API Mapping using Self Reflection in Large Language Models

Anonymous ACL submission

Abstract

API sequencing has received significant attention, leading to the development of datasets and solutions aimed at generating correct sequences of API calls for complex tasks. However, little work has been done on the task of API mapping—a novel task that involves identifying and linking functionally equivalent endpoints across different tools to accomplish tasks that require accessing multiple functionalities, much like performing joins in database querying. In this work, we focus on mapping APIs within the ToolBench dataset. By leveraging rich annotation resources and a self-reflection mechanism to iteratively refine mapping decisions, our approach identifies common fields and functional overlaps between API endpoints, thereby enabling the integration of multiple tools for multi-faceted tasks. Unlike existing systems that rely on the function calling capabilities of frontier models and require meticulously organized API data, our method demonstrates that effective API mapping can be achieved with smaller open source models and more flexible data organization, thereby providing a more accessible and cost-efficient solution for real-world applications.

1 Introduction

In recent years, the use of large language models (LLMs) for task completion through tool sequencing and subsequent tool invocation has garnered significant interest. While LLMs excel at tasks like summarization, sentiment analysis, reasoning, and even code generation due to their intrinsic knowledge, they are not always up-to-date on current events and are inherently incapable of interacting with the outside world to perform real-world tasks. To harness the full reasoning and problem-solving capabilities of LLMs, they must be augmented with tools that either supplement their knowledge—such as web search or document retrieval systems—or enable them to perform specific actions (e.g., APIs for booking a table in a restaurant or purchasing a flight ticket).

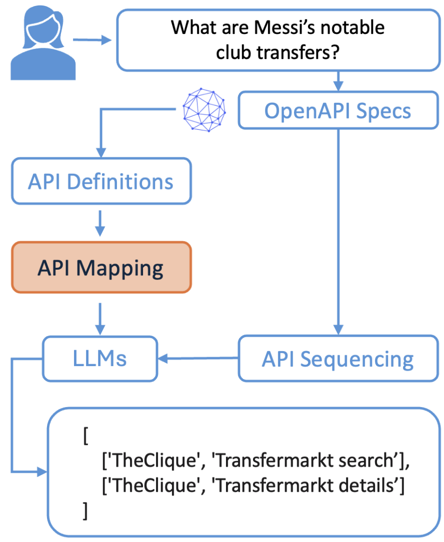

Figure 1: API Mapping can improve API sequencing performance of Large Language Models.

To enhance the tool calling capabilities of LLM, state-of-the-art approaches such as ToolLLM (Qin et al., 2023) and AnyTool (Du et al., 2024) have been developed. These methods construct benchmark datasets from RapidAPI (RapidAPI, 2023), where natural language utterances designate tasks that can be solved using APIs. They also provide corresponding solution paths obtained using state-of-the-art LLMs such as ChatGPT (OpenAI, 2023) and GPT-4 (Achiam et al., 2023). To retrieve relevant APIs from over 16,000 available endpoints, ToolLLM employed a specially trained BERT-based neural API retriever, whereas AnyTool leverages GPT-4's function calling in combination with a hierarchical organization of APIs. Once relevant APIs are retrieved, the LLM must discover the correct set of APIs to call to accomplish the task—often using prompting strategies such as ReAct and DFSDT (Qin et al., 2023)—until the LLM determines that the task has been solved or deems it unsolvable.

In this work, we introduce a novel task of API mapping across multiple datasets—a capability that is not currently available in the literature. Figure 1 describes the API mapping task with an example. The API mapping task would map APIs with alternative equivalent APIs, thereby returning multiple sequences for the ultimate task. This task is motivated by the need to build data products that seamlessly integrate multiple APIs, enabling users to select alternative endpoints based on criteria such as API availability, latency, and the number of required processing steps. For instance, when a user is limited by subscription constraints or performance issues, our approach can identify alternative APIs that deliver similar functionality but with lower cost or faster response times. We argue that in many cases there may be multiple APIs with similar or overlapping functionality that can satisfy a user's requirements. In such scenarios, it is desirable to present the user with alternative APIs and corresponding solution paths—thus allowing the selection of a preferred alternative based on factors such as cost, access, or latency.

In addition, our work leverages rich annotation resources—including detailed annotations on the ToolLLM dataset—that have not been publicly released. (Du et al., 2024) had explored self-reflection for the API sequencing task. In a similar vein, we propose a self-reflection mechanism for API mapping that allows the model to iteratively refine its mapping decisions, thereby improving the accuracy of matching common fields. This stands in contrast to systems like AnyTool, which rely on highly organized API data and the function-calling capabilities of GPT-4. By demonstrating that our approach can achieve competitive performance using smaller models and less stringent data organization requirements, we offer a more cost-effective and broadly applicable solution for multi-API environments.

2 Related Work

ToolLLM (Qin et al., 2023) introduced the ToolBench dataset based on the APIs in RapidAPI Hub. ToolLLM includes an API Retriever that, given a natural language instruction, identifies and returns a relevant set of APIs. It introduces solution path annotation by constructing chains of API calls using ChatGPT, and enhances the reasoning capabilities of the system through a depth-first search based decision tree. In addition, the work presents an evaluation framework, ToolEval that includes metrics such as pass rate and win rate. ToolLLaMA is fined tuned on the ToolBench dataset achieving performance comparable to Gorilla.

AnyTool (Du et al., 2024) incorporates a self-reflection mechanism to improve the reliability of its solutions. Upon receiving a user query, the system proposes an initial solution that is subsequently evaluated for feasibility by GPT-4. If the solution is found impractical, AnyTool re-activates itself, taking into account the reasons for failure and relevant historical contexts. This iterative process, typically involving four to six rounds of self-reflection, is shown to enhance the pass rate significantly. The authors claim to have addressed some of the problems in the evaluation strategy used in ToolLLM. There are other works like MetaTool (Huang et al., 2023) concerned with tool calling.

Retrieving APIs for the sequencing task is a well known problem. (Li et al., 2023), (Patil et al., 2023) used text embedding based API retrievers while (Qin et al., 2023) used a method fine tuned with curated API retrieval data. We believe using tools and frontier models for this retrieval step is too expensive and not practical in real world applications. Hence we propose a cheaper solution based on indexing and observe that it does not adversely affect API mapping or even the sequencing task.

(Saha et al., 2024) and (Mandal et al., 2024) discuss API sequencing. (Basu et al., 2024) introduced a corpora for training LLMs and (Patil et al., 2023; Abdelaziz et al., 2024) presented different approaches to function calling. (Mandal et al., 2024) proposed using an API Graph to guide the sequencing process. (Chen et al., 2024) discusses API recommendation. (Moon et al., 2024) presented API-miner which extracts APIs from the OpenAPI specifications and finds similar APIs to help design new APIs. (Song et al., 2023) introduced the RestBench dataset which consists of python like function signatures. (Liang et al., 2024) discusses training foundation models to connect them to APIs to accomplish tasks.

Mapping API end points predates the advent of Large Language Models. (Lu et al., 2017) tried to do API mapping by reading API documents. (Liu et al., 2023) uses a knowledge graph to recommend analogical APIs. (Shao et al., 2022) constructs a API documentation graph using Graph Neural

Networks.

3 API Mapping

To perform API mapping on enterprise datasets, we receive a set of OpenAPI specifications as input. For the APIs in the ToolBench dataset, we take the API details provided in the dataset. We merge the API endpoints into a combined collection. We then organize them to allow search, mapping, and sequencing. The search layer supports queries over many APIs. The mapping step identifies similar functionalities across different endpoints. The sequencing step produces a plan of API calls in response to user utterances. We employ large language models (LLMs) for these tasks and include a self-reflection component.

In order to identify matching APIs for enterprise datasets, we analyze their response structure including fields and descriptions. We then identify matching fields. If the descriptions of the two APIs are similar or if there is sufficient overlap between the fields in the response, then we can conclude that the APIs are matching alternatives. We then index the APIs in such a way that if the user's utterance matches the description of one API, then we can also retrieve the alternative APIs in a single call to the API retriever. We do this by creating clusters of matching APIs and indexing them together in a vector database. While generating the API mappings, we create a graph of operations within the same OpenAPI specification. Two operations are neighbours if they share common fields or schema. While the operations are being matched, we retrieve the neighbours of the candidate operations and put them in the same cluster. Then the related operations are mapped at the same time so that the mapping algorithm considers all operations with shared fields/schema at the same time.

For generating API mappings from the ToolBench dataset, we use a probing strategy to retrieve candidates for matching. We first index all the 49937 APIs (belonging to 10388 tools from 50 categories) in ToolBench as a single collection in Milvus using the API name and the API description. We then fire the first 10000 queries from the train set of ToolBench and retrieve 10 matching APIs for each query. We then pass these 10 APIs to an LLM (Mistral Large) in a single prompt to find mappings between the APIs. In this way, we generate a variable number of API mappings per query along with reasoning (this number depends on how many mappings the LLM is able to detect within the 10 APIs). We also include an additional step of self-reflection where the LLM is asked to criticize the reasoning behind the mappings and generate alternative API mappings if necessary. It is expected that the mappings produced after self-reflection will be more reasonable (or intuitive) than the original mappings. Finally we collate all LLM generated mappings from the 10000 queries.

Mappings	Few Shot ICL	Self-Reflection
Total	24235	27643
Correct	14940	17210
Accuracy	0.6160	0.6226

Table 1: API mappings with annotations in the ToolBench dataset. We annotated 10000 queries covering 49937 APIs from 10388 tools.

Annotation

We developed an annotation tool to obtain realistic ground truth data for the API mapping task. Although the tool is capable of supporting annotations for API retrieval and sequencing as well, our primary focus is on mapping APIs—that is, identifying and validating relationships between endpoints based on similarities in their descriptions and shared fields. For each natural language user utterance, the tool displays a set of candidate API endpoints along with the model-generated mappings between these endpoints and their corresponding fields. Annotators review these mapping outputs to determine if they accurately represent the relationships and functionalities; when they do not, annotators have the option to provide corrected mappings. In addition, the tool offers functionality for the annotation of retrieval and sequencing outputs, but these features are only briefly introduced. The overall goal of our annotation process is to generate high-quality, expert-verified mapping annotations that can be used to evaluate and improve our API mapping approach.

4 API Sequencing

In our API sequencing experiment, we investigate the ability of large language models (LLMs) to generate correct sequences of REST API calls in response to natural language instructions. Our approach comprises two main stages: (i) indexing canonical API lists and (ii) retrieving candidate APIs from a vector database (Milvus) based on user

Model	Prompt	Set Metrics			Order Metrics		
		Precision	Recall	F1	Precision	Recall	F1
mixtral-8x7b-instruct	ICL	0.2208	0.1904	0.2045	0.1650	0.1422	0.1528
mixtral-8x7b-instruct	Self-Reflection	0.2803	0.2735	0.2769	0.2332	0.2276	0.2303
granite-3-1-8b-instruct	ICL	0.0964	0.0875	0.0917	0.0771	0.0700	0.0734
granite-3-1-8b-instruct	Self-Reflection	0.1579	0.1904	0.1726	0.1143	0.1379	0.1250

Table 2: Average set and order precision, recall, and F1 scores for API sequencing on the Toolbench dataset.

queries. We experiment with annotated version of the **ToolBench** dataset. For each dataset, we first construct a canonical API list from raw OpenAPI specifications, which preserves all the original information (e.g., tool name, API name, description, and parameter information). These canonical lists are used to populate vectordb collections.

API Retrieval

Given a set of natural language queries for ToolBench, our system retrieves candidate APIs from the corresponding vector database collection. These candidates—along with the original query—are used to build few-shot prompt examples for sequencing.

Prompting Strategies We experimented with different prompting styles for generating API sequences. Few-Shot Prompting uses a simple prompt that provides a description of the task along with a few input-output examples. Next, the Chain-of-Thought (CoT) (Wei et al., 2022) approach encourages the model to generate intermediate reasoning steps before producing the final sequence. The ReAct style interleaves explicit Thought, Observation, and Action steps to guide the model's output. The difference in performance between few shot in-context learning and CoT and ReAct style prompts were not statistically signficant. Hence we only report few shot ICL numbers in Table 2. Similar to the API Mapping task, self-reflection seems to help models in the API sequencing task. Here self-reflection strategy instructs the model to autonomously refine its output until a satisfactory API sequence is produced.

Evaluation SEAL (Kim et al., 2024) provides a suite of tools for evaluating LLMs in API related tasks. (Qin et al., 2023) also introduced ToolEval in their work. Given we want to avoid the use of frontier models and agents merely for the task of retrieval, we have also used a much simpler evaluation strategy. To evaluate the generated API sequences, we compute precision, recall, and F1 scores for each test example by comparing the API sequence produced by our system against the corresponding ground truth. For a given example, precision is defined as the proportion of generated API endpoints that match those in the ground truth, while recall is defined as the proportion of ground truth endpoints that are correctly retrieved by our system. The F1 score is then calculated as the harmonic mean of precision and recall. We report both set and order level metrics. As shown in Table 2, self-reflection seems to help the smaller models in our experiments. However, we recognize that the overall performance of these models on this task requires lot more improvement.

We aexperimented with incorporating API mapping information into the API sequencing task by expanding the set of candidate APIs available for the model after retrieval. But as expected, this did not give promising results since we introduce more choices rather than reducing them. A more promising approach seems to be using the API mapping information to induce the model to self-reflect. As shown in Appendix A.1.3, we tell the model that a number of API mappings exist between its output and candidate APIs and the model can make use of them to refine its answer. This gave us the improvements reported in Table 2.

Conclusion

We introduced the new API Mapping task that has so far not been discussed in the literature. We designed a novel method for API mapping using Large Language Models that benefits from self-reflection. This API mapping when used as a preprocessing step, partly mitigates the need for using expensive models like GPT4 for API retrieval and function calling tasks. We conducted experiments on the ToolBench dataset and showed how different open source models using various prompt styles seem to benefit from API mapping while performing the API sequencing task.

Limitations

Our work on API mapping has some limitations that should be acknowledged. While we demonstrate the effectiveness of API mapping on the ToolBench dataset, our approach currently focuses primarily on functional similarity and field matching. This may miss more nuanced relationships between APIs that could be captured through deeper semantic analysis. The self-reflection mechanism, while effective, still relies on the capabilities of the underlying LLM. As shown in our experiments, the performance varies significantly across different models and prompting strategies, indicating room for improvement in the mapping methodology.

Our evaluation metrics for API mapping could be expanded to include more real-world considerations such as API performance, cost, and reliability - factors that may influence API selection. The current approach does not address versioning or evolution of APIs over time, which could affect the stability and reliability of the mappings. Furthermore, while we demonstrate that smaller models can be effective for API mapping, complex cases will benefit from larger models.

Ethical Statement

Our work focuses on improving the efficiency and accessibility of API mapping and sequencing using publicly available datasets and open-source models. The ToolBench dataset and associated APIs are already publicly accessible through RapidAPI, and our work simply helps organize and optimize their usage. From an ethical perspective, our approach actually promotes responsible API usage by enabling users to find legitimate alternative APIs when primary services are unavailable, reducing dependency on expensive, proprietary models for API discovery and mapping, and making API integration more accessible to developers working with limited resources.

The API mapping task we introduce is focused on improving software development efficiency and does not involve personal data or sensitive information. Our work builds upon existing, public API documentation and specifications, and therefore does not introduce new privacy or security concerns beyond those already addressed by the API providers themselves. The primary goal is to enhance developer productivity and system reliability through better understanding and utilization of available API resources.

References

Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, Shajith Ikbal, Sachin Joshi, Hima Karanam, Vineet Kumar, Asim Munawar, Sumit Neelam, Dinesh Raghu, Udit Sharma, Adriana Meza Soria, Dheeraj Sreedhar, Praveen Venkateswaran, Merve Unuvar, David Cox, Salim Roukos, Luis Lastras, and Pavan Kapanipathi. 2024. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. *Preprint*, arXiv:2407.00121.

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

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Vernon Austel, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis Lastras. 2024. API-BLEND: A comprehensive corpora for training and benchmarking API LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12859–12870, Bangkok, Thailand. Association for Computational Linguistics.

Yujia Chen, Cuiyun Gao, Muyijie Zhu, Qing Liao, Yong Wang, and Guoai Xu. 2024. Apigen: Generative api method recommendation. *arXiv preprint arXiv:2401.15843*.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: self-reflective, hierarchical agents for large-scale api calls. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. 2023. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*.

Woojeong Kim, Ashish Jagmohan, and Aditya Vempaty. 2024. Seal: Suite for evaluating api-use of llms. *Preprint*, arXiv:2409.15523.

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.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2024. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *Intelligent Computing*, 3:0063.

Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. 2023. Recommending analogical apis via knowledge graph embedding. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1496–1508, New York, NY, USA. Association for Computing Machinery.

Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. 2017. Learning to infer api mappings from api documents. In *Knowledge Science, Engineering and Management: 10th International Conference, KSEM 2017, Melbourne, VIC, Australia, August 19-20, 2017, Proceedings 10*, pages 237–248. Springer.

Lakshmi Mandal, Balaji Ganesan, Avirup Saha, and Renuka Sindhgatta. 2024. Graph-guided API sequencing with reinforcement learning. In *The Third Learning on Graphs Conference*.

Sae Young Moon, Gregor Kerr, Fran Silavong, and Sean Moran. 2024. Api-miner: an api-to-api specification recommendation engine. In *Proceedings of the 1st IEEE/ACM Workshop on Software Engineering Challenges in Financial Firms*, pages 9–16.

OpenAI. 2023. Chatgpt. https://openai.com/chatgpt/overview/. Accessed: February 15, 2025.

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*.

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*.

RapidAPI. 2023. Rapidapi hub. https://rapidapi.com. Accessed: February 15, 2025.

Avirup Saha, Lakshmi Mandal, Balaji Ganesan, Sambit Ghosh, Renuka Sindhgatta, Carlos Eberhardt, Dan Debrunner, and Sameep Mehta. 2024. Sequential API function calling using GraphQL schema. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 19452–19458, Miami, Florida, USA. Association for Computational Linguistics.

Yanjie Shao, Tianyue Luo, Xiang Ling, Limin Wang, and Senwen Zheng. 2022. Cross platform api mappings based on api documentation graphs. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 926–935. IEEE.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.

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

A Appendix

A.1 Prompt Templates for API Sequencing

We provide below the different prompt templates that we used for the API sequencing task.

A.1.1 Few Shot

This prompt template is used to instruct the language model to generate an optimal sequence of REST API calls based on a user utterance and provided API details. The template frames the task by stating that the model is an expert in searching, sequencing, and mapping REST APIs. It includes several few-shot examples (labeled as Example 1, Example 2, and Example 3) to illustrate the expected input-output behavior. Finally, the template uses a placeholder «`input`» where the actual user query and candidate API details are inserted.

You are an expert in searching, sequencing, and mapping REST APIs. Your task is to generate the optimal sequence of API calls based on the user utterance and the provided API details. Your output should be a list of API calls that, when executed in sequence, achieve the desired outcome. Do not include any extra explanation. Generate only the API call sequence.

```
EXAMPLE 1

INPUT:
{input0}

OUTPUT:
{output0}

EXAMPLE 2

INPUT:
{input1}

OUTPUT:
{output1}

EXAMPLE 3

INPUT:
{input2}

OUTPUT:
{output2}

<<input>>
```

A.1.2 Chain-of-Thought

This prompt template is designed to guide the language model to generate an optimal sequence of REST API calls. The template instructs the model to first think through the problem by generating a chain-of-thought (CoT) before providing the final API sequence. In our task, the model receives a user utterance along with a set of candidate API details. It is then expected to output the correct sequence of API calls by first outlining its reasoning. The chain-of-thought portion helps ensure that the model carefully considers the API selection and ordering before producing the final output. The output should be in JSON format, consisting of a list of API calls (each formatted as `["tool_name", "api_name"]`) that, when executed in sequence, accomplish the desired task.

Below is the complete text of the Chain-of-Thought prompt template:

You are an expert in searching, sequencing, and mapping REST APIs. Your task is to generate the optimal sequence of API calls based on the user utterance and the provided API details. Your output should be a list of API calls that, when executed in sequence, achieve the desired outcome. Do not include any extra explanation. Generate only the API call sequence.

```
EXAMPLE 1

INPUT:
{input0}

THOUGHT:
- From the user utterance, I understand
that I need to call the following APIs:
    - {output0}

OUTPUT:
{output0}

EXAMPLE 2

INPUT:
{input1}

THOUGHT:
- From the user utterance, I understand
that I need to call the following APIs:
    - {output1}
```

```
OUTPUT:
{output1}

EXAMPLE 3

INPUT:
{input2}

THOUGHT:
- From the user utterance, I understand
that I need to call the following APIs:
    - {output2}

OUTPUT:
{output2}

<<input>>
THOUGHT:
```

A.1.3 Self-Reflection

This prompt template is designed to enable self-refinement of an initially generated API sequence. In our task, the language model first produces an initial API sequence based on a user query and a list of candidate APIs. This template then instructs the model to reflect on its initial output by comparing it against the provided API mapping analysis, ensuring that:

- The output is in the correct JSON format (i.e., a list of API calls formatted as `[["tool_name", "api_name"], ...]`).
- The words used in the API calls closely match those in the user utterance.
- Only API calls from the provided candidate list are present.

The model is prompted to either confirm its initial answer or generate a refined sequence, and to provide an explanation of how it refined its output. This two-step self-reflection process is critical for enhancing the correctness and reliability of the final API sequence.

The full content of the self-refine prompt template is provided below:

You are an expert in searching, sequencing, and mapping REST APIs.

Your learnt the task using an example like below. Notice how the generated API sequence is part of the API list in the input.

```
Example Input 1:
{input0}
Example Output 1:
{output0}
```

You had generated the following API sequence based on the user utterance and list of APIs. Now do self reflection to confirm your generation or generate a new API sequence.

```
User Utterance:
<<input>>

Your initial answer was:
<<output>>
```

Now, refine your answer to ensure that: 1. The output is in the correct format. [["tool_name", "api_name"], ["tool_name", "api_name"], ...] 2. The words in the API are similar to the words in the user utterance. 3. The tool_name and api_name that you output, must be in the given input list of APIs.

More often than not, the initial output is wrong. So changing the output could result in correct API sequence.

For example, see if the following analysis based on API mapping can help you refine your answer. «mapping_info»

Provide your refined answer in the same format as your initial answer.

Then explain how you refined your answer.