CodeTool: Enhancing Programmatic Tool Invocation of LLMs via Process Supervision

Yifei Lu[1,2*], **Fanghua Ye**[2*], **Jian Li**[2†], **Qiang Gao**[2], **Cheng Liu**[2], **Haibo Luo**[1], **Nan Du**[2], **Xiaolong Li**[2], **Feiliang Ren**[1†]
[1]School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China
[2]Hunyuan AI Digital Human, Tencent, Shenzhen, China
lyfei1126@gmail.com, {fanghuaye, jackjianli}@tencent.com
renfeiliang@cse.neu.edu.cn

Abstract

Tool invocation significantly enhances the capabilities of Large Language Models (LLMs), yet challenges persist, particularly in complex task scenarios. Current methods, such as instruction-enhanced reasoning and supervised fine-tuning, often result in unnecessarily long reasoning depths and face difficulties in verifying the correctness of intermediate steps. In this paper, we propose **CodeTool**, a novel framework for stepwise code generation that improves LLM tool invocation by leveraging the concise and easily verifiable nature of code. CodeTool incorporates two distinct process rewards: the **On-the-spot Reward**, which provides immediate feedback on the accuracy of each tool invocation, and the **Latent Reward**, which assesses the contribution of each step toward overall task completion. By maximizing the cumulative reward of the On-the-spot and Latend Rewards at each step, LLMs are guided to follow efficient and accurate reasoning paths. Extensive experiments on StableToolBench and RestBench-TMDB demonstrate the superiority of CodeTool over existing approaches. Our implementation is available at LimOkii/CodeTool.

1 Introduction

Tool invocation grants Large Language Models (LLMs) the ability to access external tools (Schick et al., 2023; Shen et al., 2023; Qin et al., 2024), thereby significantly expanding their range of capabilities. Despite the strong potential and ability demonstrated by LLMs in various tasks (Brown et al., 2020; OpenAI et al., 2024), they still encounter challenges when performing tool invocations in complex scenarios.

Early studies (Wei et al., 2022; Yao et al., 2023; Song et al., 2023) have assisted LLMs in better tool invocation by enabling them to think step by step or through instruction enhancement. While this approach is straightforward, it fails to fully leverage the potential of LLMs. More recent studies (Qin et al., 2023; Tang et al., 2023; Patil et al., 2023) have sought to enhance the tool invocation capabilities of LLMs via Supervised Fine-Tuning (SFT). However, training models on static trajectories of successful executions through text generation constrains their adaptability to novel tasks and environments. Besides, these existing studies primarily focus on tool invocation in Text or JSON format, which often leads to prolonged reasoning paths.

Programmatic tool invocation offers a more flexible and generalizable alternative to Text or JSON-based approaches (Wang et al., 2024d,c). By leveraging programming constructs such as loops (*for* or *while*) and arrays, code can efficiently handle request-intensive instructions, thereby reducing the number of interactions required, as illustrated in Figure 1. However, existing code-based approaches still face two key challenges. *First*, relying solely on the built-in functions of programming languages and a limited set of predefined libraries (Wang et al., 2024c) restricts the quantity and scope of tools that can be invoked. *Second*, generating complete code in a single pass (Shi et al., 2024), albeit increasing the number of accessible tools, lacks supervision over intermediate steps, making it difficult to detect and correct errors in complex scenarios.

Supervising the correctness of intermediate steps (i.e., process supervision) during an LLM's reasoning process has been shown to improve the final accuracy of challenging tasks (Uesato et al., 2022; Lightman et al., 2023; Wang et al., 2024b). This approach, however, typically requires large-scale annotations of process data. In addition, the supervision signal tends to direct the model toward plausible correct answers rather than ensuring absolute correctness (Cui et al., 2025). A recent study by Yu et al. (2024) attempts to address these challenges by incorporating process rewards within a

[1]Work done during an internship at Tencent Hunyuan.
[*] Equal Contribution.
[†] Corresponding Author.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 18287–18304

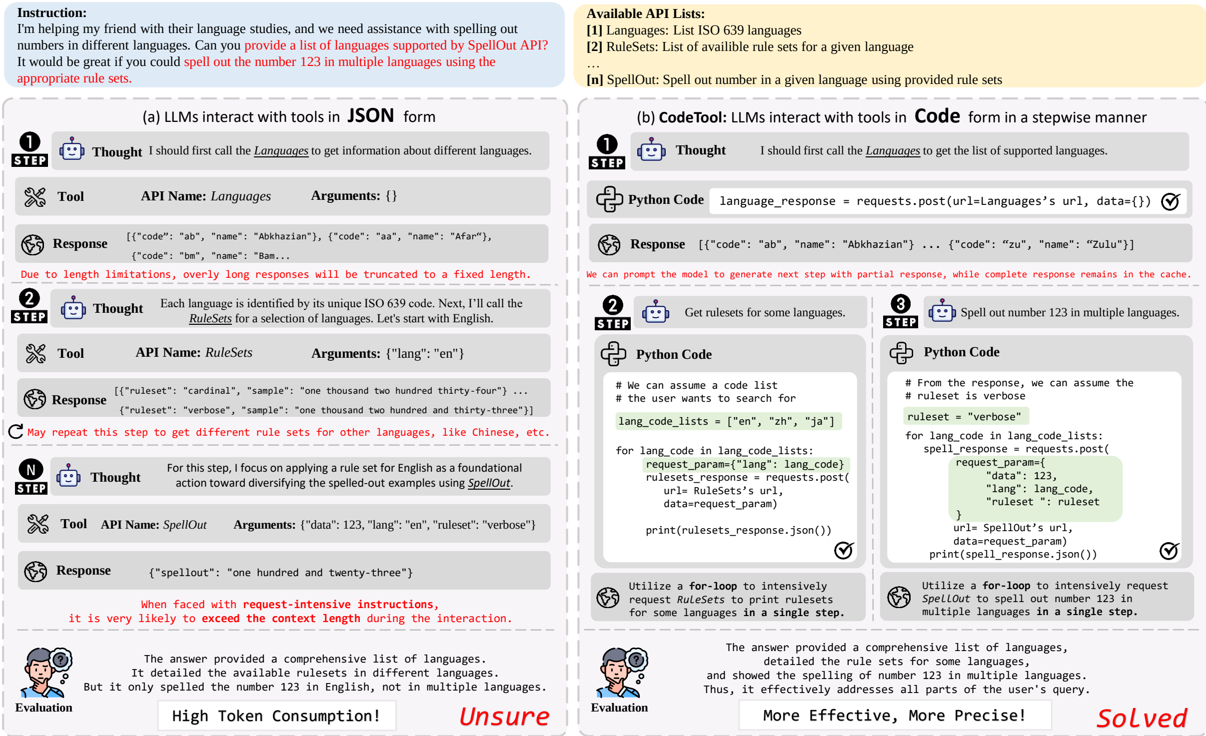

Figure 1: Comparison of tool invocation in JSON format and our proposed stepwise code generation framework: (a) JSON-based invocation is token-heavy when handling request-intensive tasks, and is prone to truncation, risking the loss of key information. (b) The stepwise code generation framework uses loops to handle request-intensive tasks efficiently, with stepwise supervision ensuring correctness of intermediate steps.

reinforcement learning framework to enhance tool invocation. While this method holds promise, its rewards are simply generated by strong LLMs, raising concerns about its objectivity and reliability.

In this work, we propose **CodeTool**, a novel stepwise code generation framework designed to enhance tool invocation of LLMs. CodeTool introduces two distinct process rewards during inference: the **On-the-spot Reward** and the **Latent Reward**. The On-the-spot Reward leverages the inherently verifiable nature of code to provide immediate feedback on the correctness of each tool invocation, ensuring precise execution at every step. In contrast, the Latent Reward, assigned by a trained Process Reward Model (PRM), evaluates the potential contribution of each step towards the overall task completion. At each step, LLMs are guided to follow the reasoning direction that maximizes the cumulative reward of the On-the-spot and Latent Rewards, as illustrated in Figure 2. This dual-reward mechanism overcomes key challenges in current programmatic approaches, particularly the lack of supervision over intermediate reasoning steps. Moreover, the On-the-spot Reward, which is grounded in the executability of the generated code, ensures objective and highly reliable feedback, as demonstrated by our experiments in Section 5.

The contributions of this work are as follows:

- We propose CodeTool, a stepwise code generation framework that leverages process supervision to enhance the capabilities of LLMs in tool invocation.
- We design two types of process rewards–*On-the-spot Reward* and *Latent Reward*–to provide high-quality process supervision, considering both immediate feedback and long-term potential.
- We conduct extensive experiments on StableToolBench (Guo et al., 2024) and RestBench-TMDB (Song et al., 2023), confirming the superiority of CodeTool over existing methods.

2 Related Work

2.1 Tool Invocation With LLMs

Previous studies (Wei et al., 2022; Yao et al., 2023; Song et al., 2023) have investigated enabling LLMs to interact with various tools, such as search engines, calculators, translation software, and third-party API services, to facilitate tool utilization. Most of these approaches rely on prompt engineering to enhance the reasoning capabilities of LLMs during inference or to design prompts tailored to specific modules and tools. Subsequent research (Qin et al., 2023; Tang et al., 2023) has focused on fine-tuning open-source LLMs to equip them with the ability to invoke tools. Recognizing that tool invocation in complex scenarios often requires multi-step reasoning, some studies (Chen et al., 2024; Wang et al., 2024a) have shifted attention toward how LLMs can learn to use tools effectively from error-prone calls. Additionally, Yu et al. (2024) has explored the integration of reward mechanisms in the intermediate decision-making process of LLMs, employing reinforcement learning techniques to improve tool invocation efficiency and outcomes.

2.2 Programmatic Tool Invocation

LLMs typically generate action units in pre-defined formats (e.g., JSON or Text) to interact with external tools. In contrast, programmatic tool invocation offers an alternative mode of interaction. Recent studies have highlighted the potential of incorporating programming to enhance the planning and reasoning capabilities of LLMs, with the feasibility of code-based reasoning particularly demonstrated in complex numerical reasoning tasks (Chen et al., 2023; Gao et al., 2023). Within the context of tool invocation, code blocks can be considered as action units for requesting or executing specific tools. For instance, Wang et al. (2024c) and Shi et al. (2024) have investigated how LLMs can generate complete code to invoke Python's built-in functions or access third-party API services, thereby addressing intricate user instructions. However, these approaches often neglect the significant impact that the accuracy of intermediate steps can have on the final outcome. Notably, code data has been integrated into LLM pretraining (Rozière et al., 2024; Luo et al., 2023; Hui et al., 2024; DeepSeek-AI et al., 2024), resulting in models that demonstrate advanced proficiency in structured programming, thus facilitating the cost-effective adoption of programmatic tool invocation.

2.3 Process Supervision Methods

While LLMs exhibit impressive capabilities across a wide range of tasks, they continue to encounter difficulties in reasoning through complex problems. Lightman et al. (2023) has shown that supervising the correctness of intermediate steps in reasoning tasks can significantly improve the likelihood of LLMs producing accurate final answers. Wang et al. (2024b) and Luo et al. (2024) have proposed automated approaches for constructing intermediate process data. However, their reward designs remain relatively simplistic, focusing solely on the potential of a given step to lead to a correct final answer, while neglecting the correctness of the step itself. Yu et al. (2024) has extended the reward framework to intermediate steps during tool invocation, but the acquisition of process rewards heavily relies on GPT-based annotations. In this work, we seek to fully automate the construction of a performant process reward system to improve programmatic tool invocation of LLMs.

3 Methodology

In this section, we first propose *CodeTool*, a stepwise code generation framework to effectively address the challenges of tool invocation in complex scenarios. Subsequently, we design a process reward system to evaluate each decision-making step during tool invocation. Finally, we train a PRM on fully automated process data and rewards. The architecture of CodeTool is illustrated in Figure 2.

3.1 Preliminaries

Addressing real-world user queries with the help of external tools can be conceptualized as a stepwise planning and reasoning process. Formally, let $\mathcal{M}$ represent an LLM with access to a set of real-world tools $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$, where each tool t_i is associated with a logging protocol $d_i \in \mathcal{D} = \{d_1, d_2, \ldots, d_{|\mathcal{D}|}\}$, which provides meta-information such as the tool's description and the parameters required to make requests.

The goal of CodeTool is to iteratively write Python code $\mathcal{C}_t$ at each step to select the appropriate tool and issue requests with the correct parameters to obtain responses, ultimately deriving the final answer. Compared to previous work (Shi et al., 2024) that solves user queries by directly generating complete code based on a well-structured dataset with detailed input-output schemas (which are hard to obtain in practice), the stepwise code generation method in CodeTool reduces reliance on such datasets. It allows the model to generate code to invoke a tool at each step, with the flexibility to utilize partial responses from previous invocations in subsequent steps. Such a sequential approach enables the model to comprehend the content and structure of intermediate outputs, thus aiding in parsing the responses and determining the next appropriate tool to invoke. Moreover, dif-

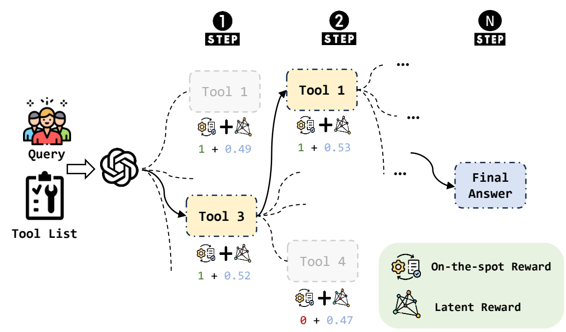

Figure 2: The architecture of **CodeTool**, a stepwise code generation framework guided by two types of process rewards during inference. At each inference step, LLMs follow the reasoning path that maximizes the cumulative rewards of ***On-the-spot Reward*** and ***Latent Reward***.

ferent from text-based responses, which are often truncated to a fixed length when they are overly long (Qin et al., 2023), the results from code-based tool invocation are complete. The critical data is preserved in the cache, remaining intact and unaffected by truncation, thereby ensuring the integrity of the tool responses throughout the entire process.

3.2 CodeTool: Stepwise Code Generation

Given a user query q, we first provide the LLM with the documented protocol $d_i \in \mathcal{D}$ for each tool t_i in the candidate toolset $\mathcal{T}$. Each protocol d_i contains meta-information, including a description of the purpose of the tool, the URL to be requested, and the argument requirements for invoking it. Then, we instruct the LLM $\mathcal{M}$ to generate executable programs step by step to utilize multiple tools and ultimately solve the query q.

Formally, this process can be formulated as:

$$\mathcal{C}_t = \mathcal{M}(q, \mathcal{T}, \mathcal{D}, I_c, r_{t-1}), \quad r_0 = \emptyset, \qquad (1)$$

where I_c indicates a concise instruction for stepwise code generation (refer to Appendix A for details). The intermediate result r_t of the t-th step is obtained by executing the generated program $\mathcal{C}_t$ through a code interpreter, which is formulated as:

$$r_t = \textsc{Execute}(\mathcal{C}_t). \qquad (2)$$

It is worth noting that there is initially no response from the tool, meaning that $r_0 = \emptyset$. The subsequent program generation operations will parse specific field information based on the responses from the current tool and then invoke the next tool, continuing this process until the final answer is obtained.

3.3 Process Reward Supervision

In this part, we detail the integration of process supervision into the stepwise code generation framework to enhance its performance. As illustrated in Figure 2, at each step, we first sample multiple candidate actions for the next step and then select the optimal action based on a calculated process reward to proceed to the next step. This approach facilitates the exploration of a broader range of options, thereby increasing the likelihood of selecting the appropriate tool and generating accurate code for each step. We consider two types of process rewards: *On-the-spot Reward* and *Latent Reward*.

On-the-spot Reward On-the-spot Reward evaluates whether the model has provided correct and executable code at the current step, including verifying whether a valid request body has been given within the candidate toolset or not. It can be obtained without any external supervision, as it only requires the automatic execution of the generated code $\mathcal{C}_t$ using a Python interpreter $\textsc{Execute}$.

On-the-spot Reward is defined as:

$$R_{\text{spot},t} = \begin{cases} 1, & \text{if } \textsc{Execute}(\mathcal{C}_t) \text{ is successful;} \\ 0, & \text{otherwise.} \end{cases} \qquad (3)$$

This reward serves as a necessary condition to ensure that the model's reasoning moves along a potentially correct path. It provides immediate feedback on whether the generated code is executable and correct at each step. However, it only measures the correctness of the current step, ignoring whether the current step contributes to completing the user's query. Therefore, we also need to introduce another Latent Reward to assess the potential value of the current step in completing the query.

Latent Reward Latent Reward evaluates the potential value of the current step in helping the model successfully complete the task, considering factors such as whether redundant tools are invoked, leading to an unnecessarily long reasoning path, and whether the model selects an incorrect tool.

Drawing on previous approaches (Lightman et al., 2023; Wang et al., 2024b; Luo et al., 2024), we employ the Monte Carlo Tree Search (MCTS) algorithm to estimate the Latent Reward. Specifically, from each reasoning step, we expand the search tree based on a random sampling of the search space, resulting in multiple executed paths or rollouts. Then, the Latent Reward is defined as:

$$LR(q, s_{1:t}) = \frac{\Delta_{correct}}{\Delta_{total}}, \quad (4)$$

where $\Delta_{correct}$ and Δ_{total} denote the number of paths that are correctly executed with final answer labeled as "`Solved`" and the total number of executed paths, respectively, from the t-th step to its leaf nodes, while $s_{1:t}$ denotes the reasoning sequence from the first step to step t.

However, in complex scenarios, tool invocation may suffer from issues such as repetitive calls to a specific tool, particularly when the tool is no longer accessible, and long and redundant tool invocations. To mitigate unnecessary resource consumption and inefficiency, it is crucial to prioritize shorter, yet correct, tool invocation paths. Therefore, a penalty mechanism should be implemented to discourage such inefficiencies (Luo et al., 2024). In light of this, the final Latent Reward is defined as:

$$R_{\text{latent},t}(q, s_{1:t}) = \alpha^{1-LR(q,s_{1:t})} \cdot \beta^{\frac{\tau}{L}}, \quad (5)$$

where $\alpha, \beta \in (0, 1]$ and $L > 0$ are constant hyperparameters, while τ denotes the average number of steps of executed paths starting from the t-th step. This formula comprehensively takes into account both the initial Latent Reward and the penalty for reasoning with overly long or redundant steps.

Based on the above two types of rewards, we obtain the cumulative reward of the t-th step:

$$R_{\text{total},t}(q, s_{1:t}) = R_{\text{spot},t} + R_{\text{latent},t}(q, s_{1:t}). \quad (6)$$

Then, we select the candidate action with the highest cumulative reward to move to the next step.

3.4 Process Latent Reward Model Training

During inference, while the On-the-spot Reward can be readily obtained through a code interpreter, estimating the Latent Reward using the MCTS algorithm presents significant challenges, including: (1) the time-intensive and costly nature of collecting multiple rollouts, and (2) the difficulty in evaluating whether a rollout successfully addresses the task in the absence of ground-truth data, a common scenario in practice. To address these challenges, we propose training a process Latent Reward model to estimate the Latent Reward during inference, significantly enhancing efficiency.

For model training, we select user queries from the ToolBench (Qin et al., 2023) training set that remain solvable, meaning that the tools or APIs required to solve these queries are still callable, and use them to construct the intermediate process data. To balance computational resource usage and inference time, we employ a depth-first search algorithm to construct an action search space resembling a binary tree for each user query. Specifically, during the code generation for each tool invocation step, we perform sampling twice. Subsequently, at each hierarchical level of the action search space, we collect process data with varying Latent Reward values, which are regarded as a key indication for comparing the levels of potential. Ultimately, we leverage the data to train the PRM. The entire process is fully automated, ensuring efficiency and requiring no human intervention. Further details can be found in the Appendix E

	I1-I	I1-C	I1-T	I2-I	I2-C	I3-I	Total
Full	200	200	200	200	200	100	1100
Solvable	163	153	158	106	124	61	765
Filtered	131	122	118	92	100	17	580

Table 1: Statistics regarding the original full and solvable tasks provided by ToolBench and StableToolBench, along with the statistics on the data after our further screening. C, I, T stands for the 'Category', 'Instruction' and 'Tool' subgroup of the test set, respectively.

4 Experimental Setup

4.1 Datasets

ToolBench (Qin et al., 2023) is a widely adopted benchmark in the field of tool invocation, designed to evaluate the ability of models to solve user instructions in complex real-world scenarios. As mentioned above, for the training of the process Latent Reward model, we select tools and APIs that are still accessible from the ToolBench training set and automatically construct intermediate process data for tool invocation. To evaluate the performance of CodeTool, we utilize the ToolBench test set. However, the original ToolBench test set presents challenges in reproducibility due to the inaccessibility of many tools and APIs. To address this issue, we employ StableToolBench (Guo et al., 2024), a stable version of the Rapid-API access system derived from ToolBench using an API caching mechanism. StableToolBench comprises 765 solvable tasks distributed across six subsets, with each varying in tool category and instruction complexity, ranging from single-tool to multi-tool instructions. The detailed statistics are presented in Table 1.

4.2 Adaptations to StableToolBench

API Response Handling In StableToolBench, API responses are provided in Text format, and excessively long responses are truncated to a fixed length. In order to facilitate the processing of these responses, we convert them to JSON format upon receipt, ensuring that the data can be effectively utilized by the code. For responses that are truncated, we employ GPT-4 (`GPT-4-Turbo-2024-04-09`) to reconstruct them into a complete JSON format.

Test Set Filtering In our experiments, we find that even when the model provides the same API and request parameters as those used in the StableToolBench experiments, some requests fail due to the absence of cache hits. To ensure a fair comparison of experimental results, we filter out entries from the StableToolBench test set that are non-reproducible. The specific criteria for exclusion are detailed in Appendix C.

4.3 Baselines and Evaluation Metric

We conduct experiments using both open-source and closed-source LLMs. For open-source LLMs, we primarily compare our method with the well-established baseline, **ToolLLaMA** (Qin et al., 2023), which is fine-tuned from LLaMA on successful tool execution chains. Additionally, we include two reinforcement learning-based baselines, **TP-LLaMA** (Chen et al., 2024) and **StepTool** (Yu et al., 2024), which utilize direct preference optimization (Rafailov et al., 2024) and proximal policy optimization (Schulman et al., 2017), respectively. To take full advantage of the code generation capability of LLMs and ensure a fair comparison at the same time, we employ **CodeLlama-7B**[1], **Qwen2.5-7B-Instruct**[2] and **Qwen2.5-Coder-7B-Instruct**[3] as the code generation models without any additional fine-tuning for CodeTool. All models share the same number of parameters (i.e., 7B) as the baselines. For closed-source LLMs, we adopt **GPT-3.5-Turbo-0613** and **GPT-4-Turbo-Preview**, each representing different performance levels and capabilities within the GPT series. Following Qin et al. (2023); Yu et al. (2024), we employ two inference strategies for baselines: Chain of Thought (**CoT**) (Wei et al., 2022) and Depth-First Search Decision Tree (**DFSDT**) (Qin et al., 2023).

Following StableToolBench, we utilize the Solvable Pass Rate (**SoPR**) as the evaluation metric. Specifically, GPT-4 (`gpt-4-turbo-2024-04-09`) is leveraged as the evaluator to categorize the answers into "`Solved`", "`Unsure`", or "`Unsolved`", with corresponding scores of 1, 0.5, and 0, respectively, contributing to the overall SoPR calculation. However, evaluation experiments on StableToolBench have shown that different models exhibit varying preferences regarding the degree to which the final answer resolves the query, leading to unstable evaluation results. To address this, we test the evaluation script provided by StableToolBench and introduce clearer criteria for assessing these three categories (detailed in Appendix B), significantly enhancing the stability of model evaluations.

4.4 Training Settings

We only need to train the process Latent Reward model. In our experiments, we train such a model using Qwen2.5-7B-Instruct. To avoid disrupting the native structure of the LLM, we adopt a generative PRM training method. Specifically, we designate two special tokens to represent the "`more potential`" and "`less potential`" labels based on Latent Reward values, and then fully reuse the training method of SFT. We train the Qwen2.5-7B-Instruct model on the collected process data for 2 epochs with a learning rate of 1e-6. More training details can be found in the Appendix E

5 Experimental Results and Analyses

5.1 Main Results

Table 2 presents the performance comparison of CodeTool with baselines on both open-source and closed-source LLMs. Below are some key observations drawn from the results: (**1**) For open-source LLMs, when Qwen2.5-7B-Instruct is used as the code generation model, CodeTool consistently outperforms both the CoT and DFSDT strategies. (**2**) For open-source Coder LLMs, while CodeLlama-7B exhibits lower performance compared to ToolLLaMA when the DFSDT strategy is adopted, the more capable Qwen2.5-Coder-7B-Instruct, which possesses stronger coding and instruction-following abilities, achieves significantly higher SoPR scores on the test set, demonstrating the effectiveness of the CodeTool framework. Although Qwen2.5-7B-Instruct and Qwen2.5-Coder-

[1] https://huggingface.co/codellama/CodeLlama-7b-hf

[2] https://huggingface.co/Qwen/Qwen2.5-7B-Instruct

[3] https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct

Models	Strategy	Invocation Form	I1 Ins.	I1 Cat.	I1 Tool.	I2 Ins.	I2 Cat.	I3 Ins.	Avg
Open-Source LLMs									
ToolLLaMA-v2 (7B)	CoT	JSON	32.06	37.70	36.01	29.35	38.00	29.41	33.39
	DFSDT		46.56	55.74	51.27	49.46	60.50	55.89	53.24
TP-LLaMA (7B)	DFSDT	JSON	27.00	48.00	37.00	35.00	36.00	35.00	36.00
StepTool (7B)	DFSDT	JSON	44.27	47.54	42.80	42.39	43.00	44.12	44.02
Qwen2.5-Instruct-7B	CoT	JSON	56.11	58.20	51.26	54.89	61.00	44.12	54.26
	DFSDT	JSON	61.07	63.11	62.18	54.89	66.00	55.88	60.52
	CodeTool	Code	**63.74**	<u>68.44</u>	**65.68**	<u>55.98</u>	**72.50**	<u>58.82</u>	<u>64.19</u>
CodeLlama-7B	**CodeTool**	Code	45.80	46.72	42.37	45.65	47.50	35.29	43.89
Qwen2.5-Coder-7B			<u>62.59</u>	**74.59**	<u>63.98</u>	**67.93**	<u>70.00</u>	**79.41**	**69.75**
Closed-Source LLMs									
GPT-3.5-Turbo-0613	CoT	JSON	47.71	45.77	54.66	38.59	43.00	44.12	45.64
	DFSDT		59.54	62.29	66.10	52.72	62.50	70.58	62.34
	CodeTool	Code	59.92	59.84	53.39	40.21	53.00	55.88	53.71
GPT-4-Turbo-Preview	CoT	JSON	51.15	64.34	55.84	55.43	58.50	70.59	59.32
	DFSDT		59.16	61.06	47.48	62.50	65.50	76.47	62.03
	CodeTool	Code	**62.97**	**76.22**	**69.49**	**65.76**	**69.50**	**82.35**	**71.05**

Table 2: Performance comparison in terms of SoPR between CodeTool and baselines. The best performances in open-source LLMs are highlighted in **bold**, while suboptimal ones are marked with <u>underline</u>. We reproduce StepTool on ToolLLaMA-v2 using the released code. All results are assessed on the filtered StableToolBench test set using the improved SoPR evaluation prompt (refer to Section 4.3 for details about the evaluation prompt).

7B-Instruct perform comparably across different test subsets, with each showing advantages in certain areas, Qwen2.5-Coder-7B-Instruct ultimately offers more consistent and overall better performance. (**3**) For closed-source LLMs, both GPT-3.5-Turbo-0613 and GPT-4-Turbo-Preview demonstrate comparable performance on the SoPR metric when employing the DFSDT strategy with JSON-format-based tool invocations. Notably, GPT-4-Turbo-Preview, with its superior code generation capabilities, further elevates the SoPR metric when paired with CodeTool, enhancing the performance of Qwen2.5-Coder-7B-Instruct by 1.86%. (**4**) The results on both open-source and closed-source LLMs suggest that the more advanced the model's coding capabilities, the better its tool invocation performance in terms of the SoPR metric when paired with our CodeTool.

5.2 Generalization Performance on RestBench-TMDB

To evaluate the generalization capability of the CodeTool framework, we conduct additional experiments on another widely recognized benchmark, RestBench-TMDB (Song et al., 2023), which contains 100 tasks involving 54 tools designed for movie-related scenarios. Notably, we do not retrain the process Latent Reward model for this benchmark; instead, we utilize the model previously trained for StableToolBench directly. We adopt the Success Rate and Correct Path Rate metrics provided by RestBench-TMDB for evaluation. Success Rate relies on human assessment to ascertain whether the model's output successfully fulfills the user query, while Correct Path Rate measures the proportion of ground truth tools in model-generated tool invocations. The results, as shown in Table 3, indicate that CodeTool achieves the best performance, with a Success Rate of 92% and a Pass Rate of 91.15%, surpassing ATC by 3.37% and 7.60%, respectively, which involves writing complete Python code to solve user queries.

Methods	RestBench-TMDB	
	Success (%)	Path Rate (%)
GPT-3.5-Turbo-0613		
ReAct (Yao et al., 2023)	61.00	77.13
RestGPT (Song et al., 2023)	65.00	77.49
CodeAct (Wang et al., 2024c)	63.00	80.91
ToolLLaMA (Qin et al., 2023)	72.00	78.29
ATC (Shi et al., 2024)	89.00	84.71
CodeTool	**92.00**	**91.15**

Table 3: Performance comparison in terms of success rate and correct path rate on RestBench-TMDB.

5.3 Analyses

Ablation Study I: Impact of Two Types of Process Rewards on SoPR To evaluate the contribution of the two types of process rewards in CodeTool, we conduct an ablation study by con-

Methods	Solvable Pass Rate (SoPR, %)						
	I1 Ins.	I1 Cat.	I1 Tool.	I2 Ins.	I2 Cat.	I3 Ins.	Avg
Qwen2.5-Coder-7B + CodeTool	62.59	**74.59**	**63.98**	**67.93**	70.00	**79.41**	**69.75**
- w/o On-the-spot Reward	61.45	72.95	61.86	60.32	68.50	70.83	65.99
- w/o Latent Reward	**67.94**	68.03	62.71	62.50	**73.00**	58.33	65.41

Table 4: Ablation study on the two types of process rewards within CodeTool from the perspective of SoPR.

Methods	Successful Code Execution Proportion (SCEP, %)						
	I1 Ins.	I1 Cat.	I1 Tool.	I2 Ins.	I2 Cat.	I3 Ins.	Avg
Qwen2.5-Coder-7B + CodeTool	**85.33**	**84.62**	**82.47**	**87.14**	**86.61**	**95.00**	**86.86**
- w/o On-the-spot Reward	66.45	71.03	69.20	71.10	58.63	80.36	69.46
- w/o Latent Reward	84.46	84.17	80.93	84.48	86.34	91.67	85.34

Table 5: Ablation study on the two types of process rewards within CodeTool from the perspective of SCEP.

structing two variants: - *w/o On-the-spot Reward*, where the On-the-spot Reward is set to 0 regardless of the success of code execution at the current step, and - *w/o Latent Reward*, where the Latent Reward is set to 0, meaning that the new code generation step only aims for a direction of correct code execution. If multiple directions for correct execution exist at the current step, one is randomly selected. As shown in Table 4, the removal of On-the-spot Reward and Latent Reward results in a decrease in the average SoPR to 65.99 and 65.41, respectively. These findings suggest that both types of process rewards positively impact programmatic tool invocation within CodeTool. However, removing Latent Reward leads to an increase in SoPR on *I1_Ins* and *I2_Cat*. The reason may be that the queries in *I1_Ins* and *I2_Cat* are relatively simple compared to other subsets. Consequently, the guidance provided by the On-the-spot Reward is sufficient to address these queries, while the Latent Reward from the PRM may introduce additional biases, resulting in weaker performance. Nevertheless, the average SoPR for the - *w/o Latent Reward* setting is 65.41, which is much lower than the 69.75 achieved when Latent Reward is included. Therefore, it can be concluded that Latent Reward retains significant importance within CodeTool.

Ablation Study II: Impact of Two Types of Process Rewards on Successful Code Execution Proportion To further investigate the role of the process reward system within the CodeTool framework, we conduct another ablation study to evaluate the impact of the two types of process rewards on the Successful Code Execution Proportion (**SCEP**). This metric assesses the overall correctness of step-wise code execution and is defined as follows:

$$SCEP = \frac{\sum_{i=1}^{6}\sum_{j=1}^{P_i}\sum_{k=1}^{S_{i,j}} C_{i,j,k}}{\sum_{i=1}^{6}\sum_{j=1}^{P_i} S_{i,j}}, \quad (7)$$

where P_i represents the number of queries in subset i (for $i = 1, 2, \ldots, 6$), $S_{i,j}$ denotes the number of coding steps required to solve the query j in subset i, and $C_{i,j,k}$ is an indicator of whether the coding step k in addressing query j (from subset i) is correctly executed, with $C_{i,j,k} = 1$ if the step is correctly executed and $C_{i,j,k} = 0$ otherwise.

As shown in Table 5, removing On-the-spot Reward significantly lowers the average SCEP to 69.46%, indicating its crucial role in guiding the CodeTool framework to generate accurate code. This is expected, as the On-the-spot Reward ensures that the LLM's code generation is directed towards correct execution. Additionally, the removal of the Latent Reward also results in a decrease in SCEP compared to when it is retained. Although the On-the-spot Reward provides immediate guidance, the Latent Reward contributes to long-term accuracy in code generation, highlighting its importance in the overall process.

Efficiency Comparison with JSON-based Tool Invocation To further investigate the differences in reasoning efficiency between code-based and JSON-based methods, particularly in terms of reasoning depth and token consumption, we conduct additional comparative experiments on I2-Category. We use Qwen2.5-7B-Instruct as the code generation model. As shown in Figure 3, in terms of

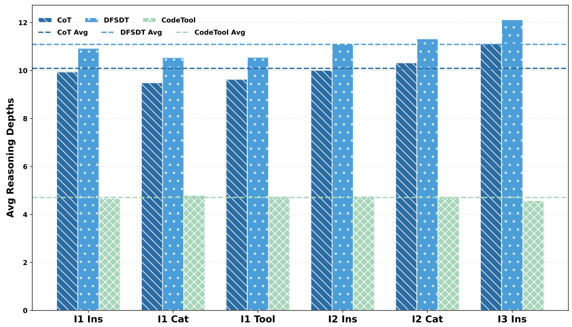

Figure 3: Average reasoning depths of three different strategies across six test subsets in StableToolBench.

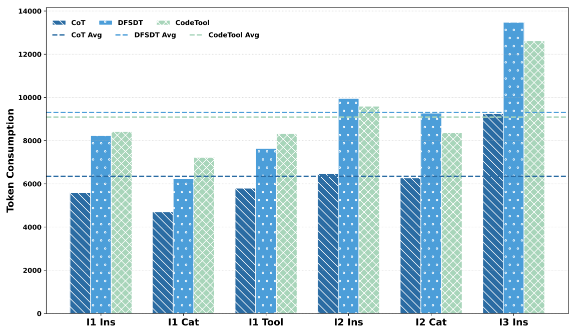

Figure 4: Token consumption of three different strategies across six test subsets in StableToolBench.

average reasoning depth, the CodeTool framework, compared with JSON-based tool invocation methods that employ CoT and DFSDT strategies, enables both tool invocation and response acquisition within a single interaction step, thereby effectively reducing the number of interactions by nearly half. As shown in Figure 4, the CodeTool framework maintains a comparable level of token consumption to the DFSDT strategy, while outperforming both DFSDT and CoT in terms of SoPR, further underscoring the effectiveness of our approach.

Performance with Varying PRM Training Methods and Number of Candidate Actions As outlined in Section 4.4, we train the Generative PRM by designating two special tokens to represent the "`more potential`" or "`less potential`" labels based on Latent Reward values through a generative approach. For comparison, we also train a Pairwise PRM by augmenting the LLM backbone with an additional linear head, enabling it to classify the intermediate process data as either "`more potential`" or "`less potential`". Apart from Qwen2.5-7B-Instruct, we also train LLaMA-3-8B-Instruct (AI@Meta, 2024) as the PRM. As shown in Figure 5, increasing the number of can-

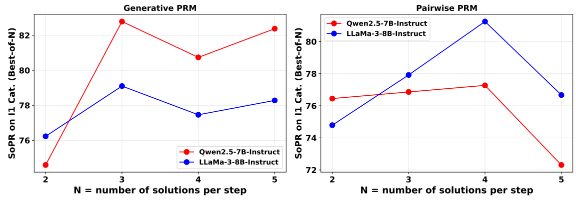

Figure 5: Performance with varying PRM training methods and number of candidate actions on I1 Category.

didate actions from 2 to 3 consistently improves SoPR across both PRM training methods and models. However, further increasing N beyond 3 tends to degrade performance, indicating a trade-off between diversity and quality in candidate selection. Under the Generative PRM setting, Qwen2.5-7B-Instruct consistently outperforms LLaMA-3-8B-Instruct, especially at $N = 3$ and $N = 5$. In contrast, under the Pairwise PRM setting, LLaMA-3-8B-Instruct achieves the highest SoPR at $N = 4$, surpassing Qwen2.5-7B-Instruct. These observations highlight the importance of both the model architecture and the training paradigm in determining the effectiveness of PRM, and suggest that model-specific tuning of N and PRM strategy may be necessary for optimal performance.

6 Conclusion

In this paper, we introduce CodeTool, a stepwise code generation framework based on process supervision. By leveraging code as a naturally verifiable format, we obtain an On-the-spot Reward to reflect step correctness and train a PRM on fully automated process data to assign a Latent Reward, which measures the potential of each step toward overall task completion. At each inference step, LLMs follow the reasoning path with maximal cumulative reward of On-the-spot Reward and Latent Reward. Extensive experiments conducted on StableToolBench and RestBench-TMDB validate the effectiveness of the CodeTool framework.

Limitations

Despite the superiority of the proposed CodeTool framework, its performance is, to some extent, influenced by the code generation capabilities of the underlying LLM. A model with high proficiency in generating accurate and efficient code will naturally enhance the performance of CodeTool. Conversely, models with less advanced coding abilities may not fully exploit the potential of this framework. In addition, previous work shows that the performance of the PRM for Latent Reward is closely tied to the quality of the collected process data, particularly the accuracy of the Latent Reward values (Zhang et al., 2025). Given that we rely on sampling methods to estimate these values and subsequently use them to train the PRM, there is a potential for suboptimal performance if the estimated values are not sufficiently accurate.

Ethics Statement

The research conducted in this paper aims at enhancing programmatic tool invocation of LLMs via process supervision. Throughout the course of this research, we have rigorously adhered to ethical standards to uphold the integrity and validity of our work. All tools (APIs) utilized in this study are sourced from publicly available platforms, ensuring full transparency and reproducibility in our experimental procedures. Moreover, we have taken great care to ensure that our research does not cause harm to individuals or groups, and we have committed to avoiding any forms of deception or misuse of information in the course of our study.

References

AI@Meta. 2024. Llama 3 model card.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. 2024. Advancing tool-augmented large language models: Integrating insights from errors in inference trees.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. 2025. Process reinforcement through implicit rewards.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models.

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

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, 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.

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

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom ... William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report.

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

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2024. Tool learning with foundation models.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face.

Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Zhumin Chen, Suzan Verberne, and Zhaochun Ren. 2024. Chain of tools: Large language model is an automatic multi-tool learner.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world restful apis.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process- and outcome-based feedback.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. 2024a. Llms in the imaginarium: Tool learning through simulated trial and error.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. 2024b. Math-shepherd: Verify and reinforce llms step-by-step without human annotations.

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

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2024d. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

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

Yuanqing Yu, Zhefan Wang, Weizhi Ma, Zhicheng Guo, Jingtao Zhan, Shuai Wang, Chuhan Wu, Zhiqiang Guo, and Min Zhang. 2024. Steptool: A step-grained reinforcement learning framework for tool learning in llms.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. The lessons of developing process reward models in mathematical reasoning.

A Instruction for Stepwise Code Generation

A reference prompt designed to guide LLMs in performing code generation step-by-step is illustrated in Figure 6. This prompt is structured to facilitate incremental reasoning by the model, providing clear instructions.

B Improved SoPR Evaluation Prompt

To enhance the stability of model evaluations, we introduce clearer criteria for assessing "`Solved`", "`Unsolved`", or "`Unsure`" cases for the SoPR metric. The improved evaluation prompt is shown in Table 6.

C StableToolBench Test Set Filtering Rules

Rules for filtering the test set of StableToolBench are shown in Table 7.

D Total Number of APIs Involved in the Filtered Test Set

The total number of APIs involved in the filtered test set is shown in Table 8.

	I1-I	I1-C	I1-T	I2-I	I2-C	I3-I
Task	131	122	119	92	100	17
Candidate API	631	271	371	536	341	31
Relevant API	288	171	199	197	172	19

Table 8: The total number of APIs involved in the filtered test set. C, I, T stand for the Category, Instruction, and Tool subgroup of the test set.

E Details of PRM Training

In the process data collection phase, we draw inspiration from the sampling strategy of MCTS. However, considering the high time and computational cost associated with performing multiple rollouts in MCTS, we adopt a simplified implementation. As described in Section 3.4 of the paper, to balance computational resources and inference efficiency, we employ a Depth-First Search (DFS) algorithm. Specifically, during the code generation phase at each step of tool invocation, we perform two samples for each node's next action, thereby expanding a binary tree-like search space for code generation paths. For each node in the search tree, we first compute its raw Latent Reward using Equation 4, where a key challenge lies in determining whether the current reasoning path successfully solves the user's query. In the scenario of code-based tool invocation, we adopt the practice of Shi et al. (2024) by explicitly instructing the model in the prompt to output a complete *Final Answer* via a print statement at the final step of reasoning. In relatively straightforward tool-invocation scenarios (e.g., numerical queries), the model typically follows this instruction well, accurately providing a complete *Final Answer*. However, in more complex tasks, it is difficult for the model to directly generate a complete answer in the final step. Thus, we implement a more flexible strategy: after each tool invocation, the model immediately prints out the result returned from that request, as specified explicitly in Figure 6. Subsequently, we pair the model's thought at each reasoning step with the corresponding tool response, and combine these pairs with the original user query to prompt the LLM to reorganize them into a coherent natural-language text as the *Final Answer*. After obtaining the *Final Answer*, we apply the same evaluation prompt used in the SoPR experiment (detailed in Table 6). Based on how well the model-generated *Final Answer* addresses the user's original query, LLM classifies each answer into one of three categories: "`Solved`", "`Unsure`", or "`Unsolved`". Accordingly, we automatically assign its final Latent Reward using Equation 5. Leveraging these reward signals, we further collect process data with varying Latent Reward values from every level of the search tree, thereby constructing training samples for the PRM that capture the differences in potential among various paths.

In the training phase of the PRM, to avoid interfering with the structure of the LLM, we follow a generative training strategy similar to Wang et al. (2024b). Specifically, for two child nodes sampled from the same parent node, we assign labels "`more potential`" and "`less potential`" based on their Latent Reward values (e.g., using two special tokens "`yes`" and "`no`" as supervision signals) for supervised fine-tuning.

During inference, to estimate the Latent Reward of the current step, we compute the logits of the "`yes`" and "`no`" tokens output by the model and normalize them using the following formula:

$$\text{Latent Reward} = \frac{\text{logit}_{\text{yes}}}{\text{logit}_{\text{yes}} + \text{logit}_{\text{no}}}. \tag{8}$$

SoPR Evaluation Prompt in StableToolBench
Giving the query and answer, you need to give `answer_status` of the answer by following rules: 1. If the answer is a sorry message or not a positive/straight response for the given query, return `"Unsolved"`. 2. If the answer is a positive/straight response for the given query, you have to further check. 2.1 If the answer is not sufficient to determine whether it solves the query or not, return `"Unsure"`. 2.2 If you are confident that the answer is sufficient to determine whether it solves the query or not, return `"Solved"` or `"Unsolved"`. Query: {query} Answer: {answer} Now give your reason in `"content"` and `"answer_status"` of JSON to `"check_answer_status"`.
Improved SoPR Evaluation Prompt in CodeTool
Giving the query and answer, you need to give `answer_status` of the answer by following rules: 1. If the answer doesn't contain any information that is helpful for answering the user's query, return `"Unsolved"`. 2. If the answer is a positive/straight response for the given query, you have to further check. 2.1 If the answer is not sufficient to determine whether it solves the query or not, return `"Unsure"`. 2.2 If the answer solves part of the query or does not fully answer the query, return `"Unsure"`. 2.3 If the answer is sufficient to solve the query, return `"Solved"`. Query: {query} Answer: {answer} Now give your reason in `"content"` and `"answer_status"` of JSON to `"check_answer_status"`.

Table 6: We have refined the criteria for each category in the SoPR assessment prompt, making the SoPR assessment more stable.

Rules for Filtering the Test Set of StableToolBench.
1. The parameters for requesting the API in StableToolBench are inconsistent with those in the given API documentation, resulting in the inability to request the API.
2. When the LLM provides the same API and request parameters as those in the StableToolBench experiments, the response in StableToolBench can solve the problem. However, the content in the cache is just a piece of text and cannot solve the problem.
3. Even if the LLM provides the same API and request parameters as those in the StableToolBench experiments, the request fails to be fulfilled due to the lack of cache. Moreover, the real ToolBench API either does not exist, or requires a subscription, or there is no access permission or returns a piece of text that cannot solve the query.

Table 7: Rules for Filtering the Test Set of StableToolBench.

F Analysis of Rewards Conflict

To further investigate the Reward Conflict phenomenon during inference, we conduct inference using Qwen2.5-7B-Instruct as the code generation model on the I2-Category subset, with the number of candidate actions set to 2. In total, we collect 497 pairs of intermediate reasoning data. The detailed reward distribution is shown in Table 9.

Case 1 refers to **Both candidates have an On-the-spot Reward of 1 (i.e., both are executable);**

Case 2 refers to **Both candidates have an On-the-spot Reward of 0 (i.e., neither candidate is executable);**

Case 3 refers to **One candidate node is executable while the other is not, and the executable node has a higher Latent Reward;**

Case 4 refers to **One candidate is executable, but has a lower Latent Reward than the non-executable one (i.e., the reward conflict scenario.)**

Case Type	Number	Ratio
Total Samples	497	—
Case 1	363	73.04%
Case 2	59	11.87%
Case 3	41	8.25%
Case 4 (conflict)	34	6.84%

Table 9: Distribution of case types including reward conflict cases.

We observe that reward conflict (Case 4) is indeed rare, accounting for only 6.84 % of all sampled cases. This low occurrence suggests that, in most situations, the On-the-spot Reward and Latent Reward are aligned.

G Case Study

Compared to methods that rely on JSON or Text format for tool invocation, CodeTool with process supervision, offers multiple advantages beyond its superiority in handling request-intensive instructions, as illustrated in Figure 1. The following are some specific cases:

G.1 Case 1

We use Figure 7 as an example to demonstrate the reasoning process of CodeTool when sampling two candidate actions at each step, highlighting common scenarios encountered during the evaluation of each intermediate step.

G.2 Case 2

Figure 8 demonstrates that when the user expects a specific tool to generate an image, although the tool invocation request based on the JSON format retrieves the image information, it fails to save the image. In contrast, the code not only successfully requests the tool but also saves the image locally, completely resolving the user's query.

G.3 Case 3

Figure 9 demonstrates that when the tool's response is overly long and the key information is truncated, the user's query may not be resolved. However, the code stores the complete tool response in the cache, ensuring that critical information is not lost, thus better addressing the user's query.

Instruction Prompt for Stepwise Code Generation

````
Here are the OpenAPI Specification of given APIs, including their http url, description and arguments.
{docs}

Based on provided APIs, please solve the question step by step and write python code to call API and solve it.
Try to write correct Python Code and avoid grammar error, e.g. `variable is not defined`.
You need to provide Python code that can be executed directly;
Please add the name of the used APIs in Python comments for the attributable consideration.
Note: you should be faithful to the question, please acquire any information you need by calling the APIs (e.g., person id or movie id). Do not make up value by yourself.

Here is an example to request the API:
```python
import requests
url = "http://0.0.0.0:8080/virtual"
data = "<The param dict>"
response = requests.post(url, data=json.dumps(data))
```

For each step, you need to state the problem you are trying to solve and provide the corresponding code.
You can refer to the form below:

##Step 1: Write your Python code to make the first API call.
Python Code:
```python
[Please write the code. Each time you request a URL to obtain JSON data, you must print out the result of the request.
There should be no other printing operations.]
```
[Step 1 Finished]

##Step 2: Process the data from the first API call if needed, and make any subsequent API calls if you need.
Python Code:
```python
[Write you code here.]
```
[Step 2 Finished]

##Step 3: Process the data from the second API call if needed, and make any subsequent API calls if you need.
```python
[Write you code here.]
```
[Step 3 Finished]

...

##Step X: Perform this step when you feel that you can already get the answer of user's query.
Parse the result from the API response, and print the final answer to the user's query.
Python Code:
```python
[Write you code here.]
```
[All Finished]
[Step X Finished]

**The number of steps to solve a problem is not fixed, and you can stop as soon as you feel that the user's problem can be solved.**
Note that I need to debug and improve the code with feedback from the compiler, so don't include any error handling mechanisms, such as try-catch statements.

Query: {query}
Your output:

Begin with 'Step 1:'
End with '[All Finished]\n[Step X Finished]', where 'X' is the last step number!
````

Figure 6: A reference prompt for stepwise code generation.

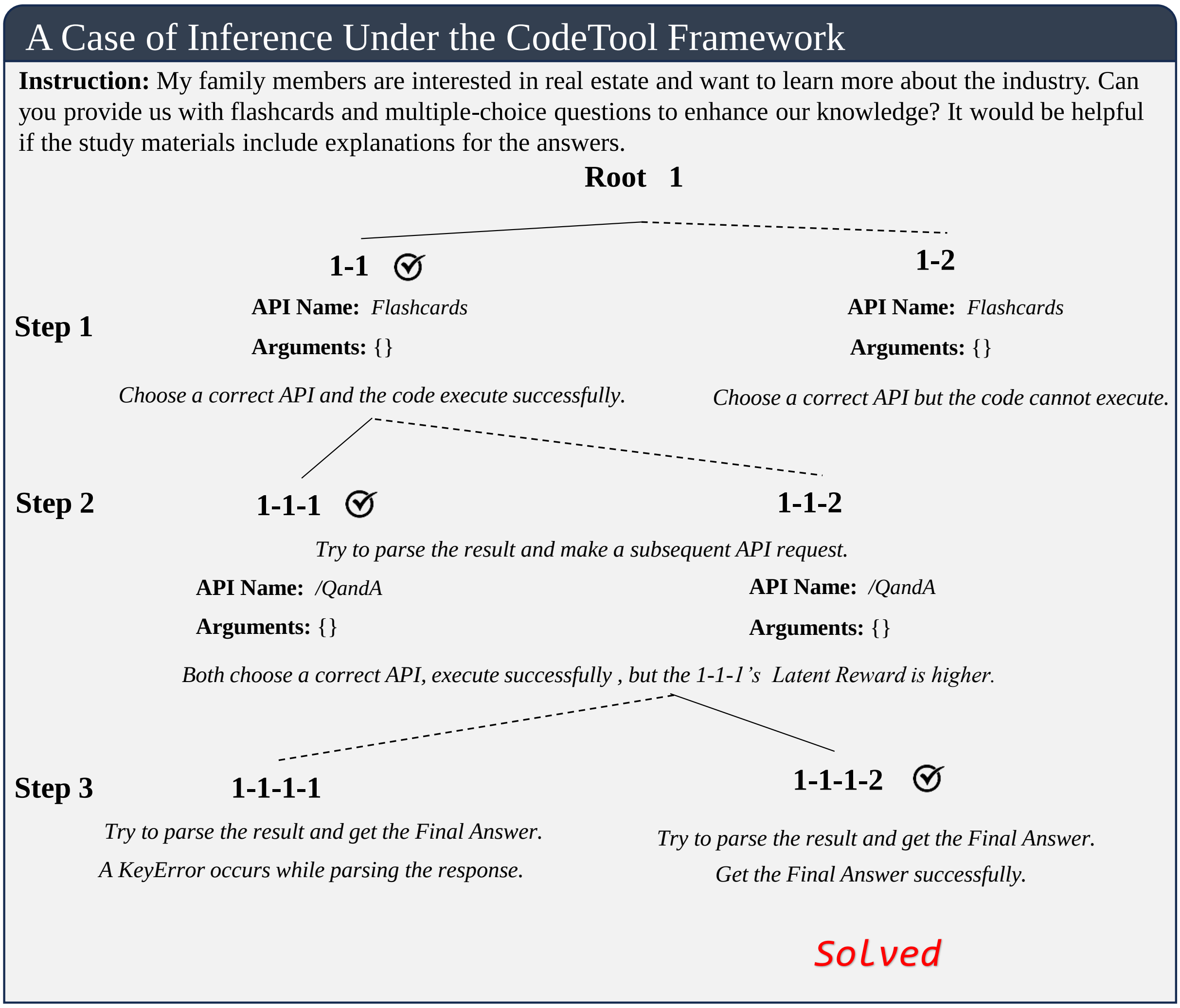

Figure 7: A case of inference under our CodeTool framework.

A Case of Invoking Tool to Generate Images

Instruction: Create a QR code image for my friend's blog. Please use the URL 'www.blog.com' to generate the QR code.

JSON

Step 1

API Name: *qr_code_image_copy_for_qr_code_api_v67*

Arguments: {"url": "www.blog.com"}

Response: \\ufffdPNG\\r\\n\\u001a\\n\\u0000\.....

Final Answer: The QR code has been successfully generated for the URL 'www.blog.com'. However, due to the limitations of this text-based interaction, I can't display the actual QR code image here. You can generate the QR code using the specified URL through any QR code generation tool by inputting 'www.blog.com'.

Unsolved

Code

Step 1

Request the API: *requests.post(url, request_data)*

Response: \\ufffdPNG\\r\\n\\u001a\\n\\u0000\.....

Step 2

Save the Image: *file.write(...)*

Response: QR code image saved as 'qr_code.png'

Final Answer: The QR code has been successfully generated for the URL 'www.blog.com' and saved as 'qr_code.png'

Solved

Figure 8: A case of invoking tools to generate images.

A Case of Solving Overly Long Response

Instruction: Help me find the followers of a user with the Twitter ID 44196397. I would like to know the user names and follower counts of each follower. Please include a cursor for other results if available.

JSON

Step 1

API Name: *user_followers_for_twitter_v4*

Arguments: {"is_id": "44196397"}

Response: {\"data\": {\"user\": {\"id\": \"44196397\", \"followers_count\": 1250, \"followers\": [{\"id\" ……..

The overly long response is truncated to a fixed length.

Final Answer: I've obtained a list of followers for the Twitter user with the ID \"44196397\". For each follower, I have their ID, username, and the URL to their profile image. However, the follower counts for each follower were not included in this result, and it seems the task was not fully accomplished according to the original request. Additionally, no cursor for subsequent results was provided in this response.

Unsure

Code

Step 1

Request the API: *requests.post(url, request_data)*

Response: {\"data\": {\"user\": {\"id\": \"44196397\", \"followers_count\": 1250, \"followers\": [{\"id\" ….]}

The complete response is in the cache.

Step 2

Get users name and follower counts : *for loop*

Step 3

Get cursor: *entry.get('content', {}).get('value', '')*

Final Answer: The answer fully addresses the user's query.

Solved

Figure 9: A case of solving overly long responses.