ToolCoder: A Systematic Code-Empowered Tool Learning Framework for Large Language Models

Hanxing Ding[1,2*]   **Shuchang Tao**[3*]   **Liang Pang**[1†]   **Zihao Wei**[1,2]
Jinyang Gao[3]   **Bolin Ding**[3]   **Huawei Shen**[1,2] **Xueqi Cheng**[1,2]

[1]State Key Laboratory of AI Safety, Institute of Computing Technology, CAS
[2] University of Chinese Academy of Sciences   [3] Alibaba Group
{dinghanxing18s, pangliang, weizihao22z, shenhuawei, cxq}@ict.ac.cn

Abstract

Tool learning has emerged as a crucial capability for large language models (LLMs) to solve complex real-world tasks through interaction with external tools. Existing approaches face significant challenges, including reliance on hand-crafted prompts, difficulty in multi-step planning, and lack of precise error diagnosis and reflection mechanisms. We propose `ToolCoder`, a novel framework that reformulates tool learning as a code generation task. Inspired by software engineering principles, `ToolCoder` transforms natural language queries into structured Python function scaffold and systematically breaks down tasks with descriptive comments, enabling LLMs to leverage coding paradigms for complex reasoning and planning. It then generates and executes function implementations to obtain final responses. Additionally, `ToolCoder` stores successfully executed functions in a repository to promote code reuse, while leveraging error traceback mechanisms for systematic debugging, optimizing both execution efficiency and robustness. Experiments demonstrate that `ToolCoder` achieves superior performance in task completion accuracy and execution reliability compared to existing approaches, establishing the effectiveness of code-centric approaches in tool learning. Our code is available at this link.

1 Introduction

As large language models (LLMs) continue to advance, tool learning has emerged as a critical capability, enabling LLMs to solve complex real-world tasks through dynamic interaction with external tools and APIs (Qu et al., 2024). This capability extends LLMs' problem-solving abilities and allows them to adapt to dynamic environments and specialized domains (Wang et al., 2024c).

Current approaches for tool learning predominantly follow a sequential *plan-execute-observe*
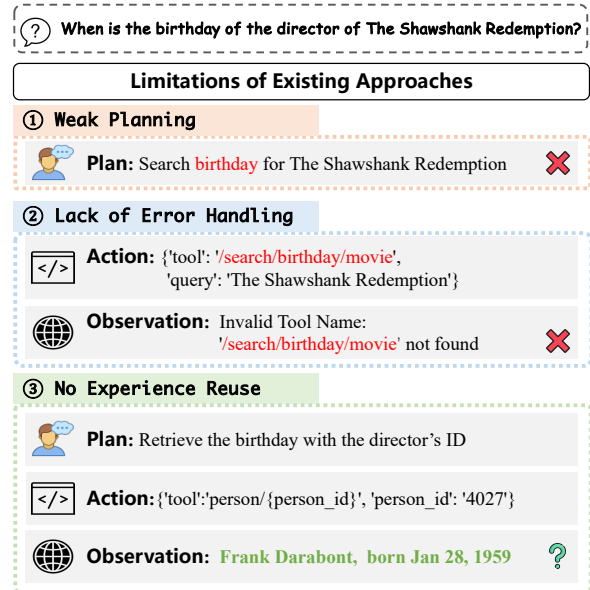

Figure 1: Illustration of key limitations in existing approaches: weak planning strategies, insufficient error handling, and lack of experience reuse.

paradigm (Yao et al., 2023; Paranjape et al., 2023; Lu et al., 2023), where LLMs iteratively plan actions, execute tools, and observe outcomes. While frameworks like ReAct (Yao et al., 2023) and Chameleon (Lu et al., 2023) have demonstrated success in simple task handling, they suffer from several limitations. As shown in Figure 1, these methods rely heavily on hand-crafted prompts and natural language reasoning, making multi-step planning difficult and often leading to unreliable performance in complex tasks (Huang et al., 2024; Wang et al., 2024b). Moreover, they lack precise error diagnosis and reflection mechanisms—when execution fails, they are unable to identify the actual causes of errors or propose targeted corrective actions (Wang et al., 2024b,a). Furthermore, current frameworks are unable to accumulate and reuse successful experiences from past executions, treating each query in isolation and repeatedly solving similar problems from scratch (Yuan et al., 2024a). These limitations hinder the robustness, adaptabil-

---

17876

ity, and scalability of tool-learning systems.

In this paper, we propose `ToolCoder`, a novel code-empowered framework that re-formulates tool learning as a code generation task. Inspired by software engineering principles, `ToolCoder` systematically addresses these challenges in tool learning through these key stages: First, `ToolCoder` converts vague natural language queries into structured Python function scaffold, analogous to the requirement analysis in software engineering, enabling clearly defined input-output specifications. Then, following modular design principles, `ToolCoder` systematically breaks down scaffold into subtasks with descriptive comments, enabling LLMs to leverage coding paradigms for complex reasoning and planning. In the implementation phase, `ToolCoder` generates well-structured runnable code for both sub-function and main functions, then executes them to obtain final responses. Successfully executed code snippets are stored in a function repository to accumulate proven implementations, helping solving similar problems. For failed executions, `ToolCoder` leverages explicit Python's error traceback mechanism for accurate error diagnosis, significantly improving the reliability of tool learning system.

To evaluate `ToolCoder`'s effectiveness, we conduct comprehensive experiments on multiple benchmark datasets (Song et al., 2023; Li et al., 2023). Our experimental results demonstrate that `ToolCoder` significantly outperforms SOTA approaches in all metrics. We also validate the necessity of each proposed component. Furthermore, we observe that `ToolCoder` brings more substantial improvements in code LLMs compared to base LLMs. These findings confirm the superiority of systematic code-centric framework and its potential for advancing tool learning systems.

The main contributions of this work are:

- We propose `ToolCoder` framework that re-formulates tool learning as a code generation task, leveraging both software engineering principles and LLMs' code generation capabilities.

- We design a systematic architecture that converts natural language task into modular code components, implements and executes them to generate final response. We also incorporate error diagnosis and code reuse mechanisms for efficiency and reliability.

- Extensive experiments on benchmarks demonstrate `ToolCoder` significantly improve the suc-

cess rate and reliability of tool learning.

2 Related works

2.1 Text-based Tool Learning

Recent approaches have focused on enhancing LLMs by integrating external tools to improve their problem-solving capabilities (Masterman et al., 2024; Qu et al., 2024; Qian et al., 2023; Yuan et al., 2024a). Typically, these methods follow a sequential *plan-execute-observe* paradigm and involve utilizing text few-shot prompt to decompose complex tasks into sequential steps, with the LLM executes each tool individually and incorporates the results into its context to manage data flow dependencies (Liu et al., 2024; Shinn et al., 2023; Shi et al., 2024b; Paranjape et al., 2023). For instance, ReAct (Yao et al., 2023) enables LLMs to generate both reasoning traces and action plans, facilitating more effective task execution. Similarly, Chameleon (Lu et al., 2023) employs a compositional reasoning framework that dynamically assembles specialized tools to address complex reasoning tasks. RestGPT (Song et al., 2023) introduces a coarse-to-fine online planning approach, allowing LLMs to iteratively refine task decomposition. However, these text-based methods heavily rely on carefully crafted prompts and struggle with complex multi-step tasks, limiting reliability. `ToolCoder` addresses these challenges by re-formulating tool learning as code generation, leveraging structured code generation and software engineering principles for better adaptability.

2.2 Code-based Tool Learning

Code pre-training has been shown to significantly enhance the chain-of-thought (CoT) performance of LLMs, enabling them to logically decompose complex tasks into smaller, more manageable subtasks (Lyu et al., 2023; Cheng et al., 2023; Ye et al., 2023). Recent studies have further highlighted the potential of code-empowered LLMs to improve the planning and reasoning capabilities of these models (Yang et al., 2024; Chen et al., 2023; Gao et al., 2023). For example, certain approaches enable LLMs to generate programmatic chains of thought to solve complex numeric reasoning tasks, yielding impressive performance (Chen et al., 2023; Gao et al., 2023). In the context of tool learning, LLMs can generate code snippets as actions (Wang et al., 2024b), leveraging widely used Python functions and simplifying operations
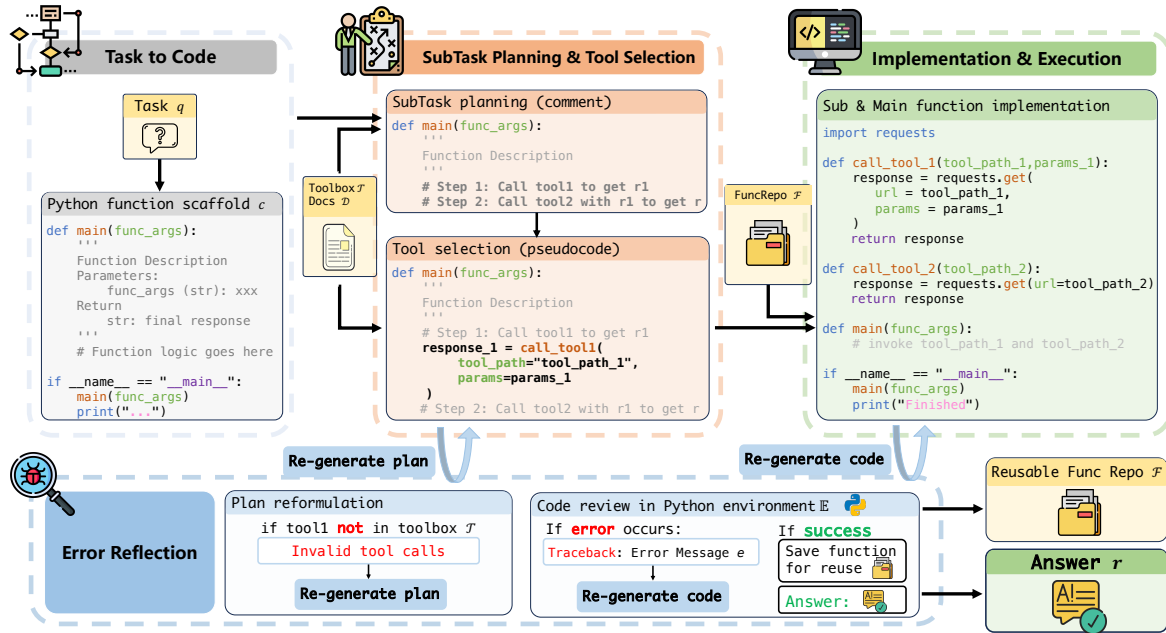
Figure 2: Overview of our `ToolCoder` framework, which converts a user task into executable Python code through task-to-code transformation, tool planning and selection, code implementation, and error reflection. The framework also leverages reusable function snippets to enhance efficiency and accuracy.

such as lengthy for-loops (Shi et al., 2024a). Previous approaches generate Python code in a single pass, lacking iterative refinement, intent alignment, and learning from past executions (Hsieh et al., 2023; Yuan et al., 2024b; Wang et al., 2024b). Our `ToolCoder` framework addresses these limitations by integrating software engineering principles, including modular code construction, stepwise verification, and experience-driven refinement, ensuring the generated code is executable and robust.

3 Methodology

We formulate the tool learning task, present our motivation, and propose the `ToolCoder` framework.

3.1 Task Formulation and Motivation

The goal of tool learning is to enable a LLM $\mathcal{M}$ to accomplish natural language tasks by generating and executing appropriate tool sequences. Given a task q, the LLM selects appropriate tools from a toolbox $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$, where each tool t_i is accompanied by its documentation $d_i \in \mathcal{D}$ describing its functionality, arguments, and output schema. Through tool interaction, $\mathcal{M}$ generates a solution response r that finally addresses the task.

Existing methods struggle with complex task planning and robust execution. To address these limitations, We draw inspiration from software engineering principles, which emphasize modular design, error diagnosis, and code reusability. Fur-

thermore, by expressing complex tasks through structured code, LLMs can leverage established programming paradigms to enhance their systematic reasoning and planning capabilities (Gao et al., 2023; Wang et al., 2024b).

Inspired by these insights, we reformulate tool learning as a code generation task. This reformulation enables us to *apply fundamental software engineering principles throughout the tool learning process: from initial requirement analysis and task decomposition, through modular implementation, to systematic error handling and code reuse.* By structuring tool operations as well-defined programming components, we can leverage both the systematic methodology of software engineering and the code generation capabilities of LLMs.

3.2 `ToolCoder`: Code-Empowered Tool Learning Framework

We propose a novel framework `ToolCoder`, which converts queries into Python function scaffolds (§ 3.2.1), performs subtask planning and tool selection (§ 3.2.2), and generates executable implementations (§ 3.2.3) with error reflection mechanisms (§ 3.2.4), as illustrated in Figure 2.

3.2.1 Task-to-Code Transformation

Our `ToolCoder` starts by identifying user needs and transforming the natural language task q into a structured Python function scaffold c, aligning with the requirements analysis phase in software

```python
def get_directed_movie_count(director_name: str) -> int:
    '''
    Get the number of movies directed by a specific director.

    Parameters:
        director_name (str): The full name of the director whose
        movies to count.

    Returns:
        int: The number of movies directed by the specified director.
    '''
    # Function logic goes here

if __name__ == "__main__":
    number_of_movies_directed = get_directed_movie_count(
        director_name="Sofia Coppola")
    print("Number of movies directed by Sofia Coppola:",
        number_of_movies_directed)
```

Figure 3: An example of the Python function scaffold c generated by ToolCoder for the query: *give me the number of movies directed by Sofia Coppola.*

engineering. This process can be formalized as:

$$c = \mathcal{M}_{T2C}(q), \tag{1}$$

where $\mathcal{M}_{T2C}$ represents the task-to-code transformation module, following the prompt template shown in Appendix Figure 9. The scaffold c, as shown in Figure 3, includes several key components: a descriptive function name that indicates the intended task, a parameter list that specifies required inputs, and a comprehensive docstring that documents the function's purpose, parameter descriptions, and expected return value. The function body is deliberately left empty at this stage, serving as a structured placeholder to be filled during subsequent planning and tool execution phases.

By converting natural language queries into structured code scaffolds, ToolCoder clearly understands the requirements, precise input and output specifications, and clear task goals, laying a solid foundation for task decomposition.

3.2.2 Subtask Planning and Tool Selection

This stage focuses on subtask planning and tool selection, decomposing tasks into modular components and determining appropriate tools, following modular design in software engineering. Given candidate toolbox $\mathcal{T}$ and the generated Python scaffold c, ToolCoder translates high-level tasks into concrete, actionable subtasks:

$$\boldsymbol{s} = \mathcal{M}_{TP}(c, \mathcal{T}) = \{s_1, s_2, \ldots, s_m\}, \tag{2}$$

where $\mathcal{M}_{TP}$ is the subtask planning module (implemented with the prompt in Appendix Figure 10). $\mathcal{M}_{TP}$ analyzes the Python scaffold c and generates concrete subtasks $\boldsymbol{s} = \{s_1, s_2, \ldots, s_m\}$. These subtasks are then embedded as structured code comments within scaffold c, forming a subtask-based

plan $c_{\boldsymbol{s}}$ that outlines all the necessary steps.

The tool selection module $\mathcal{M}_{TS}$ (prompt in Appendix Figure 11) systematically analyzes the subtask-based plan $c_{\boldsymbol{s}}$ and references the documentation of each tool to understand their input-output specifications. Based on the analysis, ToolCoder generates pseudocode $c_{\boldsymbol{p}}$ whose main function body contains proper tool invocation sequences and clear data flow pathways, such as parsing the necessary parameters from the previous API to input the next API call. Each necessary tool call is placed with a sub-function placeholder, whose specific functionality is not implemented.

Through structured planning and tool selection, ToolCoder maximizes the model's reasoning potential by leveraging coding paradigms, ensuring reliable and systematic task completion.

3.2.3 Implementation and Execution

Implementation In this stage, ToolCoder implements the specific functionality of sub-function placeholder, using the API documentation of selected tools $\mathcal{T}_{\boldsymbol{s}}$ and a repository of correctly-executed subfunction $\mathcal{F}$, assembling them into an executable program F.

$$F = \mathcal{M}_{CG}(c_{\boldsymbol{p}}, \mathcal{T}_{\boldsymbol{s}}, \mathcal{F}), \tag{3}$$

where $\mathcal{M}_{CG}$ is the code generation module (prompt in Appendix Figure 12), $\mathcal{T}_{\boldsymbol{s}}$ denotes the subset of candidate tools selected based on the pseudocode $c_{\boldsymbol{p}}$, and $\mathcal{F}$ represents the reusable repository of successfully executed functions. For each subplan s_i, ToolCoder implements corresponding sub-functions by leveraging tool documentation from $\mathcal{T}_{\boldsymbol{s}}$ and reusable repository $\mathcal{F}$.

Execution Once the function F is generated, it is executed within the Python environment $\mathbf{E}$ to produce response r or exceptions e:

$$r, e = \mathbf{E}(F). \tag{4}$$

If program F is successfully executed, the generated response r will be returned as the final answer to the user's task q. Otherwise, the error message e triggers an error reflection process to identify and resolve any issues.

Reusable Function Repository To facilitate the implementation process, ToolCoder maintains a reusable function repository $\mathcal{F}$, which stores successfully executed sub-function snippets that implement specific API functionalities. During task execution, ToolCoder extracts these successful sub-

functions from program F and adds them to repository $\mathcal{F}$, creating a collection of reusable implementations with their execution contexts.

Through collecting these sub-function snippets, `ToolCoder` builds a reliable function repository for each tool used. This enables efficient code reuse in future tasks, preventing potential errors and eliminating redundant development while enhancing system efficiency and reliability.

3.2.4 Error Reflection

To mitigate common errors like vague planning, invalid tool selection, and execution failures, `ToolCoder` leverages explicit exception tracebacks to identify and correct issues, and iteratively refine task plans and enhance the reliability of the system.

Plan Reformulation   If the plan c_{s} generated by the task decomposition module $\mathcal{M}_{TD}$ includes invalid or non-existent tools, `ToolCoder` invokes planning reformulation strategy to refine the plan. Specifically, it detects errors by cross-referencing each tool in the plan against the available tools in $\mathcal{T}$. For every invalid tool, it generates explicit instructions for the LLM, guiding LLM to select an appropriate alternative from $\mathcal{T}$ to fulfill the required functionality. This ensures that the reformulated plan $c_{s'}$ consists entirely of tools that are present in the candidate toolbox.

Code Review   If the initial function F fails, producing an error e, `ToolCoder` employs code review to analyze the error, identify its cause, and generate corrections. Python's clear and detailed exception tracebacks allow precise pinpointing of the error location and facilitate understanding of the underlying issue. The function F is revised based on the current error e and then re-executed using $\mathbf{E}$, yielding a new result r and an updated error state e. This process is performed iteratively, typically up to three times, until execution succeeds (i.e., $e = \emptyset$) or the maximum number of attempts is reached.

4   Experiments

In this section, we describe the experimental setup, including the datasets, evaluation metrics, and baseline methods. We then present the main experimental results, followed by a comprehensive analysis that offers an in-depth examination of the model's capabilities and efficiency.

4.1   Experimental Settings

Datasets.   To validate the effectiveness of our method in tool learning, we conduct experiments on two well-established benchmarks: Rest-Bench (Song et al., 2023) and API-Bank (Li et al., 2023). RestBench comprises two real-world scenarios: TMDB, a human-annotated dataset with 100 tasks utilizing 54 tools, designed for movie-related scenarios; and Spotify, which features 40 tools tailored to music-related tasks. Each tool in RestBench is paired with a detailed RESTful API document, making it inherently suitable for benchmarking the real-world tool-learning capabilities of LLMs. API-Bank consists of 73 commonly used APIs and 314 tool-use dialogues with 753 manually annotated API calls, designed to evaluate the effectiveness of LLMs in utilizing tools.

Evaluation metrics.   For RestBench (Song et al., 2023), we adopt their evaluation framework and utilize three metrics: success rate (**Success%**), which relies on human evaluation to determine whether the model's output effectively fulfills the user query; correct path rate (**Path%**), which measures the proportion of ground-truth tools correctly included in the model-generated tool calls. We also introduce **Accuracy%**, specifically for the TMDB scenario, to assess whether the generated answers align with the ground-truth answers. These ground-truth answers are derived by executing the annotated solution paths, providing a reliable benchmark for comparison. For API-Bank (Li et al., 2023), we evaluate API-calling performance using their **Correctness%** metric, which compares the ground-truth API outputs with the execution results of the model-generated API calls.

4.2   Baselines

We compare our method against seven well-known baselines: (1) ReAct (Yao et al., 2023), prompting LLMs to generate interleaved chains of thought and actions. (2) Chameleon (Lu et al., 2023), an LLM-based agent that creates multi-step plans for tool usage and executes them sequentially. (3) ConAgents (Shi et al., 2024b), which facilitates the collaboration of three specialized LLMs to solve complex tasks. (4) RestGPT (Song et al., 2023) features a coarse-to-fine planning module and a tool executor. (5) EasyTool (Yuan et al., 2024b) converts lengthy and diverse tool documentation into unified, concise instructions for easier tool usage. (6) ATC (Shi et al., 2024a) utilizes a chain of tools

Methods	RestBench-TMDB			RestBench-Spotify		API-Bank (LV1)	API-Bank (LV2)
	Success(%)	Accuracy(%)	Path(%)	Success(%)	Path(%)	Correctness(%)	Correctness(%)
Text-based Tool Learning							
ReAct (Yao et al., 2023)	76.0	48.0	50.0	68.42	52.63	73.93	56.30
Chameleon (Lu et al., 2023)	75.0	45.0	52.0	70.18	63.16	74.87	37.04
ConAgents (Shi et al., 2024b)	72.0	38.0	57.0	64.92	68.42	67.26	36.24
RestGPT (Song et al., 2023)	83.0	33.0	49.0	61.41	57.89	65.47	34.83
EasyTool (Yuan et al., 2024b)	75.0	45.0	62.0	62.19	64.92	74.97	58.24
Code-based Tool Learning							
ATC (Shi et al., 2024a)	78.0	58.0	62.0	65.47	68.42	70.21	52.18
CodeAct (Wang et al., 2024b)	80.0	56.0	67.0	71.93	66.67	75.94	54.07
ToolCoder	**85.0**	**78.0**	**83.0**	**87.72**	**78.95**	**83.08**	**62.41**
w/o Reusable Repository	83.0	71.0	78.0	78.95	71.93	83.08*	62.41*
w/o Error Reflection	75.0	65.0	77.0	73.68	70.18	79.85	58.02

Table 1: Main experimental results on RestBench and API-Bank datasets. All methods are implemented with *gpt-4o-mini*. The best results are bolded. * Results from the API-Bank dataset are reported without the reusability mechanism due to its built-in API calls in this dataset.

through programming and proposes a black-box probing method. (7) CodeAct (Wang et al., 2024b), enabling LLMs to generate executable code snippets as actions to interact with tools.

4.3 Main Results

The experimental results in Table 1 provide a comprehensive evaluation of ToolCoder in comparison to existing text-based and code-based tool learning approaches. These results highlight the significant performance improvements achieved by ToolCoder, validating the effectiveness of its code-empowered framework.

From the experimental results presented in Table 1, text-based approaches demonstrate limited effectiveness in complex tool learning scenarios, as evidenced by their low path rates and accuracy scores. This underperformance stems from two key limitations: the inherent constraints of natural language prompting for structured reasoning, and their inability to effectively handle execution errors. While code-based methods like ATC and CodeAct show significant improvements in path planning, achieving higher success rates than their text-based counterparts, they still face challenges in maintaining consistent performance across different scenarios. This is particularly evident in their accuracy scores, suggesting that even with better planning capabilities, these methods still lack robust mechanisms for error handling and output verification.

Building on the strengths of code-empowered LLMs, ToolCoder achieves SOTA performance across all benchmarks compared to baseline methods, consistently achieving the highest success, accuracy, and correct path rates, showcasing its
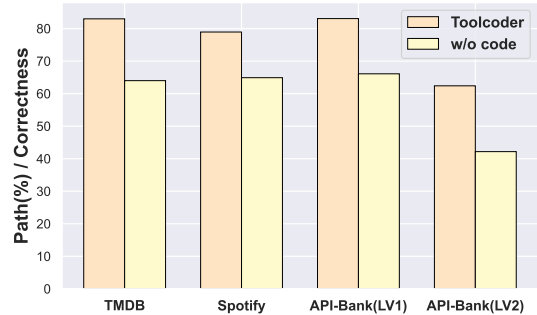

Figure 4: Impact of code-empowered planning on path correctness. Removing the structured Python function scaffold c leads to significant performance degradation across all benchmarks.

robustness and adaptability. For instance, on RestBench-Spotify, ToolCoder improves the success rate by 10.79% and the accuracy by 22.22% compared to the strongest baseline, CodeAct. Similarly, on API-Bank (LV2), it improves correctness by 8.34% over CodeAct, highlighting its ability to handle multi-step tool interactions. The performance gains stem from ToolCoder's code-empowered framework, which leverages the reasoning capabilities of code-empowered LLMs and the advantages of systematic software development principles to generate precise tool execution paths and accurate final responses.

4.4 Ablation Study

Effect of Code-Empowered Planning To investigate the effect of code-empowered planning, we remove the structured Python function scaffold c (denoted as *w/o code*) during the tool planning phase and compare its performance with the original ToolCoder. As shown in Figure 4, the correct path rate drops significantly without c across all
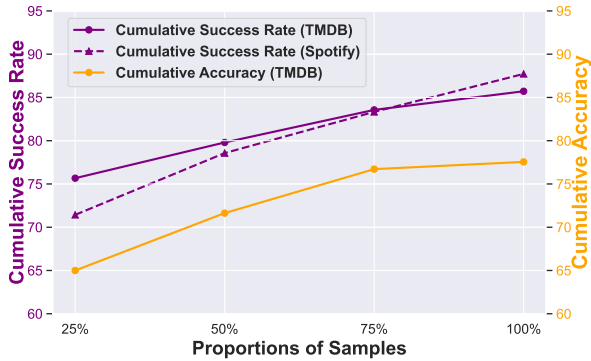
Figure 5: Effect of the function reusability mechanism on cumulative success rate and accuracy across different sample proportions (inference process progresses).

Methods	ToolAlpaca (Real-world)		
	Procedure	Response	Overall
ReAct	64.86	60.81	54.05
Chameleon	68.06	58.33	57.92
CodeAct	68.92	58.11	56.94
ToolCoder	**78.38**	**75.68**	**72.97**
w/o Error Reflection	76.46	68.11	67.94

Table 2: Performance comparison of different models on real-world API tasks using the ToolAlpaca dataset. Metrics include Procedure, Response, and Overall scores. All methods are implemented with *gpt-4o-mini*. Due to the small number of questions under each tool category, we do not use the reusability mechanism on this dataset.

benchmarks. This decline highlights the scaffold's role in enhancing the model's reasoning capabilities. Without the scaffold, the model struggles to fully leverage its code-enhanced reasoning abilities, leading to a marked decrease in correct path rate. The absence of c results in a reduced understanding of task objectives and an impaired ability to structure information into coherent plans. These findings underscore the critical importance of the structured Python function scaffold c in activating the model's reasoning potential and maintaining effective task planning.

Effect of Reusable Function Repository   To assess the impact of the reusable function repository $\mathcal{F}$, we conduct an ablation study by removing the reusability mechanism. As shown in Table 1, this results in a 2–5% performance drop across various benchmarks, highlighting the effectiveness of reusable code snippets.

Furthermore, to gain deeper insights into how reusability enhances tool execution, we analyze the cumulative success rate and cumulative accuracy across different sample proportions, as illustrated in Figure 5. The results demonstrate a significant performance boost due to the reusability mechanism. As the inference process progresses (with increasing sample proportions on the x-axis), the cumulative success rate steadily grows. This indicates that the model accumulates experience from past successes, aiding in the generation of more robust code. For example, on the TMDB dataset, the cumulative success rate approaches 90% as more samples are utilized, reflecting improved generalization and execution efficiency. Additionally, the cumulative accuracy on TMDB consistently improves, further confirming that the reusability mechanism not only enhances success rates but

also improves answer precision.

Effect of Error Reflection Mechanism   To assess the impact of the error reflection mechanism, we conduct an ablation study by removing it from `ToolCoder` and evaluating the resulting performance changes. The results, as shown in Table 1, indicate a significant performance drop across multiple datasets when the error reflection mechanism is removed. Both correct path rate and success rates decrease markedly. This decline is attributed to the error reflection mechanism's ability to diagnose planning and execution errors, allowing for iterative refinement and adaptive self-correction. Without this mechanism, the model's reliability is severely compromised. This ablation study demonstrates the critical role of the error reflection mechanism in ensuring robust tool learning and effective adaptation to complex execution challenges.

4.5   Analysis of Generalization Capabilities

Experiments on New Dataset ToolAlpaca-Real To evaluate the generalization performance of `ToolCoder`, we conduct experiments on the ToolAlpaca dataset (Tang et al., 2023) with real-world API tasks. The evaluation framework contains three metrics: Procedure, Response, and Overall. Procedure assesses the model's ability to select appropriate actions, use correct parameters, and execute steps without redundancy. Response measures the model's capability to produce outputs that fully satisfy the user's requirements. Overall is a comprehensive metric that reflects the accuracy and effectiveness of the entire process, encompassing both procedural correctness and response quality.

The experimental results in Table 2 demonstrate that `ToolCoder` outperforms traditional methods, including ReAct, Chameleon, and CodeAct, across

Methods	RestBench-TMDB		
	Success	Accuracy	Path
Qwen2.5-14B-Instruct			
ReAct (Yao et al., 2023)	67.0	40.0	42.0
Chameleon (Lu et al., 2023)	68.0	38.0	45.0
CodeAct (Wang et al., 2024b)	72.0	58.0	62.5
ToolCoder	**81.0**	**72.0**	**76.0**
Qwen2.5-Coder-14B-Instruct			
ReAct (Yao et al., 2023)	68.0	41.5	47.0
Chameleon (Lu et al., 2023)	70.0	45.4	50.0
CodeAct (Wang et al., 2024b)	80.0	65.0	75.0
ToolCoder	**85.0**	**76.0**	**80.0**
Qwen2.5-32B-Instruct			
ReAct (Yao et al., 2023)	73.0	50.0	51.0
Chameleon (Lu et al., 2023)	71.0	48.5	54.5
CodeAct (Wang et al., 2024b)	82.0	62.5	76.5
ToolCoder	**89.0**	**82.0**	**84.0**
Qwen2.5-Coder-32B-Instruct			
ReAct (Yao et al., 2023)	76.5	56.5	58.0
Chameleon (Lu et al., 2023)	78.0	56.0	57.0
CodeAct (Wang et al., 2024b)	92.0	82.0	85.0
ToolCoder	**96.0**	**90.0**	**91.0**

Table 3: Evaluate the performance of ToolCoder on different open source LLMs on RestBench-TMDB.

all metrics. ToolCoder achieves significant improvements in Procedure and Response, which lead to a superior Overall performance. These advancements can be attributed to the code-empowered LLM's superior planning and execution capabilities, enabling more accurate action selection and task completion. Additionally, the integration of a reflection strategy further enhances the quality of the responses, resulting in more effective handling of real-world API tasks compared to baselines.

Experiments on Open-Source LLMs To evaluate the effectiveness of ToolCoder with open-source LLMs, we conduct comprehensive experiments on RestBench-TMDB. The results in Table 3 reveal several significant findings. First, model scaling demonstrates consistent performance improvements across all methods. When scaling from 14B to 32B, we observe substantial gains in success rate, accuracy, and path correctness across all approaches. Second, the comparison between base models and their coder-enhanced counterparts reveals the substantial advantages of code-specialized capabilities. This is particularly evident in code-empowered methods like ToolCoder and CodeAct. These improvements demonstrate that code-specialized models are better equipped to handle the structured reasoning and precise execution required in tool learning tasks. Third,

Methods	RestBench		API-Bank	
	TMDB	Spotify	LV1	LV2
ReAct	7.78	8.68	6.45	8.38
Chameleon	6.52	7.12	5.64	6.92
ConAgents	6.78	7.68	5.45	7.38
RestGPT	9.04	10.24	7.27	9.84
ToolCoder	7.69	8.70	6.46	8.47

Table 4: Efficiency analysis of different methods on the RestBench and API-Bank datasets. We report the average number of API calls required per question, where a lower value indicates higher efficiency.

ToolCoder consistently outperforms all baselines across different model configurations. This superior performance validates the effectiveness of our code-empowered framework in leveraging the code-specific strengths of LLMs for tool learning.

4.6 Efficiency Analysis

To analyze the efficiency of API usage among different methods, we compare the average number of LLMs calls required per question on the RestBench and API-Bank benchmarks, as shown in Table 4. ToolCoder demonstrates comparable efficiency to baseline methods such as ReAct, Chameleon, and ConAgents, with basically similar API usage. This result suggests that ToolCoder achieves higher correct path rates and success rates in task planning and execution compared to other methods, without incurring additional efficiency costs, thereby maintaining a practical balance between improved performance and resource consumption.

5 Conclusion

We present ToolCoder, a novel framework that reformulates tool learning as a code generation task, leveraging software engineering principles and Python function structures to enhance complex reasoning and execution reliability. By systematically transforming natural language queries into structured function scaffolds, incorporating descriptive comments for multi-step planning, and maintaining a repository of successfully executed functions, ToolCoder establishes a robust approach to tool learning. Its integrated error diagnosis mechanism further ensures adaptability based on code-based feedback. Experimental results demonstrate its superior performance in both task completion accuracy and execution reliability compared to existing methods, validating the effectiveness of our code-centric approach to tool learning.

Acknowledgement

Limitations

Despite the promising performance demonstrated by our approach, several limitations constrain its applicability and highlight directions for future improvement. First, the method relies heavily on the availability of clear, well-defined, and comprehensive API documentation. When such documentation is incomplete, ambiguous, or inconsistent, the model's ability to infer tool behavior and execute tasks correctly is significantly hindered. This dependency limits the approach's robustness in real-world scenarios where documentation quality is not guaranteed. Second, our approach adopts a global planning strategy that excels in generating structured and efficient solutions but lacks the flexibility to address dynamic, real-time constraints in evolving environments. For example, unforeseen changes in task requirements or partial observability may require adaptive or incremental adjustments that our current framework does not fully support. Finally, scalability remains a challenge when dealing with tasks involving numerous tools and complex interactions. While our approach is effective for moderately complex tool planning, scenarios with a large number of interdependent tools may introduce combinatorial challenges, potentially resulting in suboptimal plans.

References

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

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *CoRR*, abs/2308.00675.

Chengrui Huang, Zhengliang Shi, Yuntao Wen, Xiuying Chen, Peng Han, Shen Gao, and Shuo Shang. 2024. What affects the stability of tool learning? an empirical study on the robustness of tool learning frameworks. *CoRR*, abs/2407.03007.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Yulong Liu, Yunlong Yuan, Chunwei Wang, Jianhua Han, Yongqiang Ma, Li Zhang, Nanning Zheng, and Hang Xu. 2024. From summary to action: Enhancing large language models for complex tasks with open world apis. *CoRR*, abs/2402.18157.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 305–329, Nusa Dua, Bali. Association for Computational Linguistics.

Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The landscape of emerging AI agent architectures for reasoning, planning, and tool calling: A survey. *CoRR*, abs/2404.11584.

Bhargavi Paranjape, Scott M. Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Túlio Ribeiro. 2023. ART: automatic multi-step reasoning and tool-use for large language models. *CoRR*, abs/2303.09014.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation

for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. Tool learning with large language models: A survey. *CoRR*, abs/2405.17935.

Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Zhumin Chen, Suzan Verberne, and Zhaochun Ren. 2024a. Chain of tools: Large language model is an automatic multi-tool learner. *CoRR*, abs/2405.16533.

Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. 2024b. Learning to use tools via cooperative and interactive agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10642–10657, Miami, Florida, USA. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world applications via restful apis. *CoRR*, abs/2306.06624.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *CoRR*, abs/2306.05301.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. 2024a. LLMs in the imaginarium: Tool learning through simulated trial and error. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10583–10604, Bangkok, Thailand. Association for Computational Linguistics.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024c. MINT: evaluating llms in multi-turn interaction with tools and language feedback. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. 2024. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *CoRR*, abs/2401.00812.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2023, Taipei, Taiwan, July 23-27, 2023*, pages 174–184. ACM.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. 2024a. CRAFT: customizing llms by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. 2024b. EASYTOOL: enhancing llm-based agents with concise tool instruction. *CoRR*, abs/2401.06201.

A Appendix

To provide an intuitive understanding of how our framework works, we present detailed examples in this section. We first provide example solutions for our `ToolCoder` for the query: *give me the number of movies directed by Sofia Coppola*. All results are generated with `gpt-4o-mini`. Furthermore, we also display the prompts used to implement each module on the RestBench-TMDB dataset.

```python
def get_directed_movie_count(director_name: str) -> int:
    """
    Get the number of movies directed by a specific director.

    Parameters:
        director_name (str): The full name of the director whose movies to count.

    Returns:
        int: The number of movies directed by the specified director.
    """
    # Step 1. Search for the director "Sofia Coppola" using the /3/search/person API to find her person_id.
    # Step 2. Retrieve the movie credits for Sofia Coppola using her person_id with the /3/person/{person_id}/movie_credits API.

if __name__ == "__main__":
    number_of_movies_directed = get_directed_movie_count(director_name="Sofia Coppola")
    print("Number of movies directed by Sofia Coppola:", number_of_movies_directed)
```

Figure 6: An example of the task decomposition result c_s generated by ToolCoder for the query: *give me the number of movies directed by Sofia Coppola.*

```python
def get_directed_movie_count(director_name: str) -> int:
    """
    Get the number of movies directed by a specific director.

    Parameters:
        director_name (str): The full name of the director whose movies to count.

    Returns:
        int: The number of movies directed by the specified director.
    """
    # Step 1. Search for the director "Sofia Coppola" using the /3/search/person API to find her person_id.
    person_search_response = call_api(
        api_path="/3/search/person", params={"query": director_name}
    )

    # Get person_id from response
    person_id = (
        person_search_response.get("results")[0]["id"]
        if person_search_response.get("results")
        else None
    )

    # Step 2. Retrieve the movie credits for Sofia Coppola using her person_id with the /3/person/{person_id}/movie_credits API.
    if person_id:
        movie_credits_response = call_api(
            api_path="/3/person/{person_id}/movie_credits",
            params={"person_id": person_id},
        )

        # Step 3. Count the number of movies in the credits retrieved.
        # Filter for movies that Sofia Coppola directed
        directed_movies = [
            movie
            for movie in movie_credits_response["crew"]
            if movie["job"] == "Director"
        ]

        number_of_movies_directed = len(directed_movies)
        return number_of_movies_directed

    return 0

if __name__ == "__main__":
    number_of_movies_directed = get_directed_movie_count(director_name="Sofia Coppola")
    print("Number of movies directed by Sofia Coppola:", number_of_movies_directed)
```

Figure 7: An example of the pseudo-code c_p generated by ToolCoder for the query: *give me the number of movies directed by Sofia Coppola.*

```
import requests

# Define the required headers for API access.
headers = {"Authorization": "YOUR API KEY"}

def search_person(query: str, include_adult=False, language="en-US", page=1):
    '''
    Omitted for simplicity due to space limitations
    '''
    url = "https://api.themoviedb.org/3/search/person"

    # Prepare the parameters for the request
    params = {
        "query": query,
        "include_adult": include_adult,
        "language": language,
        "page": page
    }

    # Make the API request
    response = requests.get(url, headers=headers, params=params)
    response.raise_for_status()
    return response.json()

def get_person_movie_credits(person_id: int, language="en-US"):
    '''
    Omitted for simplicity due to space limitations
    '''
    url = f"https://api.themoviedb.org/3/person/{person_id}/movie_credits"

    # Prepare query parameters
    params = {"language": language}

    # Make the API request
    response = requests.get(url, headers=headers, params=params)
    response.raise_for_status()
    return response.json()

def get_directed_movie_count(director_name: str) -> int:
    '''
    Get the number of movies directed by a specific director.

    Parameters:
        director_name (str): The full name of the director whose movies to count.

    Returns:
        int: The number of movies directed by the specified director.
    '''
    # Step 1. Search for the director "Sofia Coppola" using the /3/search/person API to find her person_id.
    person_search_response = search_person(query=director_name)

    # Get person_id from response
    person_id = (
        person_search_response.get("results")[0]["id"]
        if person_search_response and person_search_response.get("results")
        else None
    )

    # Step 2. Retrieve the movie credits for Sofia Coppola using her person_id with the /3/person/{person_id}/movie_credits API.
    if person_id:
        movie_credits_response = get_person_movie_credits(person_id)

        # Step 3. Count the number of movies in the credits retrieved.
        # Filter for movies that Sofia Coppola directed
        directed_movies = [
            movie
            for movie in movie_credits_response["crew"]
            if movie["job"] == "Director"
        ]

        number_of_movies_directed = len(directed_movies)
        return number_of_movies_directed

    return 0

if __name__ == "__main__":
    number_of_movies_directed = get_directed_movie_count(director_name="Sofia Coppola")
    print("Number of movies directed by Sofia Coppola:", number_of_movies_directed)
```

Figure 8: An code example of the main function F generated by ToolCoder for the query: *give me the number of movies directed by Sofia Coppola*. This step implements the actual functionality of all API calls in the call_api placeholders in Figure 7 according to their API documentations. Due to space limitations, the docstrings of some functions have been omitted.

Prompt for Task to Code Module $\mathcal{M}_{T2C}$

```
You are a coding assistant specialized in converting task requirements into precise Python
function definitions. Your role is to define a function that fully meets the task's objectives
by specifying a clear function name, parameters, return type, and docstring.

### Instructions:
For each given task, carefully follow these steps to create a well-structured function
definition:
1. Function Name: Choose a concise, meaningful function name that accurately reflects the
task's purpose.
2. Parameters: Identify and define the essential input(s) the function requires. For each
parameter:
 - Use a descriptive name and include a type annotation.
3. Return Type: Specify an appropriate return type (e.g., 'int', 'str', 'dict', 'list') that
best represents the function's output.
4. Docstring: Write a detailed, clear docstring that includes:
 - A brief description of the function's purpose.
 - Descriptions for each parameter, detailing its type and role.
 - Explanation of the return value, with type and a brief on what it represents.
 - Explicit Notes on using 'call_api', with the following rules:
   - Each 'call_api' invocation must use an 'api_path' from the provided toolbox; do not invent
or alter paths.
   - Place all request parameters within the 'params' dictionary; do not format 'api_path'
using string interpolation or formatted strings.
   - Example: use 'call_api(api_path="/3/movie/{movie_id}/credits", params={"movie_id":
movie_id})' instead of 'call_api(api_path=f"/3/movie/movie_id/credits", params={})'.
   - Refer to the example functions below for guidance on structuring this.
 - Use triple single quotes for the docstring, as shown below.
5. Function Body: Leave the function body empty. Only include the function definition,
parameters, and docstring in your output.

Now, let's begin!
You are given a question: {question}
Python Function:
```

Figure 9: The prompt used for implementing the task-to-code function module $\mathcal{M}_{T2C}$ in the RestBench-TMDB dataset during our experiments.

Prompt for Subtask Planning Module $\mathcal{M}_{TP}$

```
Your objective is to analyze the user's complex question and create a breakdown of actionable
subtasks, identifying the most appropriate APIs from the provided toolbox to address each
subtask in sequence.

### Available Tools:
{toolbox}

Each API includes a request path and a description of its functionality. Utilize these APIs
strategically to decompose and solve the user's question according to the following guidelines:

### Guidelines
1. Clarify Requirements: Carefully read the user's question to determine key requirements,
objectives, and expected outcomes. Identify any specific data or information you need to gather
to satisfy the request.
2. Break Down the Task: Divide the complex task into clear, manageable subtasks that each
address part of the user's needs and can be fulfilled using one or more APIs.
3. Select Relevant APIs: Match each subtask to the most relevant API(s) from the toolbox. Base
your choices on each API's functionality and response format to ensure it provides the required
information.
4. Handle Dependencies: Check for dependencies between APIs. For example, to access
'/3/movie/{movie_id}/credits', the 'movie_id' must first be retrieved via a compatible API.
5. Output in Pseudo-Code: Write your solution as a Python function using pseudo-code, with each
step annotated in comments (denote as Step 1 to N) describing the subtask. Do not write any
code implementation for the steps—only document the subtasks in comments.

User's Question: {question}
Pseudo-Code Task: {pseudo_code_task}
Planned Subtasks:
```

Figure 10: The prompt used for implementing the subtask planning module $\mathcal{M}_{TP}$ in the RestBench-TMDB dataset during our experiments.

```
Your task is to help the user select appropriate APIs from the provided toolbox and complete
the function template to solve the problem accurately.

### Instructions
1. Input Provided:
 - Problem Description: A specific question or request from the user (e.g., "give me the number
of movies directed by Sofia Coppola").
 - Pseudo-Code Task Template: A function template with the following components:
   - Function signature, parameters, return type, and a functional description.
   - Structured subtask comments directly embedded in the pseudo-code template, outlining each
planned subtask step-by-step.

2. Expected Output:
 - Based on the problem description and the structured action plan provided, complete the
pseudo-code by selecting and integrating suitable API calls from the provided candidate toolbox.
 - Use the placeholder 'call_api(api_path, params)' for each API call, where:
   - 'api_path' strictly matches a valid, existing API path from the provided candidate toolbox.
Do not create or assume any non-existent API paths. No 'api_path' outside this toolbox should
be used or inferred.
   - Place all request parameters within the 'params' dictionary; do not format 'api_path'
using string interpolation or formatted strings.
   - Example:  use 'call_api(api_path="/3/movie/{movie_id}/credits",  params={"movie_id":
movie_id})' instead of 'call_api(api_path=f"/3/movie/movie_id/credits", params={})'.

3. Guidelines for API Selection:
 - Do not provide a direct answer to the problem. Instead, fill in the pseudo-code template to
enable successful execution.
 - Follow these steps for each 'call_api' integration:
   - Identify any necessary preliminary calls (e.g., fetching an entity ID).
   - Ensure each step logically contributes to the function's purpose, using helper variables
and conditional checks where needed.
   - Keep pseudo-code succinct, adding comments that clarify the role of each API call.

### Provided API toolbox:
{toolbox}

Let's begin!
Question: {question}
Pseudo-Code Task: {pseudo_code_task}
Please complete the pseudo-code solution:
```

Figure 11: The prompt used for implementing the tool selection module $\mathcal{M}_{TS}$ in the RestBench-TMDB dataset during our experiments.

Figure 12: The prompt utilized for implementing the code generation module $\mathcal{M}_{CG}$ in the RestBench-TMDB dataset for our experiments.