Reverse Chain: A Generic-Rule for LLMs to Master Multi-API Planning

Yinger Zhang[*]
Zhejiang University
zhangyinger@zju.edu.cn

Hui Cai[*†]
Ant Group
biyu.ch@antgroup.com

Xierui Song
Ant Group
songxierui.sxr@antgroup.com

Yicheng Chen
Ant Group
yicheng.chen@antgroup.com

Rui Sun
Ant Group
zhengxi.sr@antgroup.com

Jing Zheng
Ant Group
jing.zheng@antgroup.com

Abstract

While enabling large language models to implement function calling (known as APIs) can greatly enhance the performance of Large Language Models (LLMs), function calling is still a challenging task due to the complicated relations between different APIs, especially in a context-learning setting without fine-tuning. This paper introduces "Reverse Chain", a controllable, target-driven approach designed to empower LLMs with the capability to operate external APIs only via prompts. Recognizing that most LLMs have limited tool-use capabilities, Reverse Chain limits LLMs to executing simple tasks, e.g., API Selection and Argument Completion. Furthermore, to manage a controllable multi-function calling, Reverse Chain adopts a generic rule-based on a backward reasoning process. This rule determines when to do API selection or Argument completion. To evaluate the multi-tool-use capability of LLMs, we have released a compositional multi-tool task dataset, available at https://github.com/zhangyingerjelly/reverse-chain. Extensive numerical experiments validate the remarkable proficiency of Reverse Chain in managing multiple API calls.

1 Introduction

Recently, there has been an impressive wave in the progress made in Large Language Models (LLMs), due to their excellent performance in a variety of tasks (Chowdhery et al., 2022; Brown et al., 2020; Scao et al., 2022; Wei et al., 2022a; Bubeck et al., 2023). However, LLMs still face difficulties with some specialized tasks due to their fundamental limitation on the information they stored and learned, which can become outdated and may not be suitable for all applications. A practical solution is to augment LLMs with external tools (known as APIs). In this setup, LLMs act as controllers, not only to understand user intents but crucially to select and orchestrate the appropriate tools to complete tasks.

Unfortunately, LLMs still lack the sophistication to fully understand human instructions and effectively implement function calling. Many works are dedicated to enhancing the function calling abilities of LLMs through fine-tuning or in-context learning methods (Patil et al., 2023; Qin et al., 2023; Schick et al., 2023; Tang et al., 2023; Parisi et al., 2022; Li et al., 2023; Liang et al., 2023; Song et al., 2023; Xu et al., 2023). Compared to fine-tuning, in-context learning approaches offer a more straightforward and scalable solution, as they eliminate the need to train an entirely new model for each new API. Consequently, the primary goal of this paper is to enhance the API planning capabilities of LLMs within the in-context learning setting.

Different from the aforementioned studies which focus on simpler tasks, such as single-tool task or independent multi-tool task (detailed in Table 1), this paper targets at enhancing LLMs' ability to handle more complicated **compositional multi-tool task** (detailed in Table 1). Implementations of this task requires to employ multiple, potentially interdependent APIs, which is common in real-world scenarios but poses a greater challenge in API planning for LLMs. It's worth noting that single-tool task and independent multi-tool task can be seen as subsets of compositional multi-tool task, and the proposed approach can also manage them with minimal modifications. The generalizability of the proposed method to different task types will be discussed in the Section 5.

In the realm of tool-use, various prompting techniques have been explored. **One-step planning** algorithms are introduced in (Shen et al., 2023; Liang et al., 2023), but its accuracy is often low in complex, ambiguous scenarios. The Chain of Thought (CoT) approach (Wei et al., 2022b) counters this by

[*] Equal Contributions
[†] this author is the corresponding author

Task Type	Example	API planning
Single-tool	What's the weather in New York ?	*getWearther*(city='New York')
Independent multi-tool	What's the weather in New York? When's my next meeting?	*getWearther*(city='New York') *showCalendar*(event='next meeting')
Compositional multi-tool	I'm Lucas, Could you find a flight and book it to my destination ?	*BookFlight*(flight_ID=*FindFlight*(destination =*GetUserDestination*(userName='Lucas'))

Table 1: Different task types, classified by the number of required tools and their dependencies for task execution.

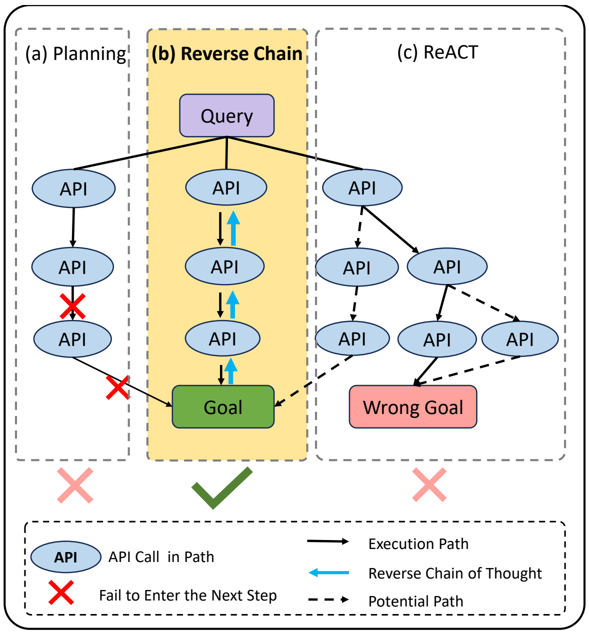


Figure 1: A comparison of our Reverse Chain with the one-step/CoT Planning and ReAct for multi-API planning.

step-by-step planning with intermediate reasoning. Known as **CoT planning**, this technique decomposes tasks into several simpler sub-tasks, thereby boosting reasoning and accuracy. Nevertheless, as illustrated in Figure 1 (a), a limitation of these planning methods is their potential for errors in the intermediate stages. While the final step of the plan is intended to achieve the ultimate goal, errors in the intermediate planning steps can lead to execution failures. For instance, as illustrated in the compositional multi-tool case of Table 1, if the value of 'destination' parameter is parsed incorrectly, e.g., destination = 'None', it is obvious that *BookFlight* could not be executed successfully. To bridge this gap, **ReAct**, as described by (Yao et al., 2022), refines reasoning by combining actions and observations for deeper insights. Expanding on this, tool-learning projects (Song et al., 2023; Ruan et al., 2023) utilize the output from each step to inform the next decision. However, as depicted in Figure 1(c), in the multi-function call scenarios, ReAct, despite successfully executing each step, may not adhere to the correct reasoning path towards the final goal, as a result, it deviates to the wrong destination and may end up early. For instance, in the previously mentioned scenario, the ReAct execution flow would be: *GetUserDestination* (userName='Lucas') -> destination, flight_ID = *FindFlight* (destination) -> Final Answer, which is not completed since the last API *BookFlight* has not been executed.

In summary, both one-step/CoT planning and ReAct are forward reasoning solutions, so they encounter similar control challenges: each step exhibits a high level of **unpredictability** and **uncertainty**, especially at the beginning when the search space is large. Errors can propagate from a wrong thought or action, leading to incorrect solution paths or final goals. This issue arises because these methods start from scratch and progress forward towards the final target, with the LLM bearing the entire burden of planning.

To address these issues, we propose a controllable yet general framework called **Reverse Chain**. This framework consists of a generic rule and two key modules: API Selection and Argument Completion, both centered on prompting an LLM. Specifically, the generic rule in Reverse Chain performs a multi-API planning task in a **backward manner**: it starts by selecting the final API for a task, and then completes the required arguments, drawing values from the query and context, or by outputs of other APIs. When a new API is selected during the argument completion stage, this process repeats. The procedure continues iteratively until all arguments of all APIs are filled. Reverse Chain distinguishes itself from previous work with the following three main advantages: 1. **Backward reasoning**, starting from the final goal, preventing planning from deviating into a wrong direction, thus ensuring the correctness of the final goal. 2. The **step-by-step decomposition** dominated by the rule makes the process controllable, with each stage being forward-executable, effectively avoiding errors such as incorrect intermediate stage. 3. The **tasks of LLMs are simplified** to just selecting APIs and filling arguments, avoiding complex planning. This strategy effectively utilizes the strengths and capabilities of the existing LLMs without depending on extensive reasoning abilities.

In summary, the **contributions** of this paper are:

1. This paper presents Reverse Chain, a straightforward framework to improve the API planning capabilities of LLMs in an in-context-learning setting. By employing a backward reasoning scheme and a step-by-step problem-solving methodology, the process becomes more manageable and controllable.

2. This paper focuses on API planning for compositional multi-tool task. To assess the capabilities of LLMs in handling such tasks, we build a high-quality dataset containing 825 APIs and 1550 instances for that task, constructed automatically using GPT-4 (OpenAI, 2023). Additionally, an automatic evaluator powered by GPT-4 is also developed for efficient evaluation purpose.

3. Extensive experiments are conducted to demonstrate the superiority of the Reverse Chain approach in multi-API calling tasks, surpassing the state-of-the-art in-context learning approaches, e.g., CoT and ReAct.

2 Related Work

Tool Learning The discussion of tool usage in LLMs has grown significantly, with models like Toolformer leading the way (Schick et al., 2023; Nakano et al., 2021). Current approaches can be divided into two categories. The first category focuses on enhancing the tool-specific capabilities of language models through fine-tuning with specialized datasets (Patil et al., 2023; Qin et al., 2023; Schick et al., 2023; Tang et al., 2023; Parisi et al., 2022; Yang et al., 2023; Qian et al., 2023). The second category directly leverages the capabilities of LLMs, prompting them to interact with various tools, ranging from AI models (Shen et al., 2023; Wu et al., 2023) to more versatile tool sets (Li et al., 2023; Liang et al., 2023; Song et al., 2023; Xu et al., 2023). Generally, the prompting approach is simpler and more scalable, but it still has a significant gap compared to fine-tuning method, so this work is proposed to enhance the API planning capability of prompting methods. It is notable that while the previously mentioned studies introduced numerous tool-learning datasets, they primarily encompass relatively simple tasks, focusing on single-tool task or independent multi-tool task. In contrast, this paper targets a more complex task called compositional task, where multiple dependent APIs are needed.

Prompting LLMs Various methods, like CoT (Wei et al., 2022b) for task decomposition and ReAct (Yao et al., 2022) for melding reasoning with action, enhance general prompting capabilities. Additionally, numerous planning methods are tailored for tool-use. (Shen et al., 2023; Liang et al., 2023) start by generating a direct solution outline, followed by selecting and executing relevant APIs. DFSDT (Qin et al., 2023) can be seen as an improved version of ReAct, enables LLMs to evaluate different reasoning paths and select the most promising one. RestGPT's (Song et al., 2023) workflow involves an iterative "plan and execute" cycle. Meanwhile, (Ruan et al., 2023) employs a sequential planning approach, feeding the outcome of each step into the subsequent one. All these works require an LLM to perform either full or step-by-step planning based on the task. However, the Reverse Chain proposed in this work simplifies this by having the LLM focus on just two tasks: API selection and argument completion, thereby greatly simplifying the task complexity. Furthermore, Unlike previous methods that progress from scratch to the final goal, Reverse Chain starts from the end goal and reasons backwards, enhancing controllability.

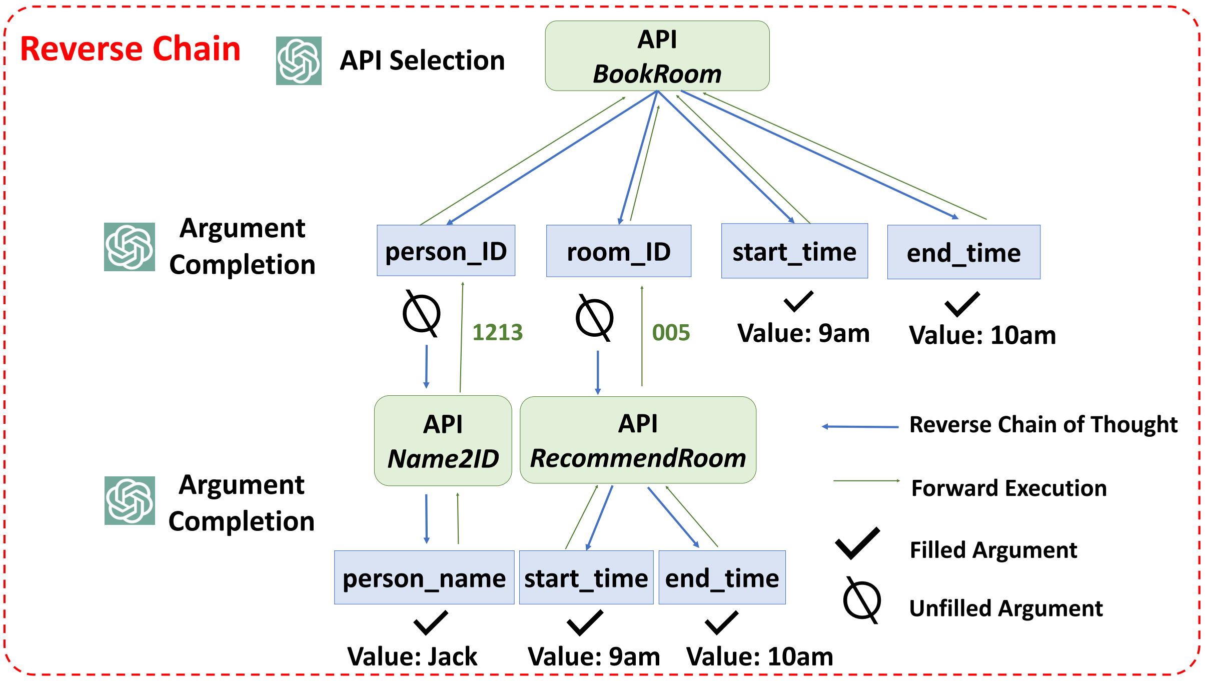
3 Reverse Chain: A Multi-API Planning Approach

The objective of this work is to generate effective API planning based on user queries and API candidates. Figure 2 provides a detailed example: A user query could be a natural language request like "Please help Jack book a meeting room from 9:00 am to 10:00 am". Each API in the API pool is characterized by its description, arguments, and output. e.g., the API *RecommendRoom* has a functionality description of "Recommend the ID of an available meeting room", arguments "start_time" and "end_time", and an output of "room_ID". A successful API planning consists of two parts: selecting the proper API and filling in all the argu-

User Query Please help Jack book a meeting room for 9am-10am

API Pool

API	Description	Arguments	Output
Name2ID	Convert user name to user ID	person_name	person_ID
RecommendRoom	Recommend the ID of an available meeting room	start_time, end_time	room_ID
BookRoom	Book a meeting room	person_ID, room_ID start_time, end_time	room_Info



API Planning *BookRoom* (person ID = *Name2ID*(person name='Jack'),
room ID = *RecommendRoom* (start time='9am', end time='10am'),
start time = '9am', end time='10am'

Figure 2: Workflow of Reverse Chain on an example.

ments correctly, where the argument values can come from the query or context, or from the output of another API.

Section 3.1 outlines the Reverse Chain process, while Section 3.2 specifically discusses the two modules that interact with LLM: **API Selection** and **Argument Completion**.

3.1 Reverse Chaining

Different from CoT and ReAct, Reverse Chain performs a task decomposition in a reverse manner, and its step-by-step problem-solving path is predefined by a generic rule. It is notable that this generic rule is not restricted with a certain type of tasks.

Figure 2 shows an example of Reverse Chain applied to API planning for a query. Initially, Reverse Chain selects the final API for a given task, this step is referred to as **API Selection**. In this example, LLM selects an API named *BookRoom* to match the task "booking a meeting room". Next, the required arguments of the selected API are identified through engineering guidance, e.g. API *BookRoom* has four required arguments, that is, person_ID, room_ID, start_time, and end_time. There are three possible approaches for arguments filling, and we define this process as **Argument Completion**:
Case 1. The argument value extracted directly from the context and user query, e.g., start_time = 9:00 am;
Case 2. When the argument value could not be obtained directly, Reverse Chain searches for another possible API whose output could complete the missing argument, e.g., the argument person_ID could be obtained from API *Name2ID*;
Case 3. If it is unable to obtain the argument value

from the above two cases, the generic rule will request the argument value directly from the user.

For the selected internal APIs in **Case 2**, Reverse Chain makes recursive calls to complete the required arguments of these APIs, e.g., the required argument of *Name2ID* is person_name, and the value 'Jack' could be obtained through **Case 1** in Argument Completion. The algorithm continues until the termination condition is met, i.e., all of the required arguments are completed. Finally, when all required arguments of an API are filled, the API is ready to be executed forward to complete the given task.

3.2 LLM Modules in Reverse Chain

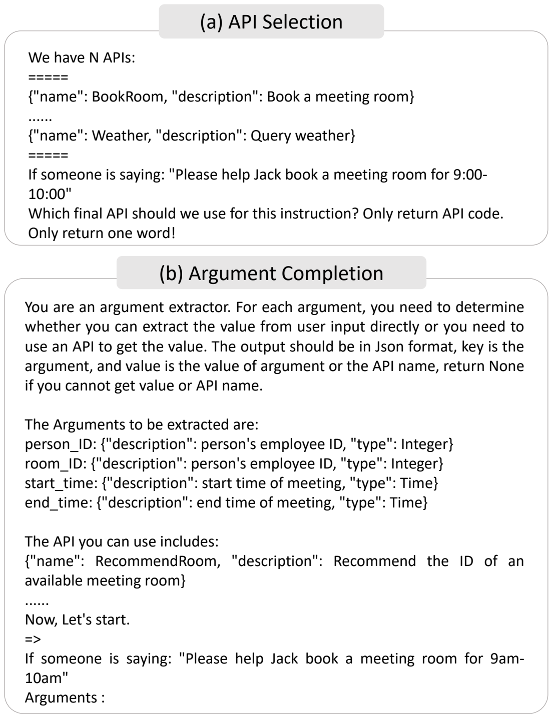
3.2.1 API Selection

In this module, the LLM effectively determines the relevant API by analyzing the task descriptions and API candidates. The specific prompt used in this module is depicted in Figure 3.(a). Within the Reverse Chain, the API Selection module is employed in two different scenarios, separated with regard to different task description and API candidates. The first scenario occurs when selecting the ultimate API. In this case, the task descriptions correspond to the user query and the API candidates refers to all APIs in the API Pool. The second scenario occurs as a sub-module of Argument Completion. When the value of an argument cannot be obtained from the user query or context, the Reverse Chain selects an appropriate API whose output can fulfill the missing argument. In such cases, the task descriptions refers to the description of the unfilled argument. The scope of API candidates can be narrowed down through variable type matching, which encompasses Time, Date, String, etc. This capability facilitates a more refined selection process, leading to a improved accuracy.

3.2.2 Argument Completion

After API Selection, the required arguments for the selected API are determined with the help of engineering guidance. In this module, the LLM is leveraged to complete these arguments using information from the query, context and API candidates. The execution follows three possible outcomes:
Case 1 The argument value is directly extracted from the context or user query.
Case 2 Another API is used to complete the missing argument value, indicating that the LLM is unable to obtain the argument value directly. It should be noted that the arguments of this new internal API must be completed before execution.
Case 3 None, indicating the inability to obtain the argument value from the context, user query, and potential API output. In this case, the generic rule will request the argument value directly from the user.

Specific optimizations have been applied to the aforementioned approach, which are further explored in Section 4.2.2. The optimized prompt used in this module is illustrated in Figure 3.(b).


(a) API Selection

```
We have N APIs:
=====
{"name": BookRoom, "description": Book a meeting room}
......
{"name": Weather, "description": Query weather}
=====
If someone is saying: "Please help Jack book a meeting room for 9:00-
10:00"
Which final API should we use for this instruction? Only return API code.
Only return one word!
```

(b) Argument Completion

```
You are an argument extractor. For each argument, you need to determine
whether you can extract the value from user input directly or you need to
use an API to get the value. The output should be in Json format, key is the
argument, and value is the value of argument or the API name, return None
if you cannot get value or API name.

The Arguments to be extracted are:
person_ID: {"description": person's employee ID, "type": Integer}
room_ID: {"description": person's employee ID, "type": Integer}
start_time: {"description": start time of meeting, "type": Time}
end_time: {"description": end time of meeting, "type": Time}

The API you can use includes:
{"name": RecommendRoom, "description": Recommend the ID of an
available meeting room}
......
Now, Let's start.
=>
If someone is saying: "Please help Jack book a meeting room for 9am-
10am"
Arguments :
```

Figure 3: The details of prompts used in Reverse Chain for API Selection and Argument Completion (when LLM is chatgpt).

4 Experiments

In this section, extensive experiments are conducted to investigate the performance of Reverse Chain. We start with generating an evaluation dataset automatically, benchmarking different in-context learning methods on function calling and defining the evaluation metrics. In Section 4.1, to benchmark Reverse Chain, we compare its API planning capabilities with the current state-of-the-art in-context learning solutions on ChatGPT. Section 4.2, details a set of ablation experiments designed to elucidate the underlying principles of Reverse Chain. Finally, Section 4.3 analyzes the factors contributing to the effectiveness of Reverse Chain.

Dataset We construct a dataset for evaluating compositional multi-tool tasks. Guided by the self-instruct paradigm (Wang et al., 2022), this dataset is generated automatically based on GPT-4 and ChatGPT (gpt-3.5-turbo), involving the following steps:

1. Initially, APIs are selected from public repositories, including API-Bank (Li et al., 2023) and public-apis. We then manually create 20 diverse seed examples for compositional multi-tool task, each comprising three components: {API and its description, User query, System response}. A specific seed example is detailed in Figure 4 in Appendix A.1 .

2. These seed instances serve as in-context examples for GPT-4, so as to generating more complex new samples. The prompts for GPT-4 are detailed in Figure 6 in Appendix A.2, include a general description of the task, a randomly chosen seed example, and a prescribed response format. Then we conduct manual quality checks to filter out erroneous samples, achieving a 50% filtration rate. The high-quality samples produced are used as new seed examples for further data collection, repeating the process multiple times. To enhance dataset diversity, GPT-4's temperature is set at 0.8.

3. Additionally, we employ ChatGPT to enhance API information and uniformly standardize the samples into a JSON format. A detailed example is in the Figure 5 in Appendix A.1. Each sample includes fields: {APIs, Query, Label}, with each API in APIs represented as a JSON object with fields: {name, description, arguments, output, format}. Notably, the fields {arguments, output, and format} are generated by leveraging existing information. The prompt for this is outlined in Figure 7 Appendix A.2.

It's worth mentioning that the dataset comprises 825 unique APIs across 20 categories, totaling 1550 labeled instances, with the categories detailed in Table 7 in Appendix A.1. Focused on compositional multi-tool tasks, the samples are classified into three levels based on API nesting complexity: Level-1, two levels of API nesting, containing 798 instances; Level-2, three levels of API nesting, containing 693 instances; and Level-3, more than four levels of API nesting, containing 59 instances. Each Instance has an average of 2.93 function calls.

It is clear that this synthetic dataset is suitable for evaluation since: 1. Automated data generation guarantees unbiased data; 2. The APIs are spread across diverse domains, accurately reflecting real-world situations; 3. The inclusion of various nesting levels in compositional multi-tool tasks ensures a rich diversity.

Baseline To benchmark Reverse Chain, we measure its performance against five other in-context learning methods: **Zero-Shot**, **Few-Shot**, **Zero-Shot-CoT**, **Few-Shot-CoT**, and **ReAct**, using ChatGPT as the underlying LLM. Each method integrates API data into the prompt, utilizing the LLM's in-context learning for API planning. The Zero-Shot approach uses API information and user queries in the prompt, Few-Shot adds extra examples to prompt. Zero-Shot-CoT includes step-by-step instructions, while Few-Shot-CoT adds explanations to these steps in the examples. ReAct, implemented via the langchain framework, uses a (thought, action, observation) format for task execution. Examples of prompts for these methods can be found in the Appendix A.3. Experiments are conducted on GPT-3.5-turbo at the gpt-3.5-turbo-0301 checkpoint with the temperature set to 0.1.

Metrics We use accuracy as a metric to evaluate API planning, which consists of two aspects: API name and API arguments. The value of argument consists of direct value filling or another API calling.

Given the diversity of output formats across solutions, we rule out simple string matching due to its inefficiency and manual annotation for its time-consuming nature. Instead, we craft an efficient automated evaluator using GPT-4. Tailored prompts are designed for each baseline method to match its output characteristics. The prompts are presented in Appendix A.4. We manually test 200 samples, comparing human annotations with GPT-4 evaluations, and discover that the GPT-4 evaluator exhibits a strong 89% correlation with human assessments.

4.1 Main Results

Throughout the experiments, the given API candidates set in prompt only includes the needed APIs for a given task since the focus of this paper is primarily on evaluating the capability of LLMs on generating a proper API calling rather than the retrieval

https://github.com/public-apis/public-apis

Method	level 1	level 2	level 3	Overall
Zero-Shot	72.06	67.68	42.37	68.97
Few-Shot	86.46	77.48	71.18	81.87
Zero-Shot-CoT	82.45	81.38	57.62	81.29
Few-Shot-CoT	89.72	85.71	66.10	87.16
ReAct	72.68	69.11	45.76	70.06
Reverse Chain	**93.99**	**90.33**	**86.44**	**92.06**

Table 2: Evaluation results on various in-context learning methods. We can observe that the proposed Reverse Chain outperforms all other approaches.

of API. Table 2 compares the accuracy of different in-context learning methods. Under a **Zero-Shot** setting, the LLM's API planning accuracy stands at approximately 68.97%. Although **Few-Shot** methods raises this to 81.87%, the addition of Chains of Thought (CoT) further elevates performance to 87.16% in **Few-Shot-CoT**, which indicates the benefit of decomposing complex tasks. The **ReAct** strategy, with its reasoning-action-observation approach, also improves upon the zero-shot method. However, the standout performer is the **Reverse Chain** method, which surpasses all others by simplifying the multi-API calling problem into two easier tasks (API Selection and Argument Completion) and adopting a target-driven approach, thereby minimizing uncertainty. Impressively, **Reverse Chain** achieves superior results even in a zero-shot context surpassing both the **Few-Shot-CoT** and **Few-Shot** methods. Additionally, Table 2 displays results across different levels of API planning where higher levels indicates greater difficulty. As expected, all methods exhibit increased error rates as the complexity of API planning escalates. In these more challenging scenarios, the Reverse Chain approach demonstrates a more pronounced improvement compared to other methods. This significant gap underscores its robustness and effectiveness in handling complex multi-API calling tasks.

4.2 Ablation Study

In this section, we mainly focus on exploring the impact of creativity of LLMs and different argument completion strategies on the performance of Reverse Chain. The experiments are conducted on GPT-3.5-turbo.

4.2.1 Creativity and imagination of LLMs on Reverse Chain

We first investigate the impact of LLM's temperature on Reverse Chain. Temperature controls the randomness of the LLM's output. A lower temperature results in more focused and deterministic responses, while a higher temperature generates more diverse and creative answers. Table 3 shows that Reverse Chain performs better at lower temperatures, with accuracy decreasing when it seeks more creative responses. It makes sense as we require LLM to make rational and accurate decisions.

Method	level 1	level 2	level 3	Overall
T=0.1	**93.99**	**90.33**	**86.44**	**92.06**
T=0.5	78.45	59.88	59.32	69.42
T=1	69.80	50.50	49.15	60.39

Table 3: The impact of different temperatures of LLMs on the performance of Reverse Chain. T represents the temperature of ChatGPT

4.2.2 Argument Completion Optimization

Reverse Chain	**92.06**
Reverse Chain_one-by-one	74.19
Reverse Chain_three-step	38.71

Table 4: Ablation study for the design of Argument Completion in Reverse Chain.

In this part, a series of ablation studies are performed to examine various optimizations during the development of the Reverse Chain Algorithm. The optimizations discussed there primarily concentrate on the stage Argument Completion.

Reverse Chain_one-by-one In the existing Reverse Chain method, LLMs simultaneously extracts all argument results. An alternative strategy involves processing each argument completion sequentially, a method we term Reverse Chain_one-by-one. For instance, the API *FlightBooking* has two arguments: departure_point and destination. While the standard Reverse Chain completes both departure_point and destination arguments concurrently, Reverse Chain_one-by-one first fills the argument departure_point, followed by the destination.

Table 4 shows that Reverse Chain achieves a 92.06% accuracy, surpassing Reverse Chain_one-by-one's 74.19%. The performance disparity arises because the LLM in Reverse Chain can access all information about unfilled arguments during the argument completion process. This comprehensive insight enables more precise and accurate argument filling. Consider the API example *FlightBooking* with the user query: "help me book a flight from

London to Los Angeles", Table 5 demonstrates that in Reverse Chain_one-by-one, both arguments mistakenly extract the value 'London', as the LLM interprets the query's location as the destination. Conversely, Reverse Chain, recognizing two separate arguments for departure_point and destination, accurately distinguishes between the two locations in the query.

In addition to its superior performance, Reverse Chain is also more efficient in terms of time and computational resources since it only requires one interaction with the LLM.

	departure	destination
One-by-one	London	**London (wrong)**
Reverse Chain	London	Los Angeles

Table 5: Examples of Reverse Chain_one-by-one and Reverse Chain

Reverse Chain_three-step Here is an example: user query is "help Jack book a meeting room", requiring the filling of the person_ID argument for the API *BookRoom*. In the Argument Completion step of standard Reverse Chain, both the query and API candidate sets are available to the LLM, enabling direct value extraction from the query or API selection. However, in the Reverse Chain_three-step setting, argument completion is further split into two steps: initially, the LLM is given only the query for value extraction, potentially returning the extracted value or 'None'. If 'None' is returned, then it will move to API selection, choosing from the API candidate set.

Table 4 reveals that Reverse Chain_three-step attains just a 38.71% accuracy rate. This is mainly due to the absence of API information during the value extraction step, often leading to forced extraction of incorrect values even when certainty is low. In the given example, the LLM mistakenly identifies 'Jack' as the person_ID value. This confusion is not surprising given the vague nature of the person_id concept. However, with API information, the LLM can discern between using APIs or forcibly extracting values, thus enhancing accuracy. For instance, the LLM might find that person_ID is retrievable through the API PersonName2ID, and consequently, it disregards the erroneously extracted 'Jack'.

	Wrong Final Tool	Wrong Argument
Zero-Shot	33	132
Few-Shot	29	75
Zero-Shot-CoT	36	68
Few-Shot-CoT	22	58
ReAct	91	70
Reverse Chain	**20**	**40**

Table 6: Error cause statistics all methods.

4.3 Why Reverse Chain works?

In this section, we dissect common errors in API planning and illustrate how the Reverse Chain method mitigates them for improved results. We categorize the errors, identify through manual review, into two primary types, **Wrong Final Tool** and **Wrong Argument**, detailed in Table 6. This statistics is done on 500 randomly sampled instances.

Wrong Final Tool arises when the final API is missing, leading to incorrect API termination and incomplete instructions. This error is prevalent across all comparison methods due to their tendency to plan from the scratch, increases the likelihood of deviating from the final goal. Particularly, ReAct is more susceptible to this mistake because of its thought-action-observation approach that lacks global planning. Reverse Chain, by planning based on the final goal, minimizes this error, except when the query's ultimate intention is ambiguous.

The second error, **Wrong Argument**, predominates in planning methods, can be further categorized into **Wrong Argument_API** and **Wrong Argument_Value**. Wrong Argument_API error occurs when a required argument is the output of another API, but the predicted result bypasses this API, filling in an incorrect value. For instance, the correct argument is person_ID = *PersonName2ID* (name='Jack'), but the prediction inaccurately inputs person_ID='Jack'. This error often results from mistakes in the intermediate planning steps. In Reverse Chain's argument completion phase, using the optimization approach from Section 4.2.2, these errors can be greatly reduced, which allows the LLM to choose between using the API or extracting the argument value. **Wrong Argument_Value** involves extracting incorrect values for the argument. Specific cases and optimization strategies for Reverse Chain are discussed discussed in Section 4.2.2.

5 Conclusion

This paper proposed Reverse Chain, a concise, target-driven approach developed to empower LLMs with the capability to interact with external APIs in an in-context learning setting. By implementing a backward reasoning strategy and a generic rule, Reverse Chain effectively broke down complex function-calling challenges into two fundamental tasks for LLMs: API selection and argument completion. Additionally, we collected a compositional multi-tool dataset for evaluation. Extensive experiments revealed that Reverse Chain markably enhances the tool-use proficiency of the existing LLM ChatGPT, achieving superior performance compared to methods like CoT and ReAct.

Although the current work concentrates on compositional multi-tool tasks, it can also be easily extended to other types of tasks. For instance, in the case of independent multi-tool tasks, after identifying sub-intents at the beginning of the task (known as Intent Detection, a well-established problem in NLP with numerous robust solutions), we could employ the reverse chain process for each identified sub-task separately.

6 Limitations

We identify some limitations with our current work that can be addressed in future work.

- The tasks/datasets in this work assume a sequential execution of APIs, Reverse chain cannot deal with branching ("if ... then ... else ...") or looping ("while ... do/check ...") situation, both of which are important cases in multi-API planning.
- The in-context learning approach generally struggles with handling a large number of API candidates due to length limitations. A solution similar to the one in (Qin et al., 2023), which involves adding a retrieval module at the beginning of the pipeline, can be adopted.
- While our demonstration shows that Reverse Chain surpasses other in-context learning methods in performance, it does require more calls to the LLM. This highlights a trade-off between performance enhancement and increased computational resource use.
- The API in the dataset is fake and it is supposed the function is called successfully. However, in reality, API calls often fail, thus multiple calls are required, so there is a gap between the simulation and the real world.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901.

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311.

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

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. arXiv preprint arXiv:2303.16434.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. arXiv preprint arXiv:2112.09332.

OpenAI. 2023. Gpt-4 technical report. ArXiv, abs/2303.08774.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. arXiv preprint arXiv:2205.12255.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. arXiv preprint arXiv:2305.15334.

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.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. arXiv preprint arXiv:2307.16789.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Xingyu Zeng, and Rui Zhao. 2023. Tptu: Task planning and tool usage of large language model-based ai agents. arXiv preprint arXiv:2308.03427.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. arXiv preprint arXiv:2211.05100.

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. arXiv preprint arXiv:2302.04761.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. arXiv preprint arXiv:2303.17580.

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. arXiv preprint arXiv: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. arXiv preprint arXiv:2306.05301.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. arXiv preprint arXiv:2212.10560.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022a. Emergent abilities of large language models. arXiv preprint arXiv:2206.07682.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35:24824–24837.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. arXiv preprint arXiv:2303.04671.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. On the tool manipulation capability of open-source large language models. arXiv preprint arXiv:2305.16504.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction. arXiv preprint arXiv:2305.18752.

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

A Appendix

A.1 Sample in dataset

In this section, we show the details of the dataset. Figure 4 is an example among the 20 diverse seed examples designed by human. Figure 5 is an example in the dataset of final version. The category and examples of APIs are listed Table 7.

A.2 Prompts for dataset construction

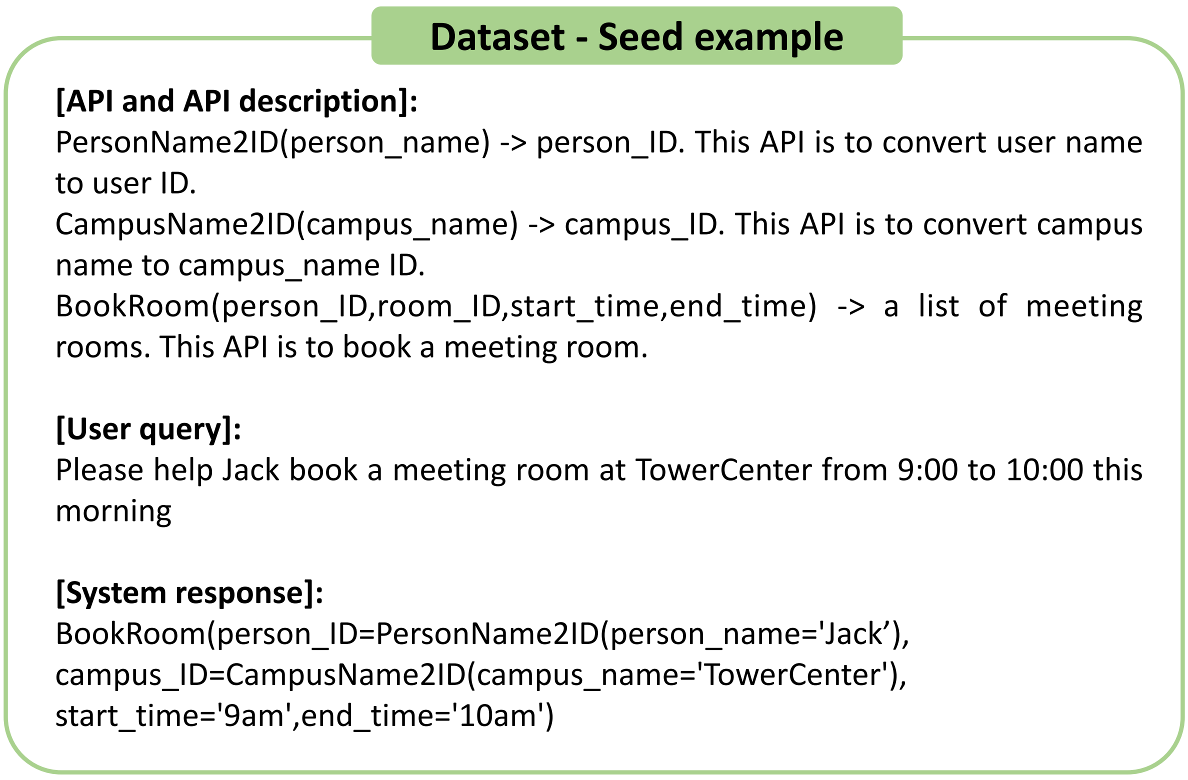
In this section, we show the details of prompt templates in data construction. Figure 6 is the prompt of new sample generation for GPT-4. Figure 7 is the prompt of format conversion for ChatGPT.

A.3 Prompts for baseline methods

The prompt for baseline methods are listed in Figure 8, Figure 9, Figure 10 and Figure 11.

A.4 Prompts for evaluation

Following the evaluation method used by (Tang et al., 2023), We use GPT-4 as our evaluator. The evaluation prompts for different methods are shown in Figure 12, 13, 14, 15,16,17. It should be noted that prior to conducting the ReAct evaluation, it is necessary to preprocess the answer to extract the function callings.



Dataset - Seed example

[API and API description]:
PersonName2ID(person_name) -> person_ID. This API is to convert user name to user ID.
CampusName2ID(campus_name) -> campus_ID. This API is to convert campus name to campus_name ID.
BookRoom(person_ID,room_ID,start_time,end_time) -> a list of meeting rooms. This API is to book a meeting room.

[User query]:
Please help Jack book a meeting room at TowerCenter from 9:00 to 10:00 this morning

[System response]:
BookRoom(person_ID=PersonName2ID(person_name='Jack’),
campus_ID=CampusName2ID(campus_name='TowerCenter'),
start_time='9am',end_time='10am')

Figure 4: An example of seed example.

Category	**example APIs**
Geocoding	GetDirections,GetUserDietaryRestrictions, DistanceCalculator
Weather	GPS2Weather,WeatherVerification
Book	AddBookToReadingList,BooksByAuthor
Transportation	FlightBooking,FindFlightByDestination
Music	AddSongToPlaylist,MusicConcert
Food & Drink	SearchRestaurant,TableReservation,RestaurantReviews
Entertainment	CinemaShowtimes,MovieReview, TheatrePlay
Shopping	FindProductId,NearestStore, ComparePrices
Health	GetExerciseRoutine,NearbyHospitalQuery,GetHealthInformation
Travel	SearchHotel,CheckBaggageAllowance,PlanTrip
Database	CheckInventory,DateConversion
Calculator	TaxCostCalculator,CalculateCalorie
Email	UserEmail2UserId,SendReview
Finance	InvestmentSuggestion,CountryTaxRate,
Convertor	User2Age,HotelName2ID
Clothes	SelectOutfit,OutfitSuggestion,FindClothingType
Time	ConvertTime,GetEventCalendar
Activity	ActivityBook,PlanDayOut
Currency Exchange	CurrencyConversion,GetExchangeRate
Search	GetCurrentFuelPrice,ProductSearch

Table 7: Domain distribution and examples of APIs in our dataset.

Dataset – Sample

```
{
        "APIs": [
          {
            "name": "CheckWeather",
            "Description": "This API checks the weather of a specific location.",
            "input_params": {
              "location": {
                "description": "the specific location",
                "type": "String"
              }
            },
            "output_params": {
              "weather": {
                "description": "the weather at the specific location",
                "type": "String"
              }
            },
            "format": "CheckWeather(location) -> weather"
          },
          {
            "name": "SelectOutfit",
            "Description": "This API selects an appropriate outfit based on the weather and
occasion.",
            "input_params": {
              "weather": {
                "description": "the weather condition",
                "type": "String"
              },
              "occasion": {
                "description": "the occasion",
                "type": "String"
              }
            },
            "output_params": {
              "outfit": {
                "description": "the recommended outfit",
                "type": "String"
              }
            },
            "format": "SelectOutfit(weather, occasion) -> outfit"
          }
        ],
        "Query": "I'm attending a birthday party in San Francisco tomorrow, what should I
wear?",
        "Label": "SelectOutfit(weather=CheckWeather(location='San Francisco'),
occasion='birthday party’)”,
},
```

Figure 5: An example of sample in dataset.

**Dataset Construction –
Sample Generation Prompt**

Your task is to first generate multiple APIs with their descriptions, and then generate a pair of user query and the corresponding label only using the predefined APIs in a nested manner, which means the output of one API is the input of another API. Note that for each user query, system response had better employ at least three APIs. Here is an example:

Example:
[API and API descriptions]:
PersonName2ID(person_name) -> person_ID. This API is to convert user name to user ID.
CampusName2ID(campus_name) -> campus_ID. This API is to convert campus name to campus_name ID.
BookRoom(person_ID,room_ID,start_time,end_time) -> a list of meeting rooms. This API is to book a meeting room.
[User query]:
Please help Jack book a meeting room at TowerCenter from 9:00 to 10:00 this morning
[System response]:
BookRoom(person_ID=PersonName2ID(person_name='Jack'),
campus_ID=CampusName2ID(campus_name='TowerCenter'),
start_time='9am',end_time='10am')

Given above example, please assume you are a professional assistant who generate multiple reasonable APIs with their descriptions (not limited to above mentioned ones), User query and system response using at least three APIs in a nested manner. Let's take a deep breadth and start generating APIs with their descriptions, user query and the corresponding system response using APIs in a nested manner. please give 2 different answers.
your answer should strictly follow the format:
answer1:
[API and API descriptions]:
xxx
[User query]:
xxx
[System response]:
xxx

answer2:
[API and API descriptions]:
xxx
[User query]:
xxx
[System response]:
xxx

your answer:

Figure 6: Prompt for new sample generation.

Dataset Construction –
Format Conversion Prompt

There are some APIs, related query and system response below. Please follow the format in the example, add the detailed infomation of "input_params" and "output_params" to the APIs, the detailed information includes the description and the type of the parameter. please return in a Json format.

Example:
[input]:
[API and API descriptions]:
PersonName2ID(person_name) -> person_ID. This API is to convert user name to user ID.
RoomName2ID(room_name) -> room_ID. This API is to convert room name to room ID.
BookRoom(person_ID,room_ID,start_time,end_time) -> a list of meeting rooms. This API is to book a meeting room.
[User query]:
Please help Jack book a meeting room at TowerCenter room from 9:00 to 10:00 this morning
[System response]:
BookRoom(person_ID=PersonName2ID(person_name='Jack'),
room_ID=RoomName2ID(room_name='TowerCenter'), start_time='9am',end_time='10am')

[output]:
```
{      "APIs": [
          {"name": "PersonName2ID", "Description": "This API is to convert user name to user ID.",
         "input_params": {"person_name": {"description": "the name of the person", "type": "String"}},
          "output_params": {"person_ID": {"description": "the ID of the person","type": "Integer"}},
           "format": "PersonName2ID(person_name) -> person_ID"},
          {"name": "RoomName2ID","Description": "This API is to convert room name to room ID.",
           "input_params": {"room_name": {"description": "the name of the room","type": "String"}},
           "output_params": {"room_ID": {"description": "the ID of the room","type": "Integer"}},
           "format": "RoomName2ID(room_name) -> room_ID"},
          {"name": "BookRoom","Description": "This API is to book a meeting room.",
           "input_params": {"person_ID": {"description": "the ID of the person","type": "Integer"},
                          "room_ID": {"description": "the ID of the room","type": "Integer"},
                            "start_time": {"description": "the start time of the meet","type": "Time"},
                            "end_time": {"description": "the end time of the meet","type": "Time"}},
          "output_params": {"booking status": {"description": "the status of the booking","type":
"String"}},
          "format": "BookRoom(person_ID,room_ID,start_time,end_time)-> booking status."}
       ],
       "Query": "Please help Jack book a meeting room at TowerCenter from 9:00 to 10:00 this morning",
      "Label":"BookRoom(person_ID=PersonName2ID(person_name='Jack'),room_ID=RoomName2ID
                        (room_name='TowerCenter'), start_time='9am',end_time='10am')"
 }
```

Please note that parameter types include Strings, Integer, Floats, Time, Dates, etc., and can be determined based on actual meanings. If the output of API 1 is the input of API 2, the type of the output parameter of your API 1 and the type of the corresponding input parameter in API 2 are the same.

now let's start with new case:
[API and API descriptions]:
xxx
[User query]:
xxx
[System response]:
xxxx

your answer, only return json format, don't generate any other content:

Figure 7: Prompt for Json format conversion.

Zero-Shot Prompt

We have the following functions. Please return function calling according to user instruction with the following format.
APIs: {api info}
user instruction: {user instruction}
please generate the function calling:

Figure 8: Prompt for Zero-Shot method.

Few-Shot Prompt

We have a list of APIs. Please return function calling according to user instruction.

Here is an example :

APIs:
{"Name": "MakeAppointment", "Description": "This API is to make an appointment.", "input_params": {"hospital_name": {"description": "hospital name", "type": "String"}, "department_name": {"description": "department name", "type": "String"}}, "output_params": {"appointment_status": {"description": "the status of the appointment", "type": "String"}}, "format": "MakeAppointment(hospital_name, department_name) -> appointment status"}
{"Name": "GetDepartment", "Description": "This API is to find the corresponding department given user symptom.", "input_params": {"symptom": {"description": "patient's symptom", "type": "String"}}, "output_params": {"department_name": {"description": "department name", "type": "String"}}, "format": "GetDepartment(symptom) -> department_name"}

user instruction: I'm in zheyi hospital, I have a stomachache and want to make an appointment to see a doctor.
function calling: MakeAppointment (hospital_name='zheyi', department_name= GetDepartment (symptom = 'stomachache')) "

Given above example, Please generate function calling according to user instruction and the given apis.
APIs: {api info}
user instruction: {user instruction}
please generate the function calling,the format must be the same as example:

Figure 9: Prompt for Few-Shot method.

Zero-Shot-CoT Prompt

We have the following functions. Please return function calling according to user instruction with the following format.
APIs: {api info}
user instruction: {user instruction}
please generate the function calling, let's think step by step:

Figure 10: Prompt for Zero-Shot-CoT method.

Few-Shot-CoT Prompt

We have a list of APIs. Please return function calling according to user instruction.

Here is an example :

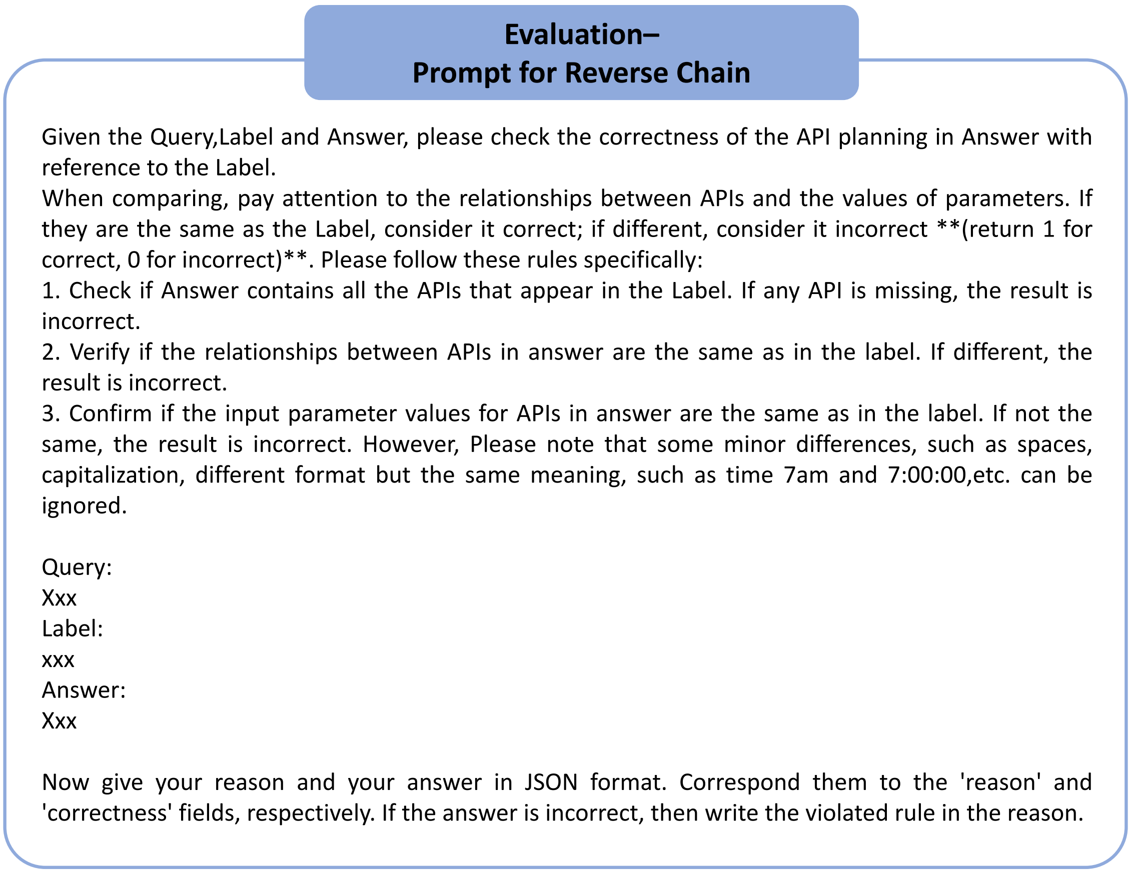
APIs:
{"Name": "MakeAppointment", "Description": "This API is to make an appointment.", "input_params": {"hospital_name": {"description": "hospital name", "type": "String"}, "department_name": {"description": "department name", "type": "String"}}, "output_params": {"appointment_status": {"description": "the status of the appointment", "type": "String"}}, "format": "MakeAppointment(hospital_name, department_name) -> appointment status"}
{"Name": "GetDepartment", "Description": "This API is to find the corresponding department given user symptom.", "input_params": {"symptom": {"description": "patient's symptom", "type": "String"}}, "output_params": {"department_name": {"description": "department name", "type": "String"}}, "format": "GetDepartment(symptom) -> department_name"}

user instruction: I'm in zheyi hospital, I have a stomachache and want to make an appointment to see a doctor.
thought:
1. you choose the API named 'GetDepartment', the value for reqiured parameter 'symptom' is 'stomachache', then you will get the output parameter department_name.
2. then you get hospital_name='zheyi'.
3. Finally, you choose the API named 'MakeAppointment'.

so the function calling:
MakeAppointment (hospital_name='zheyi', department_name= GetDepartment (symptom = 'stomachache')) "

Given above example, Please generate function calling according to user instruction and the given apis.
APIs: {api info}
user instruction: {user instruction}
please generate the function calling,the format must be the same as example:

Figure 11: Prompt for Few-Shot-CoT method.


**Evaluation–
Prompt for Reverse Chain**

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
When comparing, pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**. Please follow these rules specifically:
1. Check if Answer contains all the APIs that appear in the Label. If any API is missing, the result is incorrect.
2. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
3. Confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, different format but the same meaning, such as time 7am and 7:00:00,etc. can be ignored.

Query:
Xxx
Label:
xxx
Answer:
Xxx

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 12: Prompt for evaluation for Reverse Chain.

Evaluation–
Prompt for Zero-Shot

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
Please note that the format of answer is not fixed as that of label, so when comparing, only pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**.
Please follow these rules specifically:
1. The format of the answer is not a criterion for correctness. For instance, the following situations are considered correct:
   1.1 Missing the names of parameters, but not missing the parameter values, such as:
   [Label]
   AddSongToPlaylist(user_ID=UserName2ID(user_name='Emily'), playlist_ID=PlaylistName2ID(playlist_name='Classic Disco Hits'), song_name='Billie Jean')
   [Answer]
   AddSongToPlaylist(UserName2ID("Emily"), PlaylistName2ID("Classic Disco Hits"), "Billie Jean") -> playlist_songs

   In the Answer, it lacks the input parameter names user_ID, playlist_ID, playlist_name, but the parameter values are correct, thus it is considered correct.

   1.2 Splitting the execution of APIs, and the calling relationships between the APIs are correct
   [Label]
   AddSongToPlaylist(playlist_ID=PlaylistName2ID(playlist_name='Best Songs'), song_ID=SongName2ID(song_name='Imagine'))
   [Answer]
   PlaylistName2ID("Best Songs") -> playlist_ID
   SongName2ID("Imagine") -> song_ID
   AddSongToPlaylist(playlist_ID, song_ID) -> song_status

   First, execute API PlaylistName2ID to obtain playlist_ID, then execute API SongName2ID to obtain song_ID, and finally execute API AddSongToPlaylist. Since the parameter values of each API are correct, it is considered correct.

2. Check if Answer contains all the APIs that appear in the Label. If any API is missing, the result is incorrect.
3. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
4. Confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, etc. can be ignored.

Query:
xxx
Label:
xxx
Answer:
Xxx

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 13: Prompt for evaluation for Zero-Shot.

Evaluation–
Prompt for Few-Shot

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
Please note that the format of answer is not fixed as that of label, so when comparing, only pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**.
Please follow these rules specifically:
1. The format of the answer is not a criterion for correctness. For instance, the following situations are considered correct:
   1.1 Missing the names of parameters, but not missing the parameter values, such as:
   [Label]
   AddSongToPlaylist(user_ID=UserName2ID(user_name='Emily'), playlist_ID=PlaylistName2ID(playlist_name='Classic Disco Hits'), song_name='Billie Jean')
   [Answer]
   AddSongToPlaylist(UserName2ID("Emily"), PlaylistName2ID("Classic Disco Hits"), "Billie Jean") -> playlist_songs

   In the Answer, it lacks the input parameter names user_ID, playlist_ID, playlist_name, but the parameter values are correct, thus it is considered correct.

   1.2 Splitting the execution of APIs, and the calling relationships between the APIs are correct
   [Label]
   BuyMovieTickets(show_time=MovieShowtimes(movie_name=FindMovie(genre='romantic'), city='San Francisco'), movie_name=FindMovie(genre='romantic'), seats=2)
   [Answer]
   FindMovie(genre='romantic'), MovieShowtimes(movie_name=FindMovie(genre='romantic'), city='San Francisco') -> show_time, BuyMovieTickets(show_time=MovieShowtimes(movie_name=FindMovie(genre='romantic'), city='San Francisco'), movie_name=FindMovie(genre='romantic'), seats=2) -> booking_status
   First, execute API FindMovie to obtain movie_name, then execute API MovieShowtimes to obtain show_time, and finally execute API BuyMovieTickets. Since the parameter values of each API are correct, it is considered correct.

2. Check if Answer contains all the APIs that appear in the Label. If any API is missing, the result is incorrect.
3. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
4. Confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, different format but the same meaning, such as time 7am and 7:00:00, etc. can be ignored.

Query:
xxx
Label:
xxx
Answer:
Xxx

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 14: Prompt for evaluation for Few-Shot.

Evaluation–
Prompt for Zero-Shot-CoT

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
Please note that the format of answer is not fixed as that of label, answer may include step-by-step thoughts and final function calling, so when comparing, only pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**.
Please follow these rules specifically:
1. The format of the answer is not a criterion for correctness. For instance, the following situations are considered correct:
  1.1 Missing the names of parameters, but not missing the parameter values, such as:
  [Label]
  AddSongToPlaylist(user_ID=UserName2ID(user_name='Emily'), playlist_ID=PlaylistName2ID(playlist_name='Classic Disco Hits'), song_name='Billie Jean')
  [Answer]
  AddSongToPlaylist(UserName2ID("Emily"), PlaylistName2ID("Classic Disco Hits"), "Billie Jean") -> playlist_songs
  In the Answer, it lacks the input parameter names user_ID, playlist_ID, playlist_name, but the parameter values are correct, thus it is considered correct.
  1.2 Splitting the execution of APIs, and the calling relationships between the APIs are correct, such as:
  [Label] SetAlarm(timezone=GeoLocation2TimeZone(geolocation=GetUserGeolocation(user_name='Daniel')), time='5:30am')
  [Answer]
  Step 1: Get the user's geolocation
  Function calling: GetUserGeolocation("Daniel") -> user_geolocation
  Step 2: Convert the geolocation to timezone
    Function calling: GeoLocation2TimeZone(user_geolocation) -> timezone
  Step 3: Set the alarm in the specified timezone
  Function calling: SetAlarm(timezone, "5:30am") -> alarm_status
  In this case, first, execute API GetUserGeolocation to obtain user_geolocation, then execute API GeoLocation2TimeZone to obtain timezone, and finally execute API SetAlarm. Since the parameter values of each API are correct, it is considered correct.

2. Check if Answer contains all the API that appear in the Label. If any API is missing, the result is incorrect.
3. Function calling must include explicit API names and must match those in the label to be considered correct. answer in the following example lacks explicit API names, so it is considered incorrect:
  [Label]
  AddSongToPlaylist(user_ID=UserName2ID(user_name='Olivia'), playlist_ID=PlaylistName2ID(playlist_name='90s Nostalgia'), song_name='Smooth Criminal')
  [Answer]:
  1. Get the user ID of Olivia
  2. Get the ID of the '90s Nostalgia' playlist
  3. Add 'smooth Criminal' to the playlist
  Please generate the function calling according to the user instruction.
  Please note that the input and output parameters of the functions are just examples, and the actual parameters may vary depending on the specific implementation of the API.

4. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
5. Verify whether each input parameter for API in answer has a value, if there is no value, consider it incorrect.
6. Confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, different format but the same meaning, such as time 7am and 7:00:00, etc. can be ignored.

Query:
XXX
Label:
XXX
Answer:
XXX

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 15: Prompt for evaluation for Zero-Shot-CoT.

Evaluation–
Prompt for Few-Shot-CoT

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
Please note that the format of answer is not fixed as that of label, In general, answer consists of two components: thought and function calling. You only need to focus on whether the function calling part is correct.
when comparing, only pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**.
Please follow these rules specifically:
1. The format of the answer is not a criterion for correctness. For instance, the following situations are considered correct:
   1.1 Missing the names of parameters, but not missing the parameter values, such as:
   [Label]
   AddSongToPlaylist(user_ID=UserName2ID(user_name='Emily'), playlist_ID=PlaylistName2ID(playlist_name='Classic Disco Hits'), song_name='Billie Jean')
   [Answer]
   AddSongToPlaylist(UserName2ID("Emily"), PlaylistName2ID("Classic Disco Hits"), "Billie Jean") -> playlist_songs

   In the Answer, it lacks the input parameter names user_ID, playlist_ID, playlist_name, but the parameter values are correct, thus it is considered correct.

   1.2 Splitting the execution of APIs, and the calling relationships between the APIs are correct, such as:
   [Label]
 SetAlarm(timezone=GeoLocation2TimeZone(geolocation=GetUserGeolocation(user_name='Daniel')), time='5:30am')
   [Answer]
   GetUserGeolocation(user_name='Daniel') ->geolocation
   GeoLocation2TimeZone(geolocation) ->timezone
   SetAlarm(timezone,time='5:30am') -> alarm_status
   In this case, first, execute API GetUserGeolocation to obtain geolocation, then execute API GeoLocation2TimeZone to obtain timezone, and finally execute API SetAlarm. Since the parameter values of each API are correct, it is considered correct.

2. Check if Answer contains all the API that appear in the Label. If any API is missing, the result is incorrect.
3. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
4. Verify whether each input parameter for API in answer has a value, if there is no value, consider it incorrect.
5. Confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, different format but the same meaning, such as time 7am and 7:00:00, etc. can be ignored.

Query:
xxx
Label:
xxx
Answer:
xxx

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 16: Prompt for evaluation for Few-Shot-CoT.

Evaluation–
Prompt for ReAct

Given the Query,Label and Answer, please check the correctness of the API planning in Answer with reference to the Label.
Please note that the format of answer is not fixed as that of label, and the format of the answer is not a criterion for correctness.when comparing, only pay attention to the relationships between APIs and the values of parameters. If they are the same as the Label, consider it correct; if different, consider it incorrect **(return 1 for correct, 0 for incorrect)**.
Typically, the format of the answer follows the execution of the split API, following is a correct case:
[Label]
AddSongToPlaylist(user_ID=UserName2ID(user_name='Jack'), playlist_ID=PlaylistName2ID(playlist_name='Party Mix'), song_name='Havana')
[Answer]
UserName2ID("user_name": "Jack")
PlaylistName2ID("playlist_name": "Party Mix")
AddSongToPlaylist("user_ID": "user_ID", "playlist_ID": "playlist_ID", "song_name": "Havana")

In this case, first, execute API UserName2ID to obtain user_ID, then execute API PlaylistName2ID to obtain playlist_ID, and finally execute API AddSongToPlaylist. Since the parameter values of each API are correct(from the other previous API or obatined directly), it is considered correct.

Please follow these rules specifically:

1. Check if Answer contains all the API that appear in the Label. If any API is missing, the result is incorrect.
2. Verify if the relationships between APIs in answer are the same as in the label. If different, the result is incorrect.
3. Check whether each input parameter for API in answer are mentioned, if some parameters is missed, consider it incorrect.
4. There are two possibilities for value of input parameter, both of them are considered as correct: one is a valid value directly extracted from the query (this case is judged according to rule 3.1), and the other is a placeholder or descriptive text (this case is judged according to rule 3.2).
4.1 For the former, confirm if the input parameter values for APIs in answer are the same as in the label. If not the same, the result is incorrect. However, Please note that some minor differences, such as spaces, capitalization, different format but the same meaning, such as time 7am and 7:00:00, etc. can be ignored.
4.2 For the latter case for placeholder, the answer is also correct. For example:
[Label]:
AddSongToPlaylist(user_ID=UserName2ID(user_name='Olivia'), playlist_ID=PlaylistName2ID(playlist_name='90s Nostalgia'), song_name='Smooth Criminal')
[Answer]:
UserName2ID("user_name": "Olivia")
PlaylistName2ID("playlist_name": "90s Nostalgia")
AddSongToPlaylist("user_ID": "Olivia's user ID", "playlist_ID": "90s Nostalgia playlist ID", "song_name": "smooth Criminal")

In this case, the values 'Olivia's user ID' and '90s Nostalgia playlist ID' in the AddSongToPlaylist API call are placeholders or descriptive texts, however, the value of these two placeholders can be obtained from the previously executed APIs, UserName2ID and PlaylistName2ID, therefore, it is considered correct.

5. When an API is repeatedly mentioned in the answer, it is considered correct as long as it is executed correctly at least once. For example:

[label]:
AddSongToPlaylist(user_ID=UserName2ID(user_name='Sophia'), playlist_ID=PlaylistName2ID(playlist_name='Jazz Legends'), song_name='Let It Be')

[answer]:
UserName2ID("user_name": "Sophia")
PlaylistName2ID("playlist_name": "Jazz Legends")
AddSongToPlaylist("user_ID": "user_ID", "playlist_ID": "playlist_ID", "song_name": "Let It Be")
AddSongToPlaylist("user_ID": "user_ID", "playlist_ID": "playlist_ID", "song_name": "Let It Be")

in this case, the API AddSongToPlaylist is executed twice, and it is recognized as correct since this API is executed correctly.

Query:
xxx
Label:
xxx
Answer:
xxx

Now give your reason and your answer in JSON format. Correspond them to the 'reason' and 'correctness' fields, respectively. If the answer is incorrect, then write the violated rule in the reason.

Figure 17: Prompt for evaluation for ReAct.