RestGPT: Connecting Large Language Models with Real-World Applications via 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`
https://restgpt.github.io

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 [1], mathematical reasoning [4, 5], and step-by-step planning [6, 7]. To extend the abilities of LLMs in real applications, an active research direction explores the use of external tools to augment LLMs [8, 9, 10]. Early studies equip LLMs with simple tools, such as searching engines and calculator, to access real-time information and enhance mathematical ability [11, 12, 10]. Recently, ViperGPT [13], Visual ChatGPT [14], and HuggingGPT [15] incorporate a collection of foundation models to make LLMs capable of processing multi-modal tasks. Chameleon [16] 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 [8]. As shown in Table 1, current methods can only connect with a small number of tools/APIs [12, 17, 13]. For example, Chameleon augments LLM with 15 different tools [16]. It is also noted that previous work heavily relies on specially designed APIs and meticulously crafted API descriptions, which in turn limits their extensibility [13, 18]. 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

---

[*] Corresponding author.

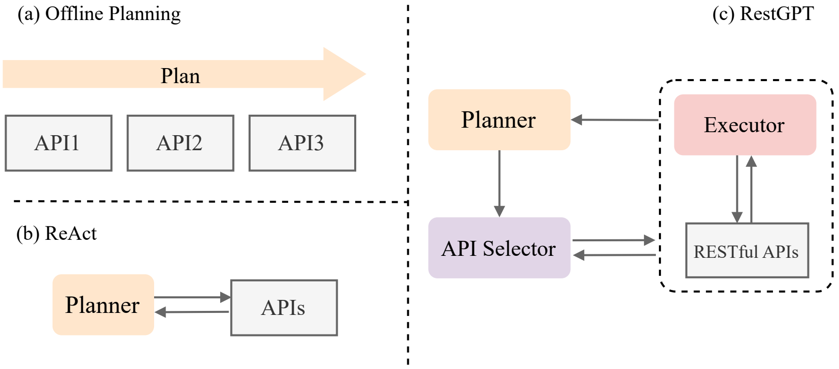


Figure 1: Comparison of RestGPT with previous work. (a) Offline-planning approaches, such as HuggingGPT [15], ViperGPT [13], Chameleon [16]. (b) ReAct [12], an online-planning approach. (c) Our proposed RestGPT, a coarse-to-fine online planning framework.

resources, which has become the *de facto* standard for web service development [19]. RESTful APIs usually follow OpenAPI Specification (OAS) [20], which describes the operations, parameters, and response schemas of each API endpoint. With the help of OAS, the resulting framework is naturally 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 [15, 13, 16]. 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 [12] 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. To ensure the model to fully comprehend API documentation within the constraint of limited input length, we assign different modules to read distinct parts of the OAS. 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 in the format of natural language, which are then mapped to 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 from 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, and music player. 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:

Model	API/Tool Use			Framework			
	Num.	Extensibility	Schema	Planning	Planning Form	Feedback	Plug-n-Play
ReAct [12]	3	−	Specialized	online	natural lang.	✓	✓
Toolformer [10]	5	−	Specialized	-	-	✗	✗
ART [17]	3	−	Specialized	online	natural lang.	✓	✓
Visual ChatGPT [14]	22	−	Specialized	-	natural lang.	human	✓
ViperGPT [13]	11	−	Python func.	offline	program	✗	✓
HuggingGPT [15]	24[1]	+	HuggingFace	offline	natural lang.	✗	✓
API-Bank [18]	53	−	Specialized	-	natural lang.	human	✓
Chameleon [16]	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 API/tool usage.

1. For the first time, we attempt to connect large language models with RESTful APIs, enabling the resulting framework to be compatible with existing real-world applications while also providing powerful extensibility.
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 [8]. 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 [12], Toolformer [10], and ART [17]. Another line of research has focused on equipping LLMs to coordinate with external models for complex AI tasks, exemplified by HuggingGPT [15], ViperGPT [13], and Visual ChatGPT [14]. Recently, Chameleon [16] incorporates 15 tools and augments LLMs with plug-and-play modules for compositional reasoning. Additionally, API-Bank [18] provides a systematic benchmark to showcase the efficacy of LLMs using tools to respond to human instructions.

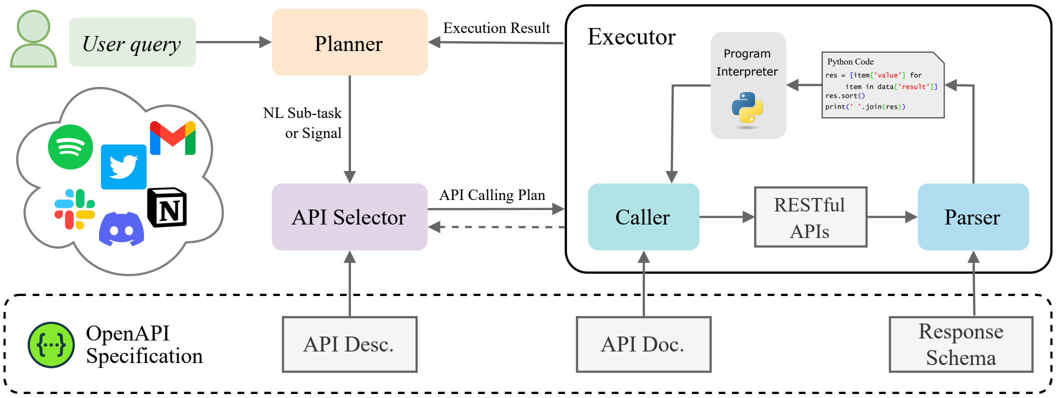
Despite the notable advancements in incorporating tools for large language models, previous methods have exhibited certain limitations, most notably their restricted support for a limited number of specially designed APIs [16] and their reliance on offline planning methods [8, 16]. We compare RestGPT with other tool-augmented language models in Table 1. As shown, our work stands out by supporting for over 100 APIs. By connecting with RESTful APIs, RestGPT also provides extraordinary extensibility for a plug-and-play integration with real-world applications. Furthermore, we employ a coarse-to-fine online planning framework with feedback, facilitating improved performance in planning and API usage when handling complex user queries. Our work shares the similar spirit of AutoGPT, an autonomous agent capable of accomplishing complex tasks with numerous tools. While AutoGPT relies on developers to ensure compatibility with various applications, RestGPT can be integrated with any RESTful API-based applications in a plug-and-play fashion.

2.2 RESTful APIs

RESTful APIs have become a popular way to expose functionalities and data of web services to client applications [21, 19]. 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 [21]. RESTful APIs also provide a standardized way of

---

[1] HuggingGPT [15] claims it has integrated hundreds of models on HuggingFace. However, all of the models only cover 24 tasks such as text classification, object detection, etc.



```
res = [item['value'] for
    item in data['result']]
res.sort()
print(' '.join(res))
```

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.

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 [20], 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:

- **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 A.

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.

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.

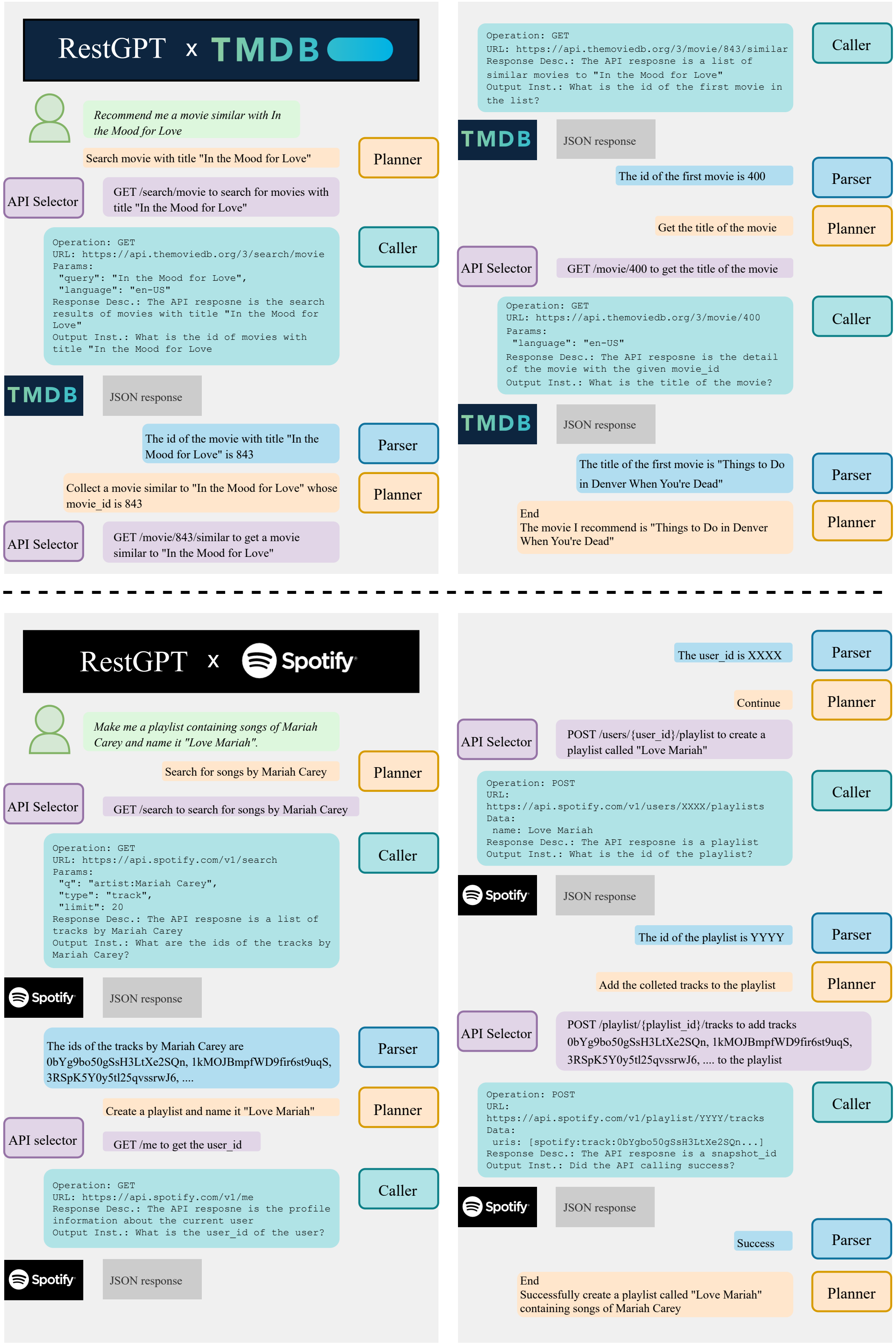
On the other hand, the planner dynamically plans the system's next actions based on the real-time results returned by the executor, which allows a more flexible online planning mechanism. It also plays a role of monitoring the whole process and outputs three kinds of signals: "continue", new NL sub-task 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 it. The executor consists of a caller and a response parser. In the execution stage, the caller should read the API documents carefully and extract 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", we only give the executor the document of the API "`GET /movie/{movie_id}`". Based on the calling plan and API documentation, the API caller generates API calling URL and parameters for the Requests Python library to call the corresponding API. Besides, to guide the response parser to extract information from the API response, the caller also generates a description for the response and output instruction for the response parser, e.g., "The API response is the crew list of a movie. Who is the director of the movie".

RESTful APIs typically return a response in JSON format when a request is received. The executor needs to extract the clean result with the format of natural language from the response and return it to the planner/API selector. As the response documents from different APIs have their own schemas, it is impossible to develop a specific parser for each API. Again we make extensive use of the capabilities



Recommend me a movie similar with In the Mood for Love

Search movie with title "In the Mood for Love"

GET /search/movie to search for movies with title "In the Mood for Love"

```
Operation: GET
URL: https://api.themoviedb.org/3/search/movie
Params:
 "query": "In the Mood for Love",
 "language": "en-US"
Response Desc.: The API response is the search
results of movies with title "In the Mood for
Love"
Output Inst.: What is the id of movies with
title "In the Mood for Love
```

JSON response

The id of the movie with title "In the Mood for Love" is 843

Collect a movie similar to "In the Mood for Love" whose movie_id is 843

GET /movie/843/similar to get a movie similar to "In the Mood for Love"

```
Operation: GET
URL: https://api.themoviedb.org/3/movie/843/similar
Response Desc.: The API response is a list of
similar movies to "In the Mood for Love"
Output Inst.: What is the id of the first movie in
the list?
```

JSON response

The id of the first movie is 400

Get the title of the movie

GET /movie/400 to get the title of the movie

```
Operation: GET
URL: https://api.themoviedb.org/3/movie/400
Params:
 "language": "en-US"
Response Desc.: The API response is the detail
of the movie with the given movie_id
Output Inst.: What is the title of the movie?
```

JSON response

The title of the first movie is "Things to Do in Denver When You're Dead"

End
The movie I recommend is "Things to Do in Denver When You're Dead"

Make me a playlist containing songs of Mariah Carey and name it "Love Mariah".

Search for songs by Mariah Carey

GET /search to search for songs by Mariah Carey

```
Operation: GET
URL: https://api.spotify.com/v1/search
Params:
 "q": "artist:Mariah Carey",
 "type": "track",
 "limit": 20
Response Desc.: The API response is a list of
tracks by Mariah Carey
Output Inst.: What are the ids of the tracks by
Mariah Carey?
```

JSON response

The ids of the tracks by Mariah Carey are 0bYg9bo50gSsH3LtXe2SQn, 1kMOJBmpfWD9fir6st9uqS, 3RSpK5Y0y5tl25qvssrwJ6, ...

Create a playlist and name it "Love Mariah"

API selector

GET /me to get the user_id

```
Operation: GET
URL: https://api.spotify.com/v1/me
Response Desc.: The API response is the profile
information about the current user
Output Inst.: What is the user_id of the user?
```

JSON response

The user_id is XXXX

Continue

POST /users/{user_id}/playlist to create a playlist called "Love Mariah"

```
Operation: POST
URL:
https://api.spotify.com/v1/users/XXXX/playlists
Data:
 name: Love Mariah
Response Desc.: The API response is a playlist
Output Inst.: What is the id of the playlist?
```

JSON response

The id of the playlist is YYYY

Add the colleted tracks to the playlist

POST /playlist/{playlist_id}/tracks to add tracks 0bYg9bo50gSsH3LtXe2SQn, 1kMOJBmpfWD9fir6st9uqS, 3RSpK5Y0y5tl25qvssrwJ6, ... to the playlist

```
Operation: POST
URL:
https://api.spotify.com/v1/playlist/YYYY/tracks
Data:
 uris: [spotify:track:0bYgbo50gSsH3LtXe2SQn...]
Response Desc.: The API resposne is a snapshot_id
Output Inst.: Did the API calling success?
```

JSON response

Success

End
Successfully create a playlist called "Love Mariah" containing songs of Mariah Carey

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

of LLMs. The response may sometimes have a complex structure or be lengthy, making it difficult to extract important information via directly prompting the LLMs. To address this problem, we make use of the response schema defined in the OAS. Instead of directly parsing the specific API response document, we utilize the coding capability of LLM to generate Python parsing code based on the provided schema and output instructions generated by the caller. Next, the Python code is executed to get the final result. If there are no execution exceptions or 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.

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[2]. We implement RestGPT based on LangChain[3]. 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 B.

We select the TMDB movie information database[4] and Spotify music player[5] 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 illustrates 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 NL 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. Next the planner generates the subsequent NL subtask and enters a new execution cycle.

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 APIs to solve each sub-task, and the caller attempts to accurately comprehend the API documents and formulate 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 [12] 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.

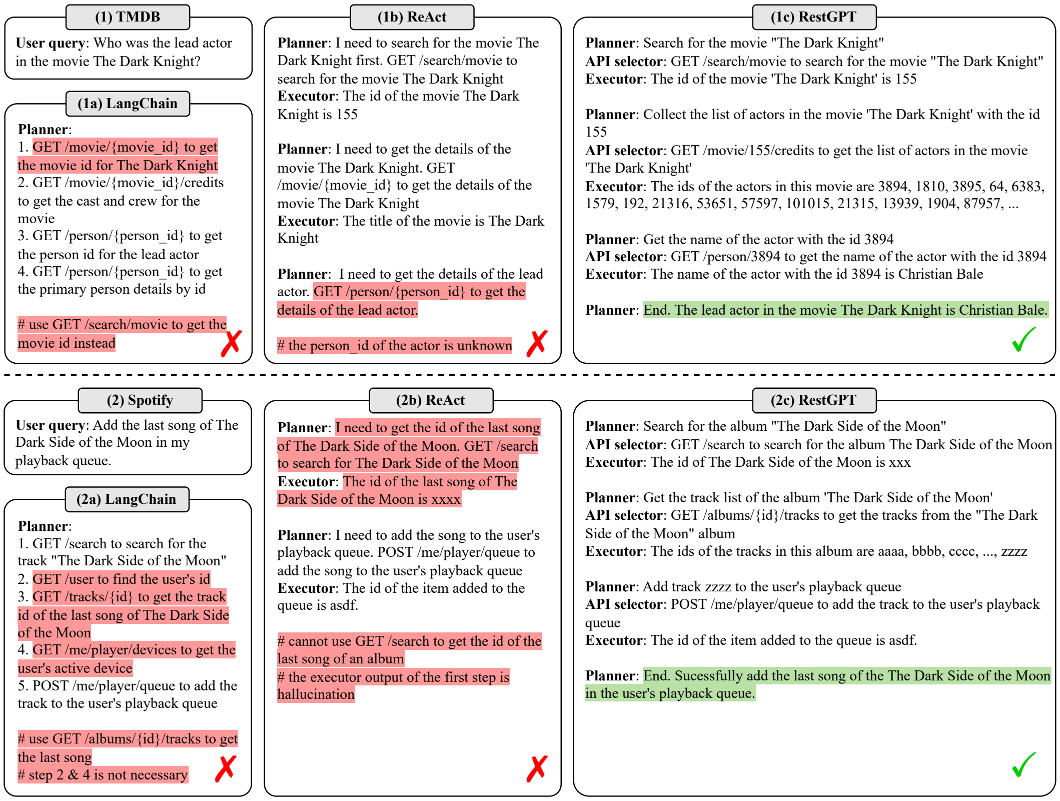
---

[2] https://platform.openai.com/

[3] https://github.com/hwchase17/langchain

[4] https://developer.themoviedb.org/docs

[5] https://developer.spotify.com/documentation/web-api]



(1) TMDB

User query: Who was the lead actor in the movie The Dark Knight?

(1a) LangChain

Planner:
1. GET /movie/{movie_id} to get the movie id for The Dark Knight
2. GET /movie/{movie_id}/credits to get the cast and crew for the movie
3. GET /person/{person_id} to get the person id for the lead actor
4. GET /person/{person_id} to get the primary person details by id

# use GET /search/movie to get the movie id instead ✗

(1b) ReAct

Planner: I need to search for the movie The Dark Knight first. GET /search/movie to search for the movie The Dark Knight
Executor: The id of the movie The Dark Knight is 155

Planner: I need to get the details of the movie The Dark Knight. GET /movie/{movie_id} to get the details of the movie The Dark Knight
Executor: The title of the movie is The Dark Knight

Planner: I need to get the details of the lead actor. GET /person/{person_id} to get the details of the lead actor.

# the person_id of the actor is unknown ✗

(1c) RestGPT

Planner: Search for the movie "The Dark Knight"
API selector: GET /search/movie to search for the movie "The Dark Knight"
Executor: The id of the movie 'The Dark Knight' is 155

Planner: Collect the list of actors in the movie 'The Dark Knight' with the id 155
API selector: GET /movie/155/credits to get the list of actors in the movie 'The Dark Knight'
Executor: The ids of the actors in this movie are 3894, 1810, 3895, 64, 6383, 1579, 192, 21316, 53651, 57597, 101015, 21315, 13939, 1904, 87957, ...

Planner: Get the name of the actor with the id 3894
API selector: GET /person/3894 to get the name of the actor with the id 3894
Executor: The name of the actor with the id 3894 is Christian Bale

Planner: End. The lead actor in the movie The Dark Knight is Christian Bale.

✓

(2) Spotify

User query: Add the last song of The Dark Side of the Moon in my playback queue.

(2a) LangChain

Planner:
1. GET /search to search for the track "The Dark Side of the Moon"
2. GET /user to find the user's id
3. GET /tracks/{id} to get the track id of the last song of The Dark Side of the Moon
4. GET /me/player/devices to get the user's active device
5. POST /me/player/queue to add the track to the user's playback queue

# use GET /albums/{id}/tracks to get the last song
# step 2 & 4 is not necessary ✗

(2b) ReAct

Planner: I need to get the id of the last song of The Dark Side of the Moon. GET /search to search for The Dark Side of the Moon
Executor: The id of the last song of The Dark Side of the Moon is xxxx

Planner: I need to add the song to the user's playback queue. POST /me/player/queue to add the song to the user's playback queue
Executor: The id of the item added to the queue is asdf.

# cannot use GET /search to get the id of the last song of an album
# the executor output of the first step is hallucination

✗

(2c) RestGPT

Planner: Search for the album "The Dark Side of the Moon"
API selector: GET /search to search for the album The Dark Side of the Moon
Executor: The id of The Dark Side of the Moon is xxx

Planner: Get the track list of the album 'The Dark Side of the Moon'
API selector: GET /albums/{id}/tracks to get the tracks from the "The Dark Side of the Moon" album
Executor: The ids of the tracks in this album are aaaa, bbbb, cccc, ..., zzzz

Planner: Add track zzzz to the user's playback queue
API selector: POST /me/player/queue to add the track to the user's playback queue
Executor: The id of the item added to the queue is asdf.

Planner: End. Sucessfully add the last song of the The Dark Side of the Moon in the user's playback queue.

✓

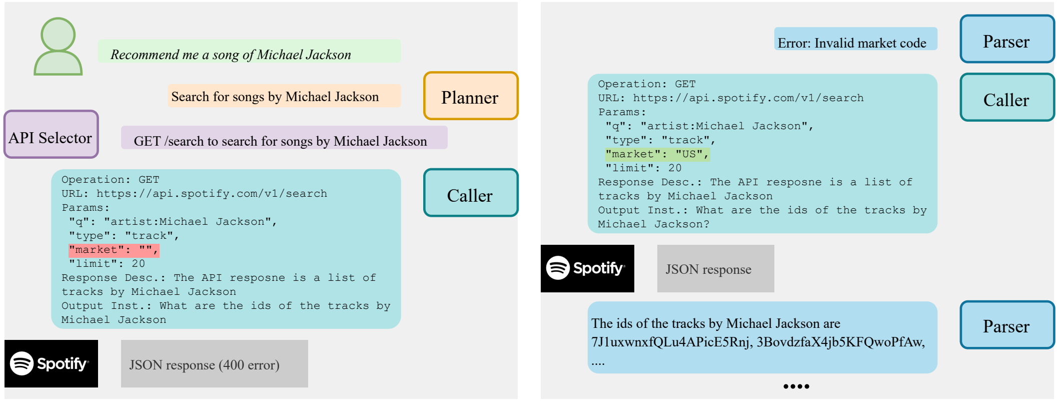
Figure 4: Comparison of three methods, (a) LangChain, (b) ReAct [12], and (c) RestGPT, connecting with TMDB and Spotify APIs to address complex user queries. For LangChain, we only show the generated plan. For ReAct and RestGPT, we omit the detailed execution process of the executor, and only show the output of the planner, API selector and the executor. RestGPT uses a planner to generate high-level NL sub-tasks and an API selector to choose appropriate APIs to solve the sub-task, formulating a coarse-to-fine planning process.

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, RestGPT employs a planner to generate high-level NL sub-tasks and an API selector to choose appropriate APIs to solve the sub-task. The coarse-to-fine online planning framework of RestGPT fully exploits the LLMs' planning and document understanding capabilities and provides the model with the flexibility to tackle complex user requests.

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



Recommend me a song of Michael Jackson

Search for songs by Michael Jackson

GET /search to search for songs by Michael Jackson

```
Operation: GET
URL: https://api.spotify.com/v1/search
Params:
 "q": "artist:Michael Jackson",
 "type": "track",
 "market": "",
 "limit": 20
Response Desc.: The API resposne is a list of
tracks by Michael Jackson
Output Inst.: What are the ids of the tracks by
Michael Jackson
```

JSON response (400 error)

Error: Invalid market code

```
Operation: GET
URL: https://api.spotify.com/v1/search
Params:
 "q": "artist:Michael Jackson",
 "type": "track",
 "market": "US",
 "limit": 20
Response Desc.: The API resposne is a list of
tracks by Michael Jackson
Output Inst.: What are the ids of the tracks by
Michael Jackson?
```

JSON response

The ids of the tracks by Michael Jackson are 7J1uxwnxfQLu4APicE5Rnj, 3BovdzfaX4jb5KFQwoPfAw, ...

••••

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.

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

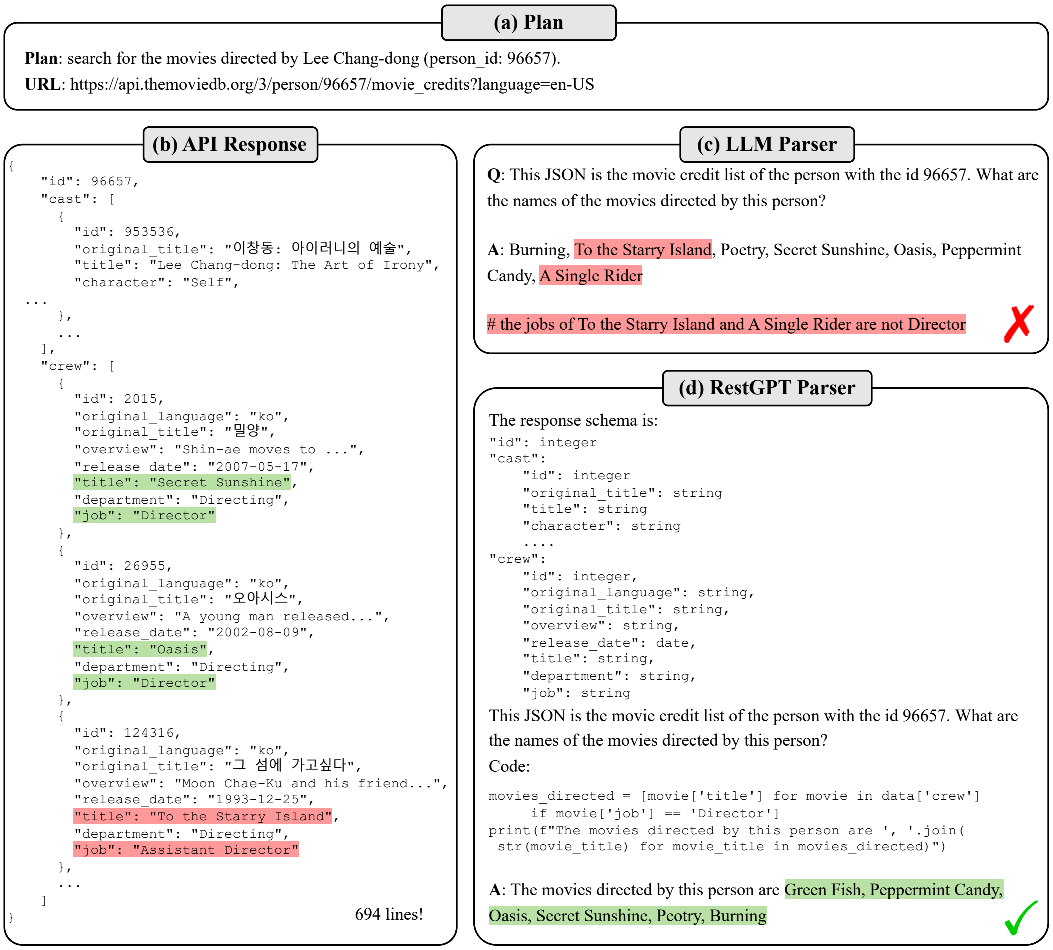
Figure 6 illustrates how RestGPT parses responses. We compare RestGPT parser with directly prompting an LLM as a parser. As shown, responses from RESTful APIs are usually in lengthy and complex JSON format (694 lines in Figure 6 (b)) The RestGPT parser generates information extraction code based on the 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 the core of each component in RestGPT 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.

6 Conclusion

In this paper, we explore the scenario of connecting current large language models (LLMs) with real-world application via RESTful APIs, which is more relevant to real-world use cases. To overcome the limitations of existing approaches and tackle the challenges 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 real-world applications has the potential to create unprecedented impact on both academia and industry.



(a) Plan

Plan: search for the movies directed by Lee Chang-dong (person_id: 96657).
URL: https://api.themoviedb.org/3/person/96657/movie_credits?language=en-US

(b) API Response

```
{
    "id": 96657,
    "cast": [
      {
        "id": 953536,
        "original_title": "이창동: 아이러니의 예술",
        "title": "Lee Chang-dong: The Art of Irony",
        "character": "Self",
  ...
      },
      ...
    ],
    "crew": [
      {
        "id": 2015,
        "original_language": "ko",
        "original_title": "밀양",
        "overview": "Shin-ae moves to ...",
        "release_date": "2007-05-17",
        "title": "Secret Sunshine",
        "department": "Directing",
        "job": "Director"
      },
      {
        "id": 26955,
        "original_language": "ko",
        "original_title": "오아시스",
        "overview": "A young man released...",
        "release_date": "2002-08-09",
        "title": "Oasis",
        "department": "Directing",
        "job": "Director"
      },
      {
        "id": 124316,
        "original_language": "ko",
        "original_title": "그 섬에 가고싶다",
        "overview": "Moon Chae-Ku and his friend...",
        "release_date": "1993-12-25",
        "title": "To the Starry Island",
        "department": "Directing",
        "job": "Assistant Director"
      },
      ...
    ]
}
```

694 lines!

(c) LLM Parser

Q: This JSON is the movie credit list of the person with the id 96657. What are the names of the movies directed by this person?

A: Burning, To the Starry Island, Poetry, Secret Sunshine, Oasis, Peppermint Candy, A Single Rider

# the jobs of To the Starry Island and A Single Rider are not Director ✗

(d) RestGPT Parser

The response schema is:

```
"id": integer
"cast":
    "id": integer
    "original_title": string
    "title": string
    "character": string
    ....
"crew":
    "id": integer,
    "original_language": string,
    "original_title": string,
    "overview": string,
    "release_date": date,
    "title": string,
    "department": string,
    "job": string
```

This JSON is the movie credit list of the person with the id 96657. What are the names of the movies directed by this person?

Code:

```
movies_directed = [movie['title'] for movie in data['crew']
     if movie['job'] == 'Director']
print(f"The movies directed by this person are ', '.join(
 str(movie_title) for movie_title in movies_directed)")
```

A: The movies directed by this person are Green Fish, Peppermint Candy, Oasis, Secret Sunshine, Peotry, Burning ✓

Figure 6: The detailed case study of the response parser.

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, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[5] 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.

[6] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[7] Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023.

[8] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

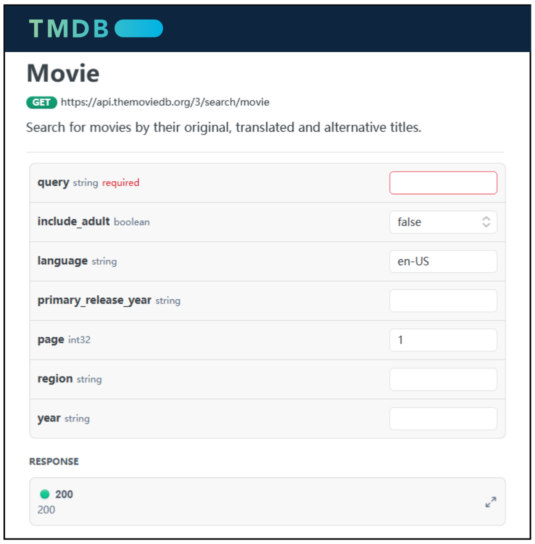
[18] Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.

[19] 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.

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

[21] Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.", 2011.

A Example of RESTful APIs and OAS



Movie

GET https://api.themoviedb.org/3/search/movie

Search for movies by their original, translated and alternative titles.

query string required	
include_adult boolean	false
language string	en-US
primary_release_year string	
page int32	1
region string	
year string	

RESPONSE

● 200
200

Figure 7: A RESTful API endpoint from TMDB.

OpenAPI Specification

```
description: Search for movies
operationId: GET_search-movie
parameters:
- description: Pass a ISO 639-1 value to display translated data for
the fields that support it.
  in: query
  name: language
  schema:
    default: en-US
    type: string
- description: Pass a text query to search. This value should be URI
encoded.
  in: query
  name: query
  required: true
  schema:
    type: string
- description: Specify which page to query.
  in: query
  name: page
  schema:
    default: 1
    type: integer
- description: Choose whether to inlcude adult (pornography) content
in the results.
  in: query
  name: include_adult
  schema:
    default: false
    type: boolean
- description: Specify a ISO 3166-1 code to filter release dates.
Must be uppercase.
  in: query
  name: region
  schema:
    type: string
- in: query
  name: year
  schema:
    type: integer
- in: query
  name: primary_release_year
  schema:
    type: integer
```

```
responses:
  '200':
    content:
      application/json:
        schema:
          properties:
            page:
              type: integer
            results:
              items:
                properties:
                  adult:
                    type: boolean
                  backdrop_path:
                    type: string
                  genre_ids:
                    items:
                      type: integer
                    type: array
                  id:
                    type: integer
                  original_language:
                    type: string
                  original_title:
                    type: string
                  overview:
                    type: string
                  popularity:
                    type: number
                  poster_path:
                    type: string
                  release_date:
                    type: string
                  title:
                    type: string
                  video:
                    type: boolean
                  vote_average:
                    type: number
                  vote_count:
                    type: integer
                title: Movie List Result Object
                type: object
              type: array
            total_pages:
              type: integer
            total_results:
              type: integer
          type: object
```

Figure 8: The OAS of the API endpoint in Figure 7

B More Implementation Details

In this section, we show the details of the prompt design in RestGPT. Some prompts are inspired by the OpenAPI agent implemented by LangChain[6].

▷ *Instruction for the planner in RestGPT*

You are an agent that plans solution to user queries. You should always give your plan in natural language. Another API selector model will receive your plan and find the right API calls and finally give you the result in natural language. If you assess that the current plan has not been fulfilled, you can output "continue" to let the API selector select another API to fulfill the plan. If you think you have got the final answer, you should first output "End" and output the answer immediately. In most case, search, filter, and sort should be completed in a single step.

Starting below, you should follow this format:

User query: the query a User wants help with related to the API
Plan step 1: the first step of your plan for how to solve the query
Result 1: the result of executing the first step of your plan
Plan step 2: based on the API response, the second step of your plan for how to solve the query. If you assess the plan has not been fulfilled, you can output "continue"
API response: the result of executing the second step of your plan
... (this Plan step n and API response can repeat N times)
Thought: I am finished executing a plan and have the information the user asked for or the data the used asked to create
Final Answer: the final output from executing the plan

Example:
`{in-context examples}`

Begin!

User query: `{query}`
Plan step 1:

Table 2: The prompt of the planner.

C More Cases

---

[6] https://python.langchain.com/en/latest/modules/agents/toolkits/examples/openapi.html

▷ *Instruction for the API selector in RestGPT*

You are an API selector that selects a sequence of RESTful API calls to tackle the queries. Another API caller will receive your plan call the corresponding APIs and finally give you the result in natural language. The API caller also has filtering, sorting functions to post-process the response of APIs. Therefore, if you think the API response should be post-processed, just tell the API caller to do so.

Here are name and description of available APIs. Do not use APIs that are not listed here.

`{endpoints}`

Starting below, you should follow this format:

Background: background information which you can use to execute the plan, e.g., the id of a person, the id of tracks by Faye Wong. In most cases, you must use the background information instead of requesting these information again.
Task: the task you should tackle with the given APIs
API calling 1: the first api call you want to make
API response: the response of API calling 1
... (this API calling n and API response can repeat N times, but most queries can be solved in 1-2 step)

Examples:

`{in-context examples}`

Note, if the API path contains "`{}`", it means that it is a variable and you should replace it with the appropriate value. For example, if the path is "`/users/{user_id}/tweets`", you should replace "`{user_id}`" with the user id. "`{`" and "`}`" cannot appear in the url. In most cases, the id value is in the background or the API response. If you do not know the id, please call an API to get it.

Begin!

Background: `{background}`
Task: `{plan}`
API calling 1:

Table 3: The prompt of the API selector.

▷ *Instruction for the caller in RestGPT*

You are an agent that gets a API calling plan and the OAS documentation of the API. Your task is to complete the corresponding api calls according to the plan. More specifically, you should generate the parameters or data for the API calling. You should also generate the description of the API response and what information you want to extract from the response. If the API response contains error information, you can reorganize the parameters and retry the API call.

Here is documentation on the API:
Base url: `{api_url}`
Endpoints:
`{api_docs}`

If the API path contains "`{}`", it means that it is a variable and you should replace it with the appropriate value. For example, if the path is "`/users/{user_id}/tweets`", you should replace "`{user_id}`" with the user id. "`{`" and "`}`" cannot appear in the url.

You can use http request method, i.e., `GET`, `POST`, `DELETE`, `PATCH`, `PUT`, and generate the corresponding parameters according to the API documentation and the plan.
The input should be a json string which has 3 base keys: url, description, output_instructions.
The value of "url" should be a string.
The value of "description" should describe what the API response is about.
The value of "output_instructions" should be instructions on what information to extract from the response, for example the id(s) for a resource(s) that the POST request creates. Note "output_instructions" must be natural language.
If you are using `GET` method, add "params" key.
If you are using `POST`, `PATCH` or `PUT` methods, add "params" and "data" key. The value of "params" and "data" should be a list of key-value pairs.

Example:
Operation: `POST`
Input:

```
{
    "url": "https://api.twitter.com/2/tweets",
    "params": {"tweet.fields": "created_at"}
    "data": {"text": "Hello world!"},
    "description": "The API reponse is a twitter object.",
    "output_instructions": "What is the URL of the new twitter?"
}
```

Starting below, you should follow this format:

Background: background information which you can use to execute the plan, e.g., the id of a person.
API calling plan: the plan of API calls to execute
Thought: you should always think about what to do
Operation: one of `GET`, `POST`, `DELETE`, `PATCH`, `PUT`
Input: the input to the action
Response: the output of the action
... (this Thought/Operation/Input/Response can repeat N times)

Begin!

Background: `{background}`
API calling plan: `{plan}`
Thought:

Table 4: The prompt of the API caller.

▷ *Instruction for the response parser in RestGPT*

Here is an API response schema from an OAS and a query. The API's response will follow the schema and be in JSON format. Assume you are given a JSON response which is stored in a python dict variable called "data", your task is to generate Python code to extract information I need from the API response. Note: I will give you "data", do not make up one, just reference it in your code
Please print the final result as brief as possible. If the result is a list, just print it in one sentence. Do not print each item in a new line.
The example output formats are:
"The release date of the album is 2002-11-03"
"The id of the person is 12345"
"The movies directed by Wong Kar-Wai are In the Mood for Love (843), My Blueberry Nights (1989), Chungking Express (11104)"
Note you should generate only Python code. Do not use fields that are not in the response schema.

API: `{api_path}`
API description: `{api_description}`
Parameters or reqeust body for this API call:
`{api_param}`

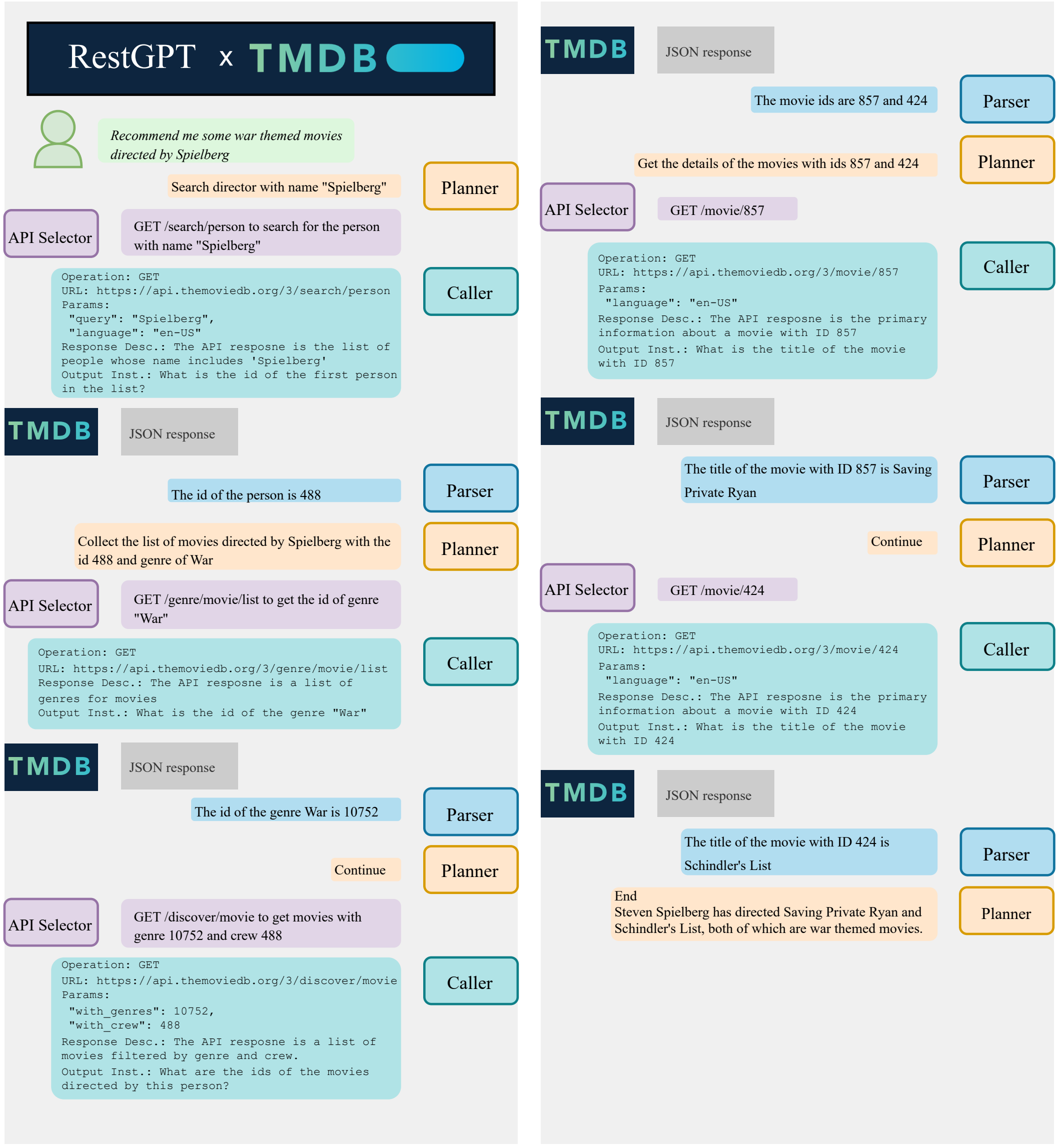
Response schema:
`{response_schema}`

Example:
`{response_example}`

The response is about: `{response_description}`

Query: `{query}`
Python Code:

Table 5: The prompt of the response parser.



Recommend me some war themed movies directed by Spielberg

Search director with name "Spielberg"

GET /search/person to search for the person with name "Spielberg"

```
Operation: GET
URL: https://api.themoviedb.org/3/search/person
Params:
 "query": "Spielberg",
 "language": "en-US"
Response Desc.: The API resposne is the list of
people whose name includes 'Spielberg'
Output Inst.: What is the id of the first person
in the list?
```

The id of the person is 488

Collect the list of movies directed by Spielberg with the id 488 and genre of War

GET /genre/movie/list to get the id of genre "War"

```
Operation: GET
URL: https://api.themoviedb.org/3/genre/movie/list
Response Desc.: The API resposne is a list of
genres for movies
Output Inst.: What is the id of the genre "War"
```

The id of the genre War is 10752

Continue

GET /discover/movie to get movies with genre 10752 and crew 488

```
Operation: GET
URL: https://api.themoviedb.org/3/discover/movie
Params:
 "with_genres": 10752,
 "with_crew": 488
Response Desc.: The API resposne is a list of
movies filtered by genre and crew.
Output Inst.: What are the ids of the movies
directed by this person?
```

The movie ids are 857 and 424

Get the details of the movies with ids 857 and 424

GET /movie/857

```
Operation: GET
URL: https://api.themoviedb.org/3/movie/857
Params:
 "language": "en-US"
Response Desc.: The API resposne is the primary
information about a movie with ID 857
Output Inst.: What is the title of the movie
with ID 857
```

The title of the movie with ID 857 is Saving Private Ryan

Continue

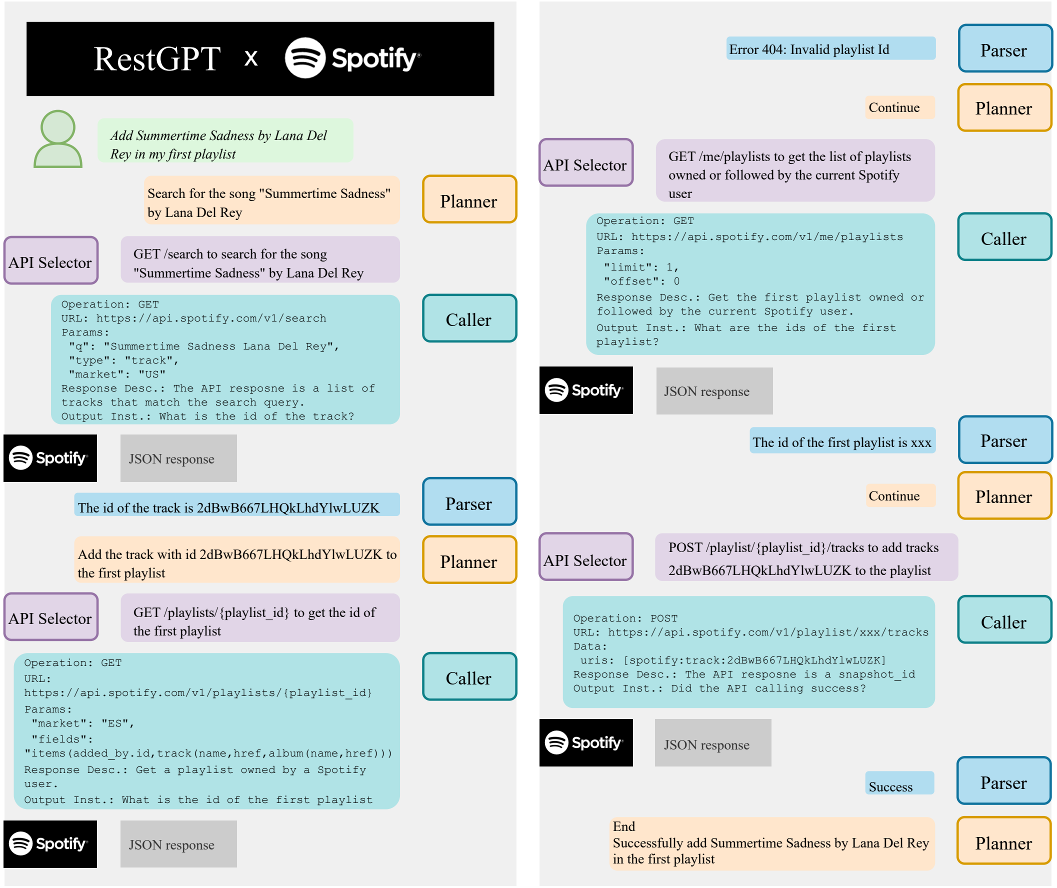
GET /movie/424

```
Operation: GET
URL: https://api.themoviedb.org/3/movie/424
Params:
 "language": "en-US"
Response Desc.: The API resposne is the primary
information about a movie with ID 424
Output Inst.: What is the title of the movie
with ID 424
```

The title of the movie with ID 424 is Schindler's List

End
Steven Spielberg has directed Saving Private Ryan and Schindler's List, both of which are war themed movies.

Figure 9: An example of connecting RestGPT with TMDB.



Add Summertime Sadness by Lana Del Rey in my first playlist

Search for the song "Summertime Sadness" by Lana Del Rey

GET /search to search for the song "Summertime Sadness" by Lana Del Rey

```
Operation: GET
URL: https://api.spotify.com/v1/search
Params:
 "q": "Summertime Sadness Lana Del Rey",
 "type": "track",
 "market": "US"
Response Desc.: The API resposne is a list of
tracks that match the search query.
Output Inst.: What is the id of the track?
```

JSON response

The id of the track is 2dBwB667LHQkLhdYlwLUZK

Add the track with id 2dBwB667LHQkLhdYlwLUZK to the first playlist

GET /playlists/{playlist_id} to get the id of the first playlist

```
Operation: GET
URL:
https://api.spotify.com/v1/playlists/{playlist_id}
Params:
 "market": "ES",
 "fields":
"items(added_by.id,track(name,href,album(name,href)))
Response Desc.: Get a playlist owned by a Spotify
user.
Output Inst.: What is the id of the first playlist
```

JSON response

Error 404: Invalid playlist Id

Continue

GET /me/playlists to get the list of playlists owned or followed by the current Spotify user

```
Operation: GET
URL: https://api.spotify.com/v1/me/playlists
Params:
 "limit": 1,
 "offset": 0
Response Desc.: Get the first playlist owned or
followed by the current Spotify user.
Output Inst.: What are the ids of the first
playlist?
```

JSON response

The id of the first playlist is xxx

Continue

POST /playlist/{playlist_id}/tracks to add tracks 2dBwB667LHQkLhdYlwLUZK to the playlist

```
Operation: POST
URL: https://api.spotify.com/v1/playlist/xxx/tracks
Data:
 uris: [spotify:track:2dBwB667LHQkLhdYlwLUZK]
Response Desc.: The API resposne is a snapshot_id
Output Inst.: Did the API calling success?
```

JSON response

Success

End
Successfully add Summertime Sadness by Lana Del Rey in the first playlist

Figure 10: An example of connecting RestGPT with Spotify.