RestGPT: Connecting LLMs with RESTful APIs

Yifan Song[1], **Weimin Xiong**[1], **Dawei Zhu**[1], **Cheng Li**[2], **Ke Wang**[2], **Ye Tian**[2], **Sujian Li**[1*]
[1]School of Computer Science, Peking University
[2]Huawei Technologies
{yfsong, lisujian}@pku.edu.cn

Abstract

Tool-augmented large language models (LLMs) have achieved remarkable progress in tackling a broad range of queries. However, existing work are still in the experimental stage and has limitations in extensibility and robustness, especially facing the real-world applications. In this paper, we consider a more realistic scenario, connecting LLMs with RESTful APIs, which use the commonly adopted REST software architectural style for web service development. To address the practical challenges of planning and API usage, we introduce RestGPT, which leverages LLMs to solve user requests by connecting with RESTful APIs. Specifically, we propose a coarse-to-fine online planning mechanism to enhance the ability of planning and API selection. For the complex scenario of calling RESTful APIs, we also specially designed an API executor to formulate parameters and parse API responses. Experiments show that RestGPT is able to achieve impressive results in complex tasks and has strong robustness, which paves a new way towards AGI.

1 Introduction

Large language models (LLMs), such as GPT-3 [1], ChatGPT [2], and GPT-4 [3], have made significant progress on various natural language processing tasks over the past several years. LLMs have demonstrated emergent abilities, including in-context learning, mathematical reasoning, and step-by-step planning [4]. To extend the abilities of LLMs in real applications, an active research direction explores the use of external tools to augment LLMs. Early studies equip LLMs with simple tools, such as searching engines and calculator, to access real-time information and enhance mathematical ability [5–7]. Recently, ViperGPT [8], Visual ChatGPT [9], and HuggingGPT [10] incorporate a collection of foundation models to make LLMs capable of processing multi-modal tasks. Chameleon [11] explores to synthesize a sequence of tools with LLM to tackle a broader range of queries.

Despite significant progresses, we can see current API-augmented LLMs are still in the experimental stage and far from tackling queries from the real-world scenarios. As shown in Table 1, current methods can only connect with a small number of tools/APIs [6, 12, 8]. For example, Chameleon augments LLM with 15 different tools [11]. It is also noted that previous work heavily relies on specially designed APIs and meticulously crafted API descriptions, which in turn limits their extensibility. In such cases, the potential for LLMs to leverage a vast number of realistic APIs remains under-explored.

In this paper, to apply LLMs in a more realistic scenario, we consider to connect them with APIs which follow the Representational State Transfer (REST) software architectural style. REST provides a simple and standard interface using HTTP methods (e.g., `GET`, `POST`) and URIs to manipulate resources, which has become the *de facto* standard for web service development [13]. RESTful APIs usually follow OpenAPI Specification (OAS) [14], which describes the operations, parameters, and response schemas of each API endpoint. With the help of OAS, the resulting framework is naturally

*Corresponding author.

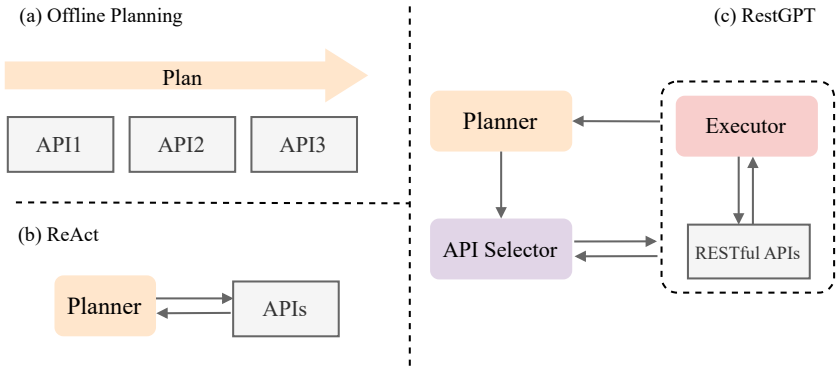

Figure 1: Comparison of RestGPT with previous work. (a) Offline-planning approaches, such as HuggingGPT [10], ViperGPT [8], Chameleon [11]. (b) ReAct [6], an online-planning approach. (c) Our proposed RestGPT, a coarse-to-fine online planning framework.

compatible with existing systems, and the API development process is standardized, leading to more powerful extensibility than previous work. However, connecting large language models to RESTful APIs also comes with practical challenges. First, it is difficult for LLMs to accurately comprehend the function and parameters of APIs under the constraint of maximum context length. At the same time, the responses of APIs often follow complex JSON formats, making it challenging to extract the pertinent information. Most importantly, while invoking APIs to address intricate user requests, there might be a multitude of unforeseeable circumstances that could arise.

Unfortunately, most existing approaches adopt an offline planning framework (Figure 1 (a)), which generates a static multi-step plan for tool use and executes the plan sequentially [10, 8, 11]. Due to the absence of feedback from tools, these systems are unable to adjust the plan to address errors that arise during tool usage. In contrast, ReAct [6] generates both reasoning traces and tool use actions in an interleaved manner, enabling online planning and exception handling (Figure 1 (b)). However, the simple structure of ReAct is insufficient to handle scenarios that use RESTful APIs to solve complex tasks.

In this work, we present a system called RestGPT to connect LLMs with large amount of RESTful APIs and handle complex user requests. As illustrated in Figure 1 (c), RestGPT consists of three main components: a Planner, an API Selector, and an Executor, the core of each component is prompting an LLM. Unlike previous work that generates static plans which are not adaptable to environment feedback, RestGPT employs a coarse-to-fine online planning framework. Specifically, the planner decomposes user instructions into sub-tasks with the format of natural language, which are then mapped onto API calls by the API selector, forming a coarse-to-fine task planning. On the other hand, the planner performs online planning of subsequent sub-tasks based on the executor's response. Another problem arises while executing RESTful API calls: the executor must be capable of accurately generating parameters, as well as parsing the intricate JSON responses generated by the APIs. Therefore, we further divided the Executor into two modules: a Caller and a response Parser. The caller reads the complete API documentation to organize the API call parameters while the parser generates Python code that parses responses based on the response schema defined in OAS.

We test RestGPT's capabilities in several scenarios, including movie information, music player, and note-taking application. The experimental results confirm that RestGPT has robust capabilities in handling complex user requests and has significant advantages in API understanding, task planning, response parsing.

Our contributions can be summarized as follows:

1. For the first time, we attempt to connect large language models with RESTful APIs, enabling the resulting framework to be compatible with existing systems while also providing powerful extensibility.

Model	API/Tool Use			Framework			
	Num.	Extensibility	Schema	Planning	Planning Form	Interaction	Plug-n-Play
ReAct	3	–	Specialized	online	natural lang.	✓	✓
Toolformer	5	–	Specialized	-	-	✗	✗
ART	3	–	Specialized	online	natural lang.	✓	✓
Visual ChatGPT	22	–	Specialized	-	natural lang.	human	✓
ViperGPT	11	–	Python func.	offline	program	✗	✓
HuggingGPT	24[1]	+	HuggingFace	offline	natural lang.	✗	✓
Chameleon	15	–	Specialized	offline	natural lang.	✗	✓
RestGPT (**ours**)	100+	++	RESTful	online	coarse-to-fine	✓	✓

Table 1: A comparison of work that augments large language models with tool usage.

2. We propose RestGPT, a multi-level online planning framework that effectively handles the practical challenges associated with integrating LLMs with RESTful APIs, including API understanding, planning, and API response parsing.
3. Experimental results from multiple scenarios demonstrate the capability of RestGPT to effectively utilize a vast number of RESTful APIs to solve complex tasks.

2 Related Work and Background

2.1 Tool-Augmented Language Models

The emergence of recent powerful LLMs has enabled artificial intelligence systems to match human skills in utilizing tools. To enhance the performance of LLMs in accessing up-to-date information and carrying out precise mathematical reasoning, some work leverages simple tools like web search engines and calculators, such as React [6], Toolformer [7], and ART [12]. Another line of research has focused on equipping LLMs to coordinate with external models for complex AI tasks, exemplified by HuggingGPT [10], ViperGPT [8], and Visual ChatGPT [9]. Recently, Chameleon [11] incorporates 15 tools and augments LLMs with plug-and-play modules for compositional reasoning. Additionally, API-Bank [15] provides a systematic benchmark to showcase the efficacy of LLMs using tools to respond to human instructions.

Despite the notable advancements in incorporating tools for large language models, previous works have exhibited certain limitations, most notably their restricted support for a limited number (typically no more than 30) of APIs and their reliance on offline planning methods. We compare RestGPT with other tool-augmented language models in Table 1. As shown in this table, our work stands out by providing support for over 100 widely adopted RESTful APIs that follow *de facto* standard in real-world development scenarios, enabling more comprehensive and practical application development. Furthermore, we employ an coarse-to-fine online planning routine with feedback, which enhances the robustness of our system by enabling dynamic adaptation and adjustment based on real-time API response. By leveraging this online planning routine, our work achieves a higher level of responsiveness and adaptability, facilitating improved performance in handling complex user queries.

2.2 RESTful APIs

RESTful APIs have become a popular way to expose functionalities and data of web services to client applications. RESTful APIs are based on the REST architectural style, which emphasizes a client-server communication via stateless HTTP requests, including `GET`, `POST`, etc, where resources are identified by self-descriptive URIs. RESTful APIs also provide a standardized way of integrating external systems together with using a simple yet powerful interface. There are also millions of RESTful APIs available on Internet, such as Spotify, Twitter, Gmail, etc.

To facilitate the development and documentation of RESTful APIs, the OpenAPI Specification (OAS), previously known as Swagger, has been widely adopted as a standard for defining RESTful API contracts. OAS is a machine-readable format for describing the endpoints, operations, parameters, responses, and other details of a RESTful API, providing a clear interface for developers to consume and extend the API. More specifically, an OAS consists of the following aspects for each API endpoint:

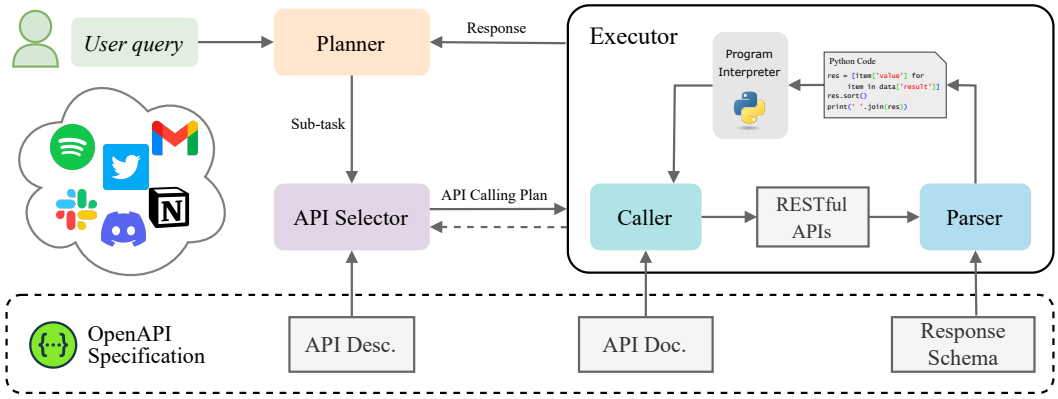

Figure 2: Overview of RestGPT. The planner, API selector, executor collaborate to form the coarse-to-fine online planning framework. The caller and response parser in the executor provides robust execution of the RESTful API calling plan.

- **API Path**: a relative path to an individual API endpoint, e.g., `/{person_id}/details`.
- **API Description**: what the API does, how it works, and any potential errors or exceptions that may be raised.
- **Request Method**: the desired action to be performed for the API, e.g., `GET`, `POST`.
- **Parameter List**: parameter name, parameter description, data type, default value, optional values of each parameter for the API.
- **Response Schema**: the schema of the response of the API. This information can assist the response parser to extract useful information from the JSON response.
- **Response Example (Optional)**: an example of a API call which can help demonstrate what the API will response.
- **Error and Exception**: potential error codes and their corresponding descriptions.

We provide an example of an OAS description of an API endpoint in the Appendix.

3 RestGPT

3.1 RestGPT Architecture

As demonstrated in Figure 2, RestGPT consists of three main components: a **Planner**, an **API Selector** and an **Executor**. The executor is composed of a **Caller** and a response **Parser**. The core of each component is an LLM with a prompt describing the function of the component.

The workflow of RestGPT can be characterized as an iterative "plan and execution" loop, which can be described as follows. In the planning stage, the planner employs LLMs' common-sense knowledge to conduct natural language planning to decompose the user query into a sub-task for the present step while the API selector thereafter chooses an appropriate API to solve the sub-task. Subsequently, the executor formulates the API call and parses the API response. The planner accepts the executor's response and generates sub-task for the next step. Once the planner outputs a termination signal, RestGPT concludes the loop and produces the final execution outcome.

One of the challenges in connecting LLMs with a vast array of APIs is to ensure that the framework is able to fully understand the API documents with the limited context window size. As depicted in Figure 2, in our method, we assign different modules to read distinct parts of the OpenAPI Specification (OAS). This approach allows us to leverage OAS information to its fullest potential when working with RESTful APIs. Specifically, the API selector reads the endpoint descriptions of all APIs to identify a proper API to address the current sub-task. The executor uses the detailed documents of the API within the API plan to generate the correct API calling parameters and data. Lastly, the parser is developed to make use of the response schema within OAS to generate the parsing code for information extraction.

3.2 Coarse-to-fine Online Planning

When applied to real-world scenarios such as RESTful APIs, the offline planning framework used by existing methods cannot dynamically adjust the plan to changing circumstances and errors that occur during execution. Moreover, we empirically find that current LLMs have poor ability to simultaneously conduct planning, API understanding and selection, especially when the system is connected to massive API endpoints. To tackle these problems, we propose a coarse-to-fine online planning mechanism in RestGPT, where we first divide one user query into coarse NL subtasks, and each NL sub-tasks is implemented by the elementary API functions.

During the planning stage, the planner and API selector collaborate to formulate a coarse-to-fine planning mechanism that fully utilizes the capabilities of LLMs. In particular, leveraging the commonsense knowledge stored in LLMs, the planner generates a natural language (NL) sub-task based on the user query, previous plan, and execution history, forming a high-level NL plan. Once the high-level sub-task plan is established, the API selector reads descriptions of available API endpoints to select APIs and construct the API calling plans. In this way, the planner and API selector are dedicated to NL sub-task planning and API selection, respectively, effectively utilizing the large language model's abilities of plan and text comprehension.

On the other hand, the planner dynamically plans the system's next actions based on the real-time results returned by the executor, which allows a more flexible online planning mechanism. It also plays a role of monitoring the whole process and outputs three kinds of signals: "continue", new NL sub-task plan, and "end". Specifically, if the planner monitors that the current executor's output has not completed the present NL sub-task, it will output the "continue" signal. The API selector will select another API based on the sub-task plan and the last executor response. If the planner estimates that the current sub-task has been fulfilled, it will plan a new natural language sub-task based on previous results and assign it to the API selector for the next round of API calling and response parsing. At last, if the planner monitors that the user's request has been completed, it will give the termination signal "end" and output the final result.

The planner, API selector, and executor collaborate to form RestGPT's coarse-to-fine online planning framework. This framework significantly enhances the ability to decompose tasks and select appropriate APIs, providing the model with the flexibility to effectively tackle user requests.

3.3 API Plan Execution

Once an API calling plan is generated, the next step is to execute the API calls. The executor consists of a caller and a response parser. In the execution stage, the caller should read the API documents carefully and assign correct parameter values for the API calling. Due to the constraints regarding maximum context length, we filter API documents and only preserve APIs appearing in current API calling plan. For example, if the API calling plan generated by API selector is "`GET /movie/{movie_id}` to get the title of the movie", then we only give the executor the document of "`GET /movie/{movie_id}`". The API caller accepts API calling plan and API documents, then generates API calling URL and the information extraction instructions. Next we use Requests Python library to call the corresponding API.

RESTful APIs typically return a response in JSON format when a request is received. Since modules in RestGPT interact and collaborate with each other via natural language, the executor must extract useful information from the response. To avoid the manual creation of parsing code for a vast range of APIs, we make extensive use of the capabilities of LLMs. However, the response may sometimes have a complex structure or be lengthy, making it difficult to extract important information directly prompting the LLMs. To address this problem, we leverage the response schema defined in the OAS. The response parser first generates Python parsing code based on the provided schema and instructions, which is then executed in the Python interpreter to process the response. If there are no execution errors, the output is returned. Otherwise, the LLM is prompted to parse the response directly as a backup.

API calling in realistic scenarios may encounter various errors and exceptions, such as missing parameters. To enhance robustness, the API executor follows an interaction agent architecture. If the API response indicates an error, the caller can read the error information and re-organize a new API calling to mitigate the issues.

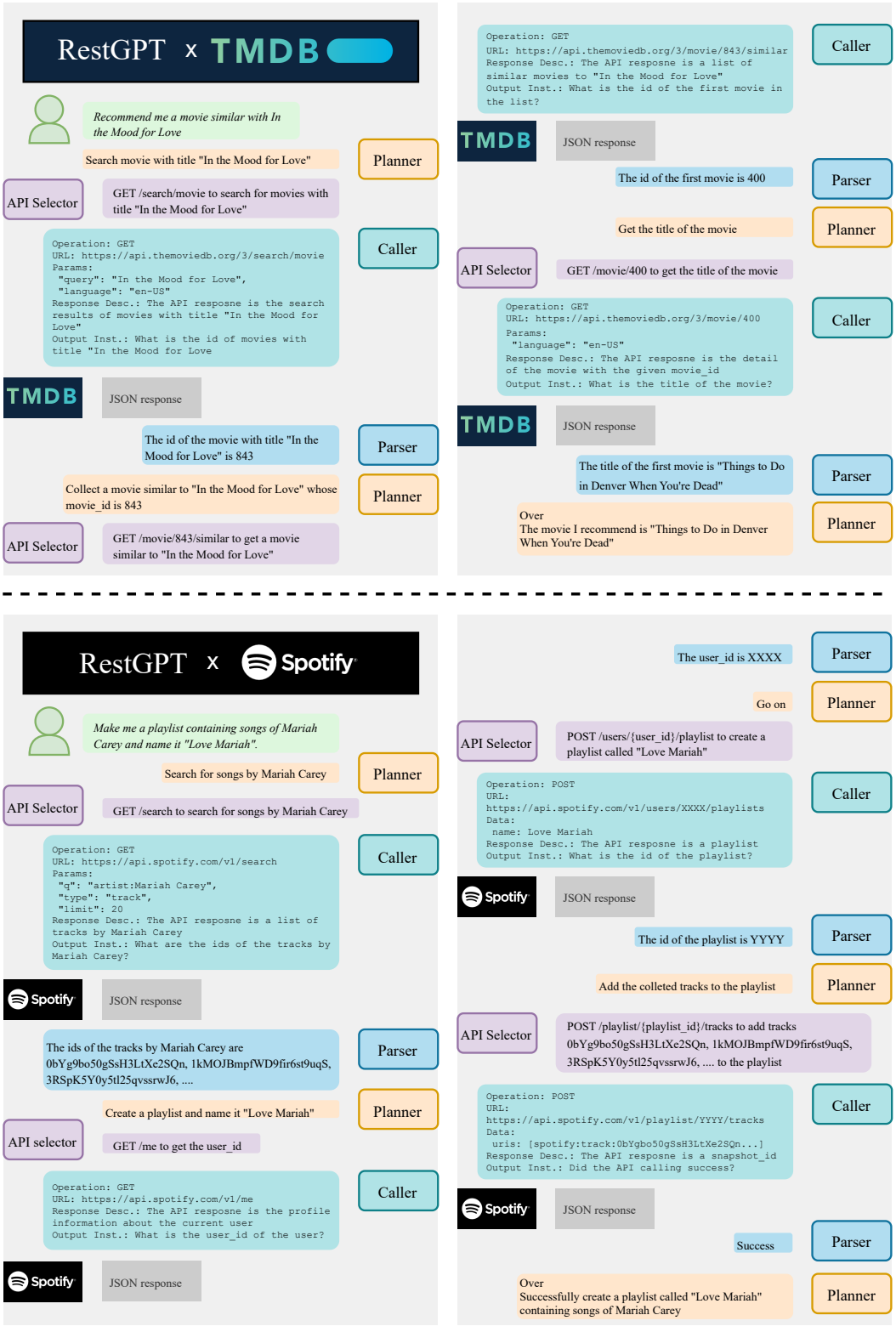

Figure 3: Examples of connecting RestGPT with TMDB and Spotify.

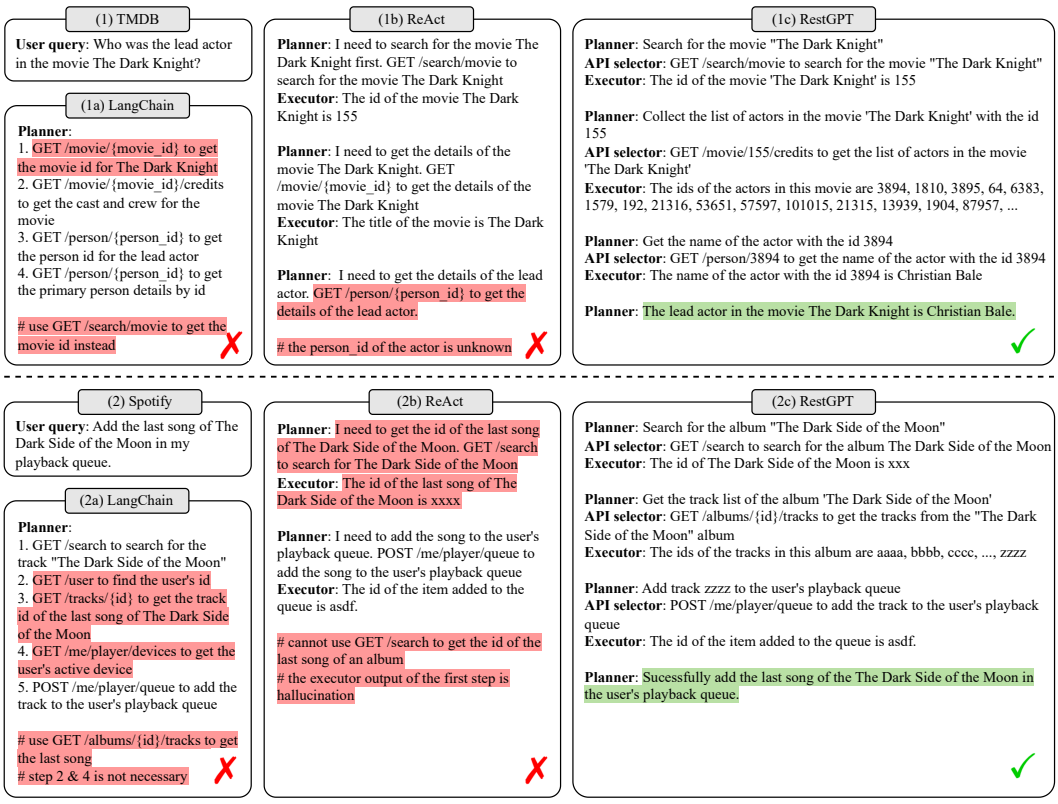

Figure 4: Comparison of three methods, (a) LangChain, (b) ReAct, and (c) RestGPT, connecting with TMDB and Spotify APIs to address complex user queries. For LangChain, we only show the generated plan. For ReAct and RestGPT, we omit the detailed execution process of the executor, and only show the output of the planner, API selector and the executor.

4 Experiments

4.1 Setting

In our experiments, we employ `text-davinci-003` as the large language model, which is publicly accessible through the OpenAI API. We implement RestGPT based on LangChain. The maximum length for generation is set to 256. We set the decoding temperature to 0 for the most deterministic generation. The detailed prompts designed for the planner, API selector, caller and response parser are provided in Appendix ?.

We select the TMDB movie information database and Spotify music player as scenarios to demonstrate the efficacy of RestGPT. Specifically, TMDB offers an OAS with 135 API endpoints, encompassing the information of movies, TVs, actors, and image APIs. As for Spotify, it provides 71 API endpoints that facilitate interaction with the platform's streaming service, enabling users to retrieve content metadata, receive recommendations, create and manage playlists, and control playback.

4.2 Qualitative Results

Figure 3 illustrate two examples of integrating RestGPT with TMDB and Spotify APIs, respectively. In each demo, the user provides a query that requires multiple API endpoints to be accessed. The planner of RestGPT first applies intent comprehension and planning to break down the user query into sub-tasks for the current step. The API selector then reads all API descriptions and identifies the appropriate APIs to solve the sub-task. The caller and parser in the executor collaborate to invoke the RESTful APIs and fulfill the specific API calling plan. Finally the planner generates instructions for the next step.

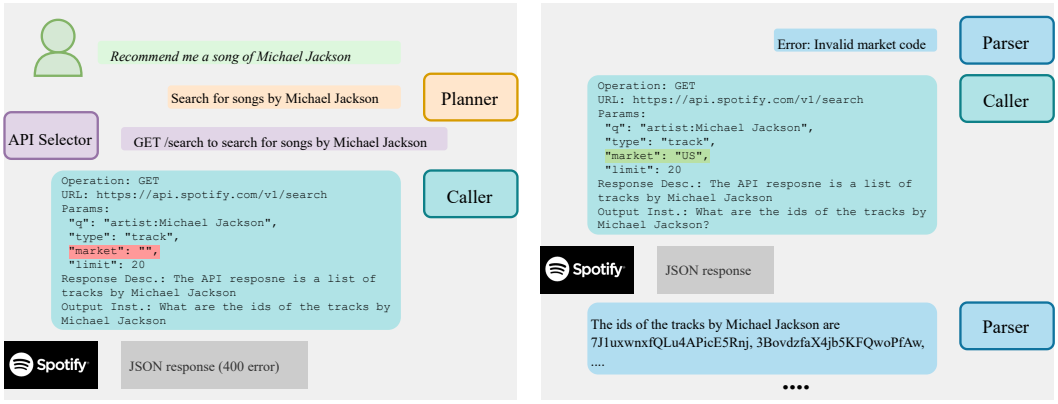

Figure 5: Case study of the caller. The caller first formulates a incorrect API parameter, leading the API to return an error message. The caller then reorganizes the call parameters based on the error message to troubleshoot and resolve the issue.

The TMDB case in Figure 3 demonstrates a recommendation request that RestGPT successfully decomposes into three steps: "search", "get similar", and "get detail" The API selector selects the most suitable API to solve each sub-task, and the caller accurately comprehends the API documents and formulates the correct API calling parameters.

The Spotify case in Figure 3 shows a slightly more complex example of creating a playlist. Notably, in the second "plan-execution" loop, the planner generates a sub-task of "create a playlist". The API selector correctly identifies that "`POST /users/{user_id}/playlist`" endpoint depends on the `user_id`, and hence calls "`GET /me`" to retrieve the `user_id` first.

4.3 Case Study of Planning: Comparison with Other Methods

In this section, we compare RestGPT with the offline planning and ReAct framework. To ensure a fair comparison, we employ the same LLMs (`text-davinci-003`) across all three methods. We notice that LangChain has implemented an agent that can connect with OpenAPI specification-compliant APIs. The architecture of LangChain's implementation can be classified as an offline planning structure, where the planner first generates a static plan first and then execute it sequentially. In addition to this, we implemented a RESTful API-enabled ReAct framework. Our ReAct implementation merges the planner and API selector components of RestGPT into a single component that simultaneously generates the chain-of-thought planning and selects the appropriate API.

The comparison results are shown in Figure 4. Firstly, the offline planning based LangChain is unable to solve most user queries. In the TMDB case, the first step of the plan is incorrect, since the "`GET /movie/{movie_id}`" requires movie id information as parameter. Due to the absence of feedback mechanism, it is unable to adjust the plan to address this fatal error. In the example of Spotify, the planner not only selects the wrong API (step 3), but also generates useless plans as well (step 2 and 4). Regarding ReAct, it can plan and execute to some extent. In the TMDB case, it successfully retrieves the movie id. However, we find that current LLMs have a poor ability to simultaneously conduct planning, API understanding and selction. The planner tends to either select an inappropriate API to fulfill the plan (Figure 4 (1b)) or generate a sub-task that is difficult to solve (Figure 4 (2b)). In contrast, the coarse-to-fine online planning framework of RestGPT fully exploits the LLMs' planning and document understanding capabilities, generating reasonable plans and selecting appropriate APIs to solve the sub-task.

4.4 Case Study of the Executor

Although LLMs have strong document comprehension abilities, RestGPT may still encounter errors when calling specific RESTful APIs. To enhance the system's robustness, the executor is designed as an agent with a feedback mechanism. When an API returns an error message, the caller can

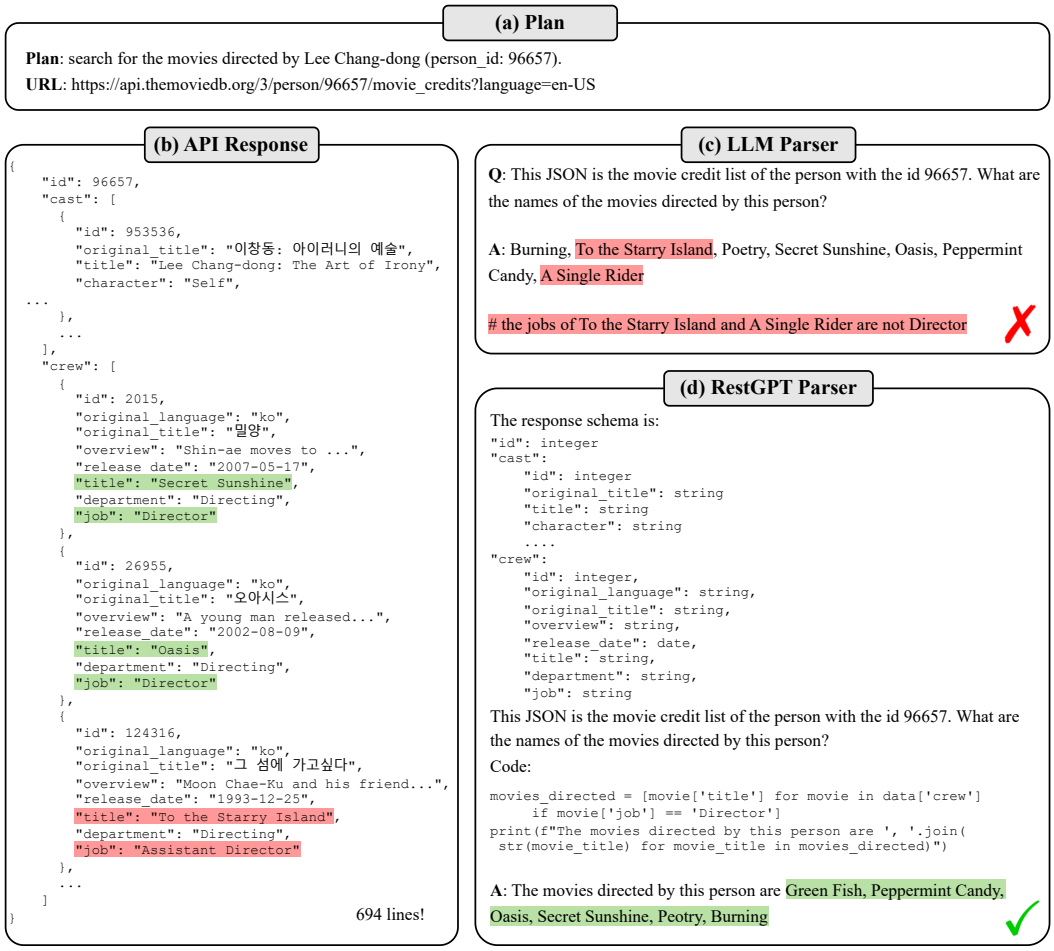

Figure 6: The detailed case study of response parser.

reorganize the call parameters based on the error message and attempt to resolve the issue. Figure 5 shows an example of this mechanism.

Figure 6 illustrates how RestGPT parses responses. We compare RestGPT parser with directly prompting an LLM as a parser. As shown, responses from RESTful APIs are usually in lengthy and complex JSON format (694 lines in Figure 6 (b)) The RestGPT parser generates information extraction code based on the response schema in the OAS to avoid parsing lengthy JSON files directly, significantly improving the efficiency and accuracy of response parsing.

5 Limitations

Although RestGPT can use RESTful API to handle complex user requests, it also faces several challenges. The most noteworthy one is its efficiency. Because each component of RestGPT's core is prompting a LLM, every plan-execution loop requires at least four interactions with LLMs. Consequently, the execution efficiency is low in practical applications. The second limitation is that RestGPT is still difficult to extend to infinitely APIs. Constrained by the maximum input length, the API selector can only access a finite number of APIs. Future work may introduce retrieval mechanisms to select the appropriate API. The third issue pertains to the framework's robustness. Although the planner, API selector, and executor all have feedback mechanisms to correct various runtime issues, the robustness during the execution of complex instructions remains insufficient. This is especially true for the response parser, which must generate correct code based solely on the response schema without actually accessing the response data.

6 Conclusion

In this paper, we explore the scenario of connecting current large language models (LLMs) with RESTful APIs, which is more relevant to real-world use cases. To overcome the limitations of existing approaches and tackle the challenges involved in integrating LLMs with RESTful APIs, we propose RestGPT, an approach that leverages LLMs to solve complex user requests. Our method features a coarse-to-fine online planning mechanism to improve planning and API selection. Furthermore, to handle the complex scenario of calling RESTful APIs, we designed a specialized API executor to formulate parameters and parse API responses. Experiments on two real-world scenarios demonstrate that RestGPT achieves impressive results in complex tasks and exhibits strong robustness. By exploiting the planning and text understanding capabilities of LLMs, RestGPT demonstrates its potential for addressing real-world queries with RESTful APIs. We believe that the integration of LLMs with existing systems has the potential to create unprecedented impact on both academia and industry.

Acknowledgment

Acknowledgment.

References

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

[2] OpenAI. Chatgpt, 2022. URL `https://openai.com/blog/chatgpt`.

[3] OpenAI. GPT-4 technical report. *ArXiv*, abs/2303.08774, 2023.

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

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

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

[7] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[8] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

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

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

[11] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*, 2023.

[12] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

[13] Li Li, Wu Chou, Wei Zhou, and Min Luo. Design patterns and extensibility of rest api for networking applications. *IEEE Transactions on Network and Service Management*, 13(1):154–167, 2016.

[14] SmartBear. Swagger, 2023. URL `https://swagger.io/`.

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

A Appendix