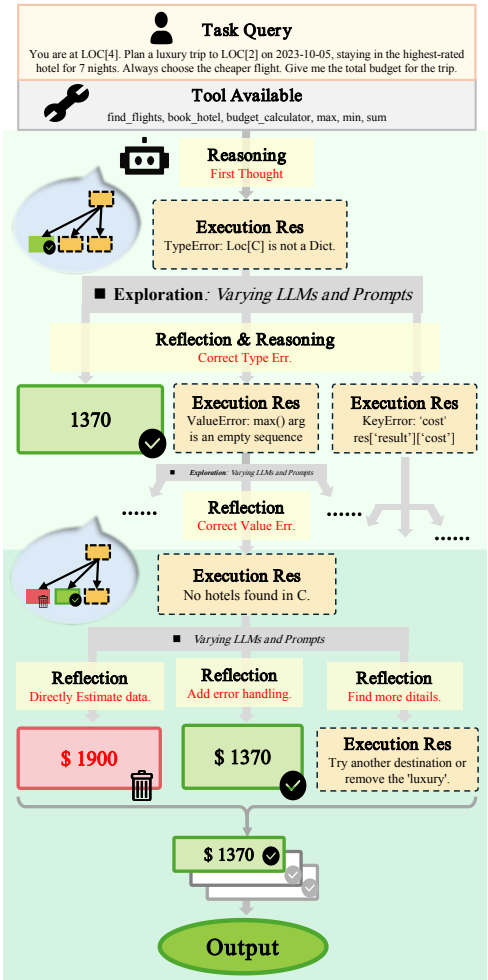

Figure 5: A detailed example illustrating ToC's execution-based reflection and expansion.

Note that the definition of 'turn' is equivalent to that of 'layer'; both terms carry the same meaning. 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 3 shows an example of a branch of ToC while Figure 5 demonstrates execution-based reflection and tree expansion. Additionally, our flexible tree-structured framework allows for the integration of any reflection method for tree expansion.

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 (ie. nodes in ToC).

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 from our list, introduced in Section 4.1, are explored randomly with a consistent temperature setting of 0.1. At the instruction level, prompts are randomly selected from a diverse pool, created through self-evolution and human crafting.

The random exploration mechanisms operate at each node individually, while the prompt pool is created just once for the entire system.

Specifically, we used ten LLMs to generate ten diverse prompts through prompt evolution from the root prompt (see Appendix A). The evolution process ensures the core content remains consistent while promoting orthogonal or divergent expressions. Six distinct prompts were manually selected and the following modifications were then applied: (1) adding detailed usage examples (beyond just printing "Hello world") to three prompts; (2) adjusting the format with line breaks and indentation; (3) randomly rearranging components, including the reflection part, usage examples, role instructions, tool descriptions, and chat history.

3.4 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. Ties are rare in our observations, so we always choose the most frequent answer without special handling.

4 Experiment and Analysis

4.1 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 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. Considering API-Bank only supports JSON format, we make following modifications to adapt it for code interaction: (1) functionalize all API tools, (2) add output examples to each function description (Figure 6). 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.

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 6: Example of the function signature in level-3.

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. We use JSON as the action space. CodeAct (Wang et al., 2024) utilizes a block of code as the LLM agent's action, enabling more efficient multi-turn interactions.

Metrics. The evaluation includes accuracy and

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

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

Mechanism	M3ToolEval			API-Bank level-3		
	Avg Turns	Correct	Output Words	Avg Turns	Correct	Output Words
ReAct	8.2	38.1 %	1.86 k	9.5	8.2 %	1.66 k
CodeAct	7.0	49.4 %	1.91 k	8.9	19.2 %	1.82 k
Tree-of-Code (3-3)	1.7 ↓	67.1 % ↑	0.44 k ↓	2.1 ↓	38.0 % ↑	0.39 k ↓

Table 1: Performance comparison of the baselines and our ToC in terms of averaged turns, output words, and accuracy on two datasets. Note: all numerical results presented in this paper are rounded.

averaged turns. Accuracy represents the percentage of complex tasks that are correctly solved. We consider the LLM-generated code at the same layer, generated in parallel, as one turn. We also record the average number of output words for the API cost evaluation.

4.2 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 LLMs used in this paper. Table 1 shows ToC achieves consistent superior performance (nearly 20% higher) with significantly fewer interaction steps and averaged output words (nearly 1/4), highlighting its efficiency in handling complex tool-use scenarios. Specifically, Figure 7 shows the comparison of ReAct, CodeAct, and ToC on the five tasks in the M3, where ToC achieves near-perfect accuracy on all tasks except the web browsing task.

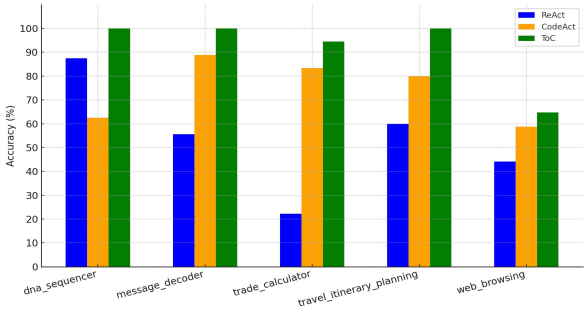

Figure 7: Comparison across five tasks in the M3.

4.2.1 Other multi-turn vs. Our one-turn

Furthermore, we explore the performance of one-layer ToC (1-x) with the fixed model. As a node in ToC, CodeProgram enables the complete solution in a single turn by leveraging code's ability to handle long logic chains. Table 2 shows that, with a significant advantage in the number of turns (one vs. multi-turn: averaged 7.0/8.9), our performance on some models even surpasses multi-turn CodeAct and ReAct, particularly with the Claude series. Compared to CodeProgram, ie. ToC (1-1), the single layer, three nodes ToC (1-3) with random prompts significantly boosts performance. Its average accuracy already surpasses CodeAct, highlighting the effectiveness of prompt randomness.

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.

4.3 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 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).

It seems the "nodes per Layer" contribute more than Layers, likely because our tree structure is designed to enhance exploration. Increasing the number of nodes certainly introduces more diverse prompts and model variations, whereas adding more layers (ie. more turns) mainly accumulates histories without significantly improving decision-making, especially with models that have limited contextual understanding.

Prompt exploration. Ablation results in Table 4 confirm the effectiveness of prompt exploration. By comparing the random model with the fixed model (claude-3-5-sonnet), prompt exploration proves to be more critical in scenarios with lower diversity.

Model	M3ToolEval				API-Bank level-3			
	ReAct	CodeAct	ToC (1-1)	ToC (1-3)	ReAct	CodeAct	ToC (1-1)	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 2: Ablation study of the model exploration. With different fixed models, the detailed performance comparison of ReAct, CodeAct, ablated ToC (1-1) (ie. the CodeProgram node), and ToC (1-3) on the M3ToolEval and API-Bank level-3 datasets is shown. The correctness is reported, with the average number of turns in parentheses.

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.

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: Ablation study of the prompt exploration.

5 Related Work

LLM Code Generation for Complex Tasks. Recent works integrating LLMs with code have largely focused on task completion in programming domains like software development (Qian et al., 2024; Wang et al., 2023), programming assistance (Islam et al., 2024; Wen et al., 2024b), and scientific problems (Chen et al., 2022; Gao et al., 2023; Hong et al., 2024). These studies primarily address pure code generation, where correct task completion only relates to accurate reasoning logic within the code. For example, Chain of Codes (Li et al., 2023a) broadens LLM capabilities by enabling "thinking in code." In contrast, our work addresses real-world, zero-shot online complex tasks that involve multiple tool calls. Only CodeAct (Wang et al., 2024) treats code as a scalable language to call multiple tools, but their approach is limited by an almost one-turn, one-tool, step-by-step mechanism. This results in stalled thinking and accumulated histories, relying heavily on ground-truth supervision for each step, which is incompatible with zero-shot, online settings. In our framework, every node represents a complete solution that can be directly evaluated via execution supervision without requiring additional labels.

Tree-based Code Generation. A recent work, CodeTree (Li et al., 2024b), uses a tree structure to explore the search space of code generation tasks. Unlike our approach, CodeTree focuses on multi-agent searching rather than an end-to-end, self-growing tree. While self-repair trees (Olausson et al., 2023) begin with a specification root node, grow into initial programs through separate feedback and repair stages–often bottlenecked by the model's limited capacity–our approach unifies reasoning (including reflection) and generation in a single cycle at each node, and directly expands the tree with prompt and model exploration. Some contemporaneous works utilizing tree-based search, such as MCTS (Xu et al., 2024a; Yu et al., 2024), require multiple rollouts and significant computational resources, making them unsuitable for online, real-time applications. Unlike these methods, our self-growing tree generates multiple valid solutions and directly selects the one closest to the ground truth through a voting mechanism. Additionally, these studies typically focus on tasks with easier-to-obtain process supervision, whereas our work addresses real-world, complex multi-tool datasets.

6 Conclusion

This paper introduced the Tree-of-Code (ToC) method, which enables self-growing, end-to-end thought-code generation based on successful execution, addressing complex multi-tool online tasks. With efficient model integration and prompt exploration, ToC outperformed baselines on two complex task datasets, improving both efficiency and task-solving performance.

Limitations

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. Even though, in practical industrial applications where a predefined toolset is available, CodeProgram's end-to-end execution remains more efficient for online, zero-shot scenarios, for fewer turns, and for fewer LLM calls.

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.

Vast Potential in Prompt Pool Design

We enhanced the diversity of strategies and the robustness of results in our Tree-of-Code by designing a prompt pool composed of multiple prompts. The introduction of multiple reasoning paths guided by diverse prompts represents a significant innovation. However, our current approach relies primarily on simple prompt evolution and manual adjustments. Future work should focus on more in-depth and systematic research into constructing prompt pools.

Acknowledgments

This work was supported by the National Science and Technology Major Project under Grant 2022ZD0116409, and the National Natural Science Foundation of China under Grant 62301559.

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*.

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*.

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.

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*.

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.

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.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

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.

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

Bin Xu, Yiguan Lin, Yinghao Li, and Yang Gao. 2024a. Sra-mcts: Self-driven reasoning augmentation with monte carlo tree search for code generation. *arXiv e-prints*, pages arXiv–2411.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. 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*.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. 2024. Outcome-refining process supervision for code generation. *arXiv preprint arXiv:2412.15118*.

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

```
res_handler():
    name="res_handler",
    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=res_handler,
    fn_signature='res_handler(prompt: str) -> str')
```

B.1.2 ResHandler Tool Function

```
from some_model_API import llm_playground

def res_handler(prompt):
    result_str = ""
    result = llm_playground(prompt[:20000], stream=False)
    for item in result:
        result_str += item
    return result_str
```

B.2 NextAction for Web Task

B.2.1 NextAction Tool Description

```
from typing import Tuple
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. Return the next action and the viewed whole page
        content.The next possible actions include click_url(URL), go_to_previous_page() and end(), which
         represent clicking a link, and go_to_previous_page() means you should go to previous page to
        find answer, 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 res_handler. click_url(URL), go_to_previous_page() can be directly called, and
        URL should be Clickable url. Note that query should be user's original question and can not be
        rewritten.',
    function=next_action,
    fn_signature="next_action(query: str, current_page_content: str, visited_urls: List[str]) -> Tuple[
        str, str]")
```

B.2.2 NextAction Tool Description

```
from some_model_API import llm_playground

def next_action(query="", current_page_content="", visited_urls=[]):
    visited_urls = [x.replace('\'', '').replace('\"', '') for x in visited_urls]
    visited_urls = list(set(visited_urls))
    whole_page_content = current_page_content
    while True:
        scroll_down_page = scroll_down()
        if scroll_down_page == "[Reached the bottom of the page.]\n":
            break
        else:
            whole_page_content += scroll_down_page
    def extract_clickable_paths(text: str) -> list[str]:
        import re
        pattern = r"Clickable '([^']*)'"
        matches = re.findall(pattern, text)
        return matches
    all_urls = extract_clickable_paths(whole_page_content)

    not_visited = []
    highlight_urls = []

    for v in all_urls:
        if v in visited_urls:
            highlight_urls.append(v)
        else:
            not_visited.append(v)

    if len(highlight_urls) == 0:
        json_str_format = "<thought>your thought of your decision</thought>\n<action>click_url(
            specific_url) or end() or not_found()</action>"
        prompt = f"You are viewing page contents, the content is: \n{whole_page_content}\n You should
            make decision on the next step. given user query {query}, you have the following options,
            please follow the output format. \n1. end(): it means current user query can be answered by
            current page content. \n2. click_url(URL): it means current user query should be checked by
            clicking one of the urls shown on the current page content for more details. specify the
```

```
            detailed url into URL field.\nPlease visit any Clickable urls as many as possible that has
            not been visited. \n3. not_found(): it means that current page does not contain answer for
            current query and all Clickable URLS have been clicked. \nYour output format: {
            json_str_format}\n\nYour Output:\n"
    else:
        visited_url_str = ', '.join(['\'' + x + '\'' for x in highlight_urls])
        json_str_format = "<thought>your thought of your decision</thought>\n<action>click_url(
            specific_url) or end() or not_found()</action>"
        prompt = f"You are viewing page contents, the content is: \n{whole_page_content}\n You should
            make decision on the next step. given user query {query}, you have the following options,
            please follow the output format. \n1. end(): it means current user query can be answered by
            current page content. \n2. click_url(URL): it means current user query should be checked by
            clicking one of the urls shown on the current page content for more details. specify the
            detailed url into URL field.\n3. not_found(): it means that current page does not contain
            answer for current query and all Clickable URLS have been clicked. \nRemember that you have
            visited the url list [{visited_url_str}]. You are not allowed to visit the urls you have
            visited. Please visit any Clickable urls as many as possible that has not been visited.\
            nYour output format: {json_str_format}\n\nYour Output:\n"
    result_str = ""
    result = llm_playground(prompt[:20000])
    for item in result:
        result_str += item

    if not "Clickable" in whole_page_content and not "end()" in result_str:
        return ("go_to_previous_page()", whole_page_content)

    if not "end()" in result_str and len(not_visited) == 0:
        return ("go_to_previous_page()", whole_page_content)

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

B.3 An example to show the performance of two helpers tools

For web-based queries, ResHandler and NextAction, these two helper tools are essential to ensure that our method can directly generate the final result without requiring intermediate feedback.

To further clarify their effectiveness, we provide a qualitative and specific example demonstrating how they function in practice.

Below is the full code output for the query: "Find the email of Henry Santiago. Answer in the format of 'xxx@xxx.xxx'." from typing import List, Tuple

```
def find_email(query: str, current_page_content: str, visited_urls: List[str], max_depth: int = 30) ->
    str:
    if max_depth <= 0:
        raise Exception("Max search depth reached, email not found.")

    # Check if the email is directly available in the current page content
    if "Henry Santiago" in current_page_content and "@" in current_page_content:
        email = res_handler(f"Extract the email of Henry Santiago from the following text: {
            current_page_content}")
        if email:
            return email.strip()

    # Identify clickable URLs and navigate to them
```

```
    next_step_action, reference_page_content = next_action(query, current_page_content, visited_urls)

    if next_step_action == "end()":
        email = res_handler(f"Extract the email of Henry Santiago from the following text: {
            reference_page_content}")
        if email:
            return email.strip()
        else:
            raise Exception("Email not found in the reference page content.")

    elif next_step_action.startswith("click_url"):
        url = next_step_action.split('(')[1].split(')')[0].strip().strip('"')
        if url not in visited_urls:
            visited_urls.append(url)
            new_page_content = click_url(url)
            return find_email(query, new_page_content, visited_urls, max_depth - 1)

    elif next_step_action == "go_to_previous_page()":
        new_page_content = go_to_previous_page()
        return find_email(query, new_page_content, visited_urls, max_depth - 1)

    else:
        raise Exception("Unknown next step action encountered.")

# Main function to start the search
def main():
    query = "Find the email of Henry Santiago"
    initial_page_content = view()
    visited_urls = []
    try:
        email = find_email(query, initial_page_content, visited_urls)
        print(email)
    except Exception as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Functionality of the Helper Tools

For res_handler:

Example usage:

```
email = res_handler("Extract the email of Henry Santiago from the following text: {
    current_page_content}")
```

We provide the content that the final result depends on (i.e., the return value of other functions in the code) as input to this function. It then passes the information to an LLM to generate the final answer.

For next_action:

Example usage:

```
# Identify clickable URLs and navigate to them
next_step_action, reference_page_content = next_action(query, current_page_content, visited_urls)
```

This function determines the next action based on: the current page content, the original query, and the list of visited URLs.

C Visualization of the Table 2

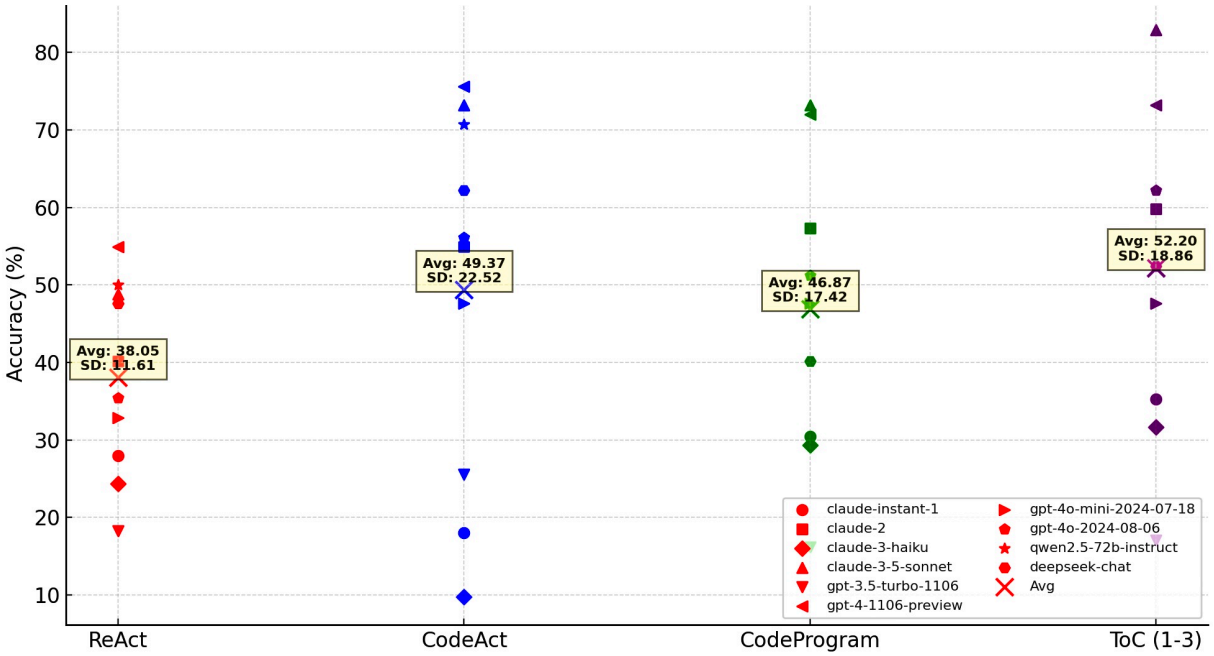

Figure 8: Performance of 10 LLMs on ReAct, CodeAct, CodeProgram, and 1-3 ToC for the M3 dataset is visualized, with average and standard deviation reported.