ToolBench Learning Survey

ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs

https://arxiv.org/abs/2307.16789

arxiv.org

💡 Abstract

Although the development of open source large language models (LLMs) has made great progress, such as LLaMA, they still have significant limitations in tool usage ability, that is, they cannot use external tools (APIs) to complete human instructions. The reason is that the current instruction fine-tuning mainly focuses on basic language tasks and ignores the field of tool usage. This is in sharp contrast to the excellent tool usage ability of advanced closed-source LLMs (such as ChatGPT). To bridge this gap, we introduce **ToolLLM** , a general tool usage framework that covers data construction, model training, and evaluation.

First, we propose **ToolBench** , a dataset for fine-tuning instructions used by tools, which is built automatically using ChatGPT. Specifically, the build can be divided into three stages:

(I) API Collection: We have collected 16,464 real-world RESTful APIs from RapidAPI Hub, covering 49 categories.

(Ii) Instruction generation: We use ChatGPT to generate various instructions related to these APIs, including cases of single tools and multiple tools.

(Iii) Solution path annotation: We use ChatGPT to find the valid solution path (API call chain) for each instruction.

In order to enhance the inference ability of LLMs, we have developed a novel algorithm based on depth-first search Decision Tree. It allows LLMs to evaluate multiple inference trajectories and expand the search space. In addition, to evaluate the tool usage ability of LLMs, we have developed an automatic evaluator: ToolEval.

Based on ToolBench, we obtained an LLM with a neural API retriever by fine-tuning **LLaMA** to recommend appropriate APIs for each instruction. Experiments show that ToolLLaMA has excellent ability to execute complex instructions and generalize to

unseen APIs, and exhibits performance similar to ChatGPT. Our ToolLLaMA also demonstrates strong zero-sample generalization ability on an out-of-distribution tool using the dataset APIBench. The code, trained model, and demonstration are publicly available in https://github.com/OpenBMB/ToolBench .

Introduction and Background

While open-source LLMs such as LLaMA (Touvron et al., 2023a) achieve diverse capabilities through instruction fine-tuning (Taori et al., 2023; Chiang et al., 2023), they **still lack complexity in performing high-level tasks** , such as appropriately interacting with tools (APIs) to complete complex human instructions.

- **This shortcoming is due to the fact that current instruction fine-tuning focuses on basic language tasks and relatively ignores the domain of using tools** .
- At the same time, today's relatively mature large language models (such as GPT-4) have not yet been open source, and **there is a lack of open source tools in the industry.**

Construction of ToolBench

In order to promote the tool usage ability in open source LLM, we introduced ToolLLM, a general tool usage framework that includes data construction, Model Training, and evaluation. As shown in Figure 1, we collected high-quality instruction fine-tuning dataset ToolBench. It was automatically built using ChatGPT (gpt-3.5-turbo-16k) with function call (link) function upgrade. Table 1 lists the comparison between ToolBench and previous work. Specifically, the construction of ToolBench includes three stages.

- API Collection: We have collected 16,464 Representation State Transfer (REST) APIs from the RapidAPI platform (link), which hosts many real-world APIs provided by developers. These APIs cover 49 different categories, such as social media, e-commerce, and weather. For each API, we crawl detailed API documentation from RapidAPI, including function descriptions, required parameters, API call code snippets, etc. By understanding these documents to learn to execute APIs, LLM can generalize to new APIs that have not been seen during training.

- Instruction Generation: We first sample APIs from the entire collection, and then prompt ChatGPT to generate various instructions for these APIs. To cover practical scenarios, we organize instructions involving single tools and multiple tool situations. This ensures that our model not only learns how to interact with individual tools, but also how to combine them to complete complex tasks.

- Solution Path Annotation: Each solution path may contain multiple rounds of model inference and real-time API calls to derive the final response. However, even the most advanced LLM, GPT-4, has a low pass rate for complex human instructions, making annotation inefficient. To this end, we have developed a novel Depth-First Search Decision Tree (DFSDT) to enhance LLM's planning and inference capabilities. Compared with traditional ReACT, DFSDT enables LLM to evaluate multiple inference paths and make wise decisions, withdraw steps or continue along promising paths. In experiments, DFSDT significantly improved annotation efficiency and successfully completed complex instructions that could not be satisfied using ReACT.

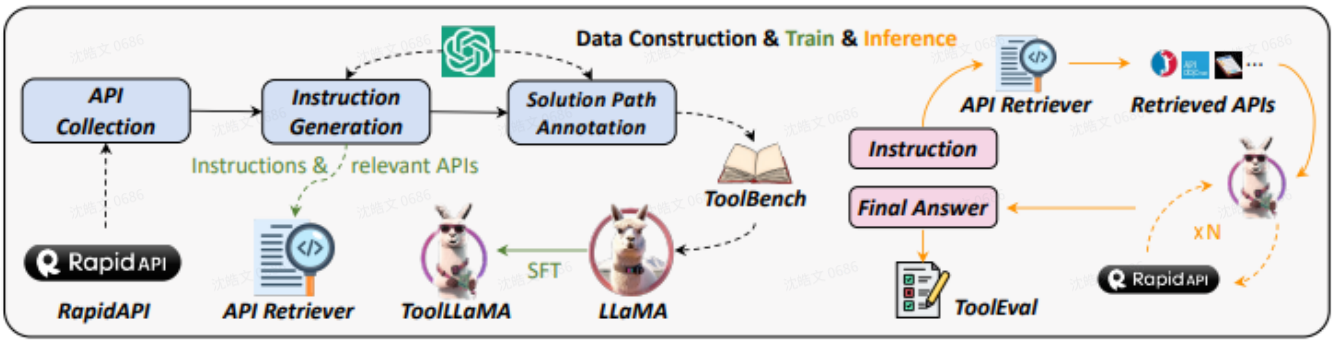

Figure 1: Three phases of constructing ToolBench and how we train our API retriever and ToolLLaMA. During inference of an instruction, the API retriever recommends relevant APIs to ToolLLaMA, which performs multiple rounds of API calls to derive the final answer. The whole reasoning process is evaluated by ToolEval.

ToolEval

In order to evaluate the tool usage ability of LLM, we have developed an automatic evaluator **ToolEval** that supports ChatGPT. It includes two key indicators: (1) pass rate, which measures the LLM's ability to successfully execute instructions within a limited budget;

(2) Win rate, comparing the quality and usefulness of two solution paths. We have proven that ToolEval is highly correlated with human evaluation and provides a robust, scalable, and reliable machine tool usage evaluation.

Data Construction

API Collection

First, we introduced the RapidAPI and its hierarchy, and then talked about how we crawl and filter APIs.

- **RapidAPI Hub** Leading API marketplace, it connects developers with thousands of real-world APIs, simplifying the process of integrating various services into applications. Developers can test and connect various APIs by registering a RapidAPI key. APIs in all RapidAPIs can be grouped into 49 rough categories (links), such as sports, finance, and weather. These categories associate APIs with the most relevant topics. In addition, the hub provides more than 500 fine classifications (links) called collections, such as Chinese APIs and database APIs. APIs in the same collection share a common feature and often have similar functions or goals.
- **Hierarchy of RapidAPIs** As shown in Figure 3, each tool can consist of multiple APIs. For each tool, we grab the following information: the name and description of the tool, the URL of the host, and all available APIs belonging to that tool; for each API, we record its name, description, HTTP method, required parameters, optional parameters, request body, executable code snippets, and examples of API call responses. **These rich and detailed Metadata provide valuable resources for LLM to understand and effectively use APIs, even with zero samples.**

- **API Filtering** Initially, we collected 10,853 tools (53,190 APIs) from RapidAPI. However, the quality and reliability of these APIs may vary greatly. In particular, some APIs may not be well maintained, such as returning 404 errors or other internal errors. To this end, we performed a rigorous filtering process (details in Appendix A.1) to ensure that the final toolset of ToolBench is reliable and fully functional. In the end, we only retained 3,451 high-quality tools (16,464 APIs).

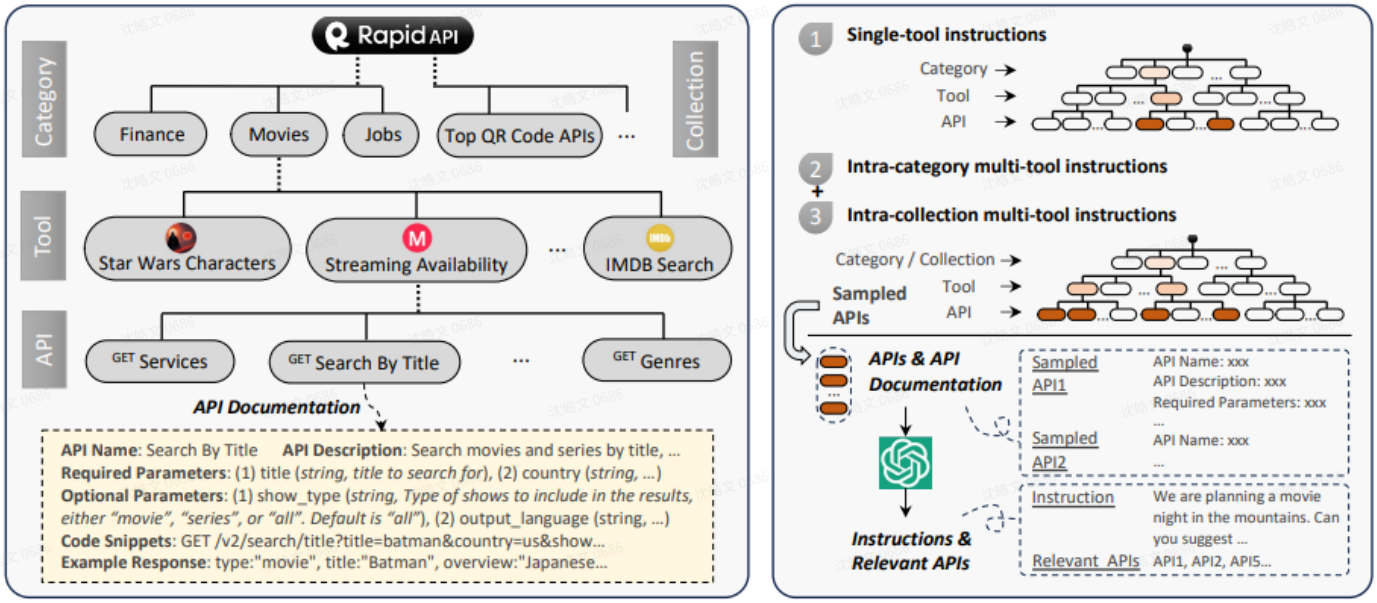

Figure 3: The hierarchy of RapidAPI (left) and the process of instruction generation (right).

Instruction Generation

Unlike previous work, we focus on two key aspects of instruction generation: (1) diversity: training LLMs to handle various API usage scenarios, thereby improving their generalization and robustness; and (2) multi-tool usage: reflecting real-world situations, often requiring the interaction of multiple tools, thereby improving the practicality and flexibility of LLMs. **To do this, instead of conceptualizing instructions from scratch and then searching for relevant APIs, we sample different combinations of APIs from the total SAPI collection and write multiple instructions involving them.**

Generating Instructions for APIs Define the total API set as $\mathbb{S}_{\text{API}}$, at each time, we sample a few APIs: $\mathbb{S}_{\text{N}}^{\text{sub}} = \{\text{API}_1, \cdots, \text{API}_{\text{N}}\}$ from $\mathbb{S}_{\text{API}}$. We prompt ChatGPT to understand the functionalities of these APIs and then generate (1) possible instructions (Inst_*) that involve APIs in $\mathbb{S}_{\text{N}}^{\text{sub}}$, and (2) relevant APIs ($\mathbb{S}_*^{\text{rel}} \subset \mathbb{S}_{\text{N}}^{\text{sub}}$) for each instruction ($\text{Inst}_*$), i.e., $\{[\mathbb{S}_1^{\text{rel}}, \text{Inst}_1], \cdots, [\mathbb{S}_{\text{N}'}^{\text{rel}}, \text{Inst}_{\text{N}'}]\}$, where N' denotes the number of generated instances. These (instruction, relevant API) pairs will be used for

The tips for ChatGPT include (1) a general description of the intent generation instructions, (2) comprehensive documentation for each API in SsubN to help ChatGPT understand its functionality and interactions, and (3) three contextual seed examples {seed1, seed2, seed3}. Each seed example is an ideal instruction generation written by human experts. These seed

examples better regulate the behavior of ChatGPT through contextual learning. In total, we wrote 12/36 diverse seed examples (Sseeds) for single-tool/multi-tool settings and randomly sampled three examples each time.

Examples of ChatGPT prompts

> **Task Description of Single-tool Instructions:**
>
> " You will be provided with a tool, its description, all of the tool’ s available API functions, the descriptions of these API functions, and the parameters required for each API function. Your task involves creating 10 varied, innovative, and detailed user queries that employ multiple API functions of a tool. For instance, if the tool ‘climate news’ has three API calls - ‘get all climate change news’ , ‘look up climate today’ , and ‘historical climate’ , your query should articulate something akin to: first, determine today’ s weather, then verify how often it rains in Ohio in September, and finally, find news about climate change to help me understand whether the climate will change anytime soon. This query exemplifies how to utilize all API calls of ‘climate news’ . A query that only uses one API call will not be accepted. Additionally, you must incorporate the input parameters required for each API call. To achieve this, generate random information for required parameters such as IP address, location, coordinates, etc. For instance, don’ t merely say ‘an address’ , provide the exact road and district names. Don’ t just mention ‘a product’ , specify wearables, milk, a blue blanket, a pan, etc. Don’ t refer to ‘my company’ , invent a company name instead. The first seven of the ten queries should be very specific. Each single query should combine all API call usages in different ways and include the necessary parameters. Note that you shouldn’ t ask ‘which API to use’ , rather, simply state your needs that can be addressed by these APIs. You should also avoid asking for the input parameters required by the API call, but instead directly provide the parameter in your query. The final three queries should be complex and lengthy, describing a complicated scenario where all the API calls can be utilized to provide assistance within a single query. You should first think about possible related API combinations, then give your query. Related apis are apis that can be used for a give query; those related apis have to strictly come from the provided api names. For each query, there should be multiple related apis; for different queries, overlap of related apis should be as little as possible. Deliver your response in this format: [Query1: ....., ‘related apis’ :[api1, api2, api3...],Query2: ....., ‘related apis’ :[api4, api5, api6...],Query3: ....., ‘related apis’ :[api1, api7, api9...], ...]"

> **Task Description of Multi-tool Instructions:**
>
> "You will be provided with several tools, tool descriptions, all of each tool’ s available API functions, the descriptions of these API functions, and the parameters required for each API function. Your task involves creating 10 varied, innovative, and detailed user queries that employ API functions of multiple tools. For instance, given three tools ‘nba news’ , ‘cat-facts’ , and ‘hotels’ : ‘nba news’ has API functions ‘Get individual NBA source news’

> and ‘Get all NBA news’, ‘cat-facts’ has API functions ‘Get all facts about cats’ and ‘Get a random fact about cats’, ‘hotels’ has API functions ‘properties/get-details (Deprecated)’, ‘properties/list (Deprecated)’ and ‘locations/v3/search’. Your query should articulate something akin to: ‘I want to name my newborn cat after Kobe and host a 17 Preprint party to celebrate its birth. Get me some cat facts and NBA news to gather inspirations for the cat name. Also, find a proper hotel around my house in Houston Downtown for the party.’ This query exemplifies how to utilize API calls of all the given tools. A query that uses API calls of only one tool will not be accepted. Additionally, you must incorporate the input parameters required for each API call. To achieve this, generate random information for required parameters such as IP address, location, coordinates, etc. For instance, don’t merely say ‘an address’, provide the exact road and district names. Don’t just mention ‘a product’, specify wearables, milk, a blue blanket, a pan, etc. Don’t refer to ‘my company’, invent a company name instead. The first seven of the ten queries should be very specific. Each single query should combine API calls of different tools in various ways and include the necessary parameters. Note that you shouldn’t ask ‘which API to use’, rather, simply state your needs that can be addressed by these APIs. You should also avoid asking for the input parameters required by the API call, but instead directly provide the parameters in your query. The final three queries should be complex and lengthy, describing a complicated scenario where all the provided API calls can be utilized to provide assistance within a single query. You should first think about possible related API combinations, then give your query. Related APIs are APIs that can be used for a given query; those related APIs have to strictly come from the provided API names. For each query, there should be multiple related APIs; for different queries, overlap of related APIs should be as little as possible. Deliver your response in this format: [Query1: ....., ‘related apis’:[[tool name, api name], [tool name, api name], [tool name, api name]...],Query2: ....., ‘related apis’:[[tool name, api name], [tool name, api name], [tool name, api name]...],Query3: ....., ‘related apis’:[[tool name, api name], [tool name, api name], [tool name, api name]...], ...]"

Solution Path Annotation

As shown in Figure 4, given the instruction Inst *, we prompt ChatGPT to search for a valid operation sequence: {a1,..., aN}. This multi-step decision-making process is regarded as a multi-round dialogue of ChatGPT. In each round t, the model generates an operation at based on the previous interaction, namely ChatGPT (at | {a1, r1,..., at-1, rt-1}, Inst *), where r * represents the real API response.

In order to utilize the function call function of ChatGPT, we treat each API as a special function and input its API document into the function field of ChatGPT. This way, the model knows how to call the API. For each instruction Inst *, we input all the sampled APIs into the available functions of ChatGPT. In order for ChatGPT to complete a sequence of operations, we define two

additional functions, namely "Finish Final Answer" and "Abandon Finish". The former has a parameter that corresponds to the detailed final answer of the original instruction; the latter is suitable for situations where the provided API cannot complete the original instruction after multiple API call attempts.

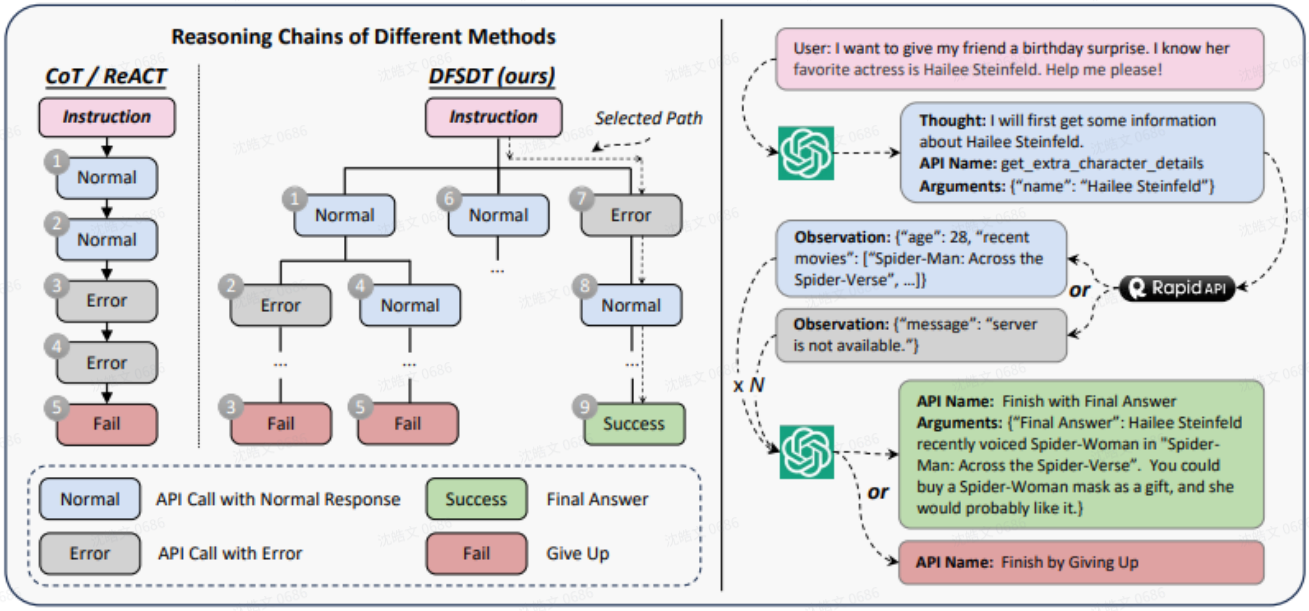

Figure 4: A comparison of our DFSDT and conventional CoT or ReACT during model reasoning (left). We show part of the solution path annotation process using ChatGPT (right).

Depth First Search-based Decision Tree

In our preliminary study, we found that either CoT (Wei et al., 2023) or ReACT (Yao et al., 2022) has inherent limitations.

Error propagation: Incorrect operations may further propagate errors, causing the model to fall into an error loop, such as continuously calling APIs or fantasy APIs in the wrong way.

Limited exploration: CoT or ReACT only explore one possible direction, which limits the exploration of the entire operation space. Therefore, even GPT-4 has difficulty finding an effective solution path, making annotation difficult.

To this end, we propose to build a Decision Tree to expand the search space and increase the likelihood of finding a valid path. As shown in Figure 4, our DFSDT allows the model to evaluate different inference paths and choose one of the following: (1) continue along a promising path or (2) abandon the current node and expand a new node by calling the "abandon completion" function. During node expansion, in order to diversify sub-nodes and expand the search space, we prompt ChatGPT to condition the information of previously generated nodes and explicitly encourage the model to generate a different node. During the search process, we prefer Depth-

First Search (DFS) to Breadth-First Search (BFS) because annotations can be completed as long as a valid path is found. Using BFS will consume too many OpenAI API calls. See Appendix A.8 for more details. We perform DFSDT on all generated instructions and only keep the solution paths that have passed. In the end, we generate 126,486 (instruction, solution path) pairs for training ToolLLaMA in Section 3.2.

Achievement

By fine-tuning LLaMA on ToolBench, we obtain ToolLLaMA. Based on our evaluation of ToolEval, we make the following findings:

ToolLLaMA demonstrates the powerful ability to handle single-tool and complex multi-tool instructions. ToolLLaMA outperforms Text-Davinci-003 and Claude-2, achieving performance comparable to the "teacher model" ChatGPT, and only slightly improving compared to GPT4. In addition, ToolLLaMA shows strong generalization ability for things that have not been seen before. **Only API documentation is needed to effectively adapt to new APIs** This flexibility allows users to seamlessly integrate novel APIs, thereby enhancing the practicality of the model.

· We demonstrate the ability of our DFSDT to serve as a general decision strategy LLM for enhanced inference. **DFSDT broadens the search space by considering multiple inference trajectories and achieves significantly better performance than ReACT.**

We trained a neural API retriever, which reduces the need for manual selection from large datasets. API pool in practice. As shown in Figure 1, given an instruction, the API retriever will recommend a set of related APIs, send them to ToolLLaMA for multiple rounds of decision-making, and obtain the final answer. Despite filtering a large number of APIs, the retriever still exhibits extraordinary performance and retrieval accuracy, and the returned APIs are closely consistent with the real situation.

ToolLLaMA exhibits strong generalization performance on out-of-distribution (OOD) datasets in APIBench (Patil et al., 2023). Despite not receiving any API training or instructions on APIBench, ToolLLaMA's performance is comparable to Gorilla, which is a pipeline designed specifically for APIBench.

Reference

- React
- PAL

- https://www.linkedin.com/pulse/i-got-99-problems-steering-llms-aint-one-function-calls-leonard-park
- https://www.linkedin.com/pulse/exploring-function-calling-openais-gpt-models-nicola-lazzari
- https://www.linkedin.com/pulse/prompt-engineering-getting-what-you-want-from-chatgpt-rick-hightower