An Evaluation Mechanism of LLM-based Agents on Manipulating APIs

Bing Liu[*], **Jianxiang Zhou**[†], **Dan Meng**[*], **Haonan Lu**
OPPO AI Center, Shenzhen, China
liubing.csai@gmail.com, mengdan90@163.com, luhaonan@oppo.com

Abstract

LLM-based agents can greatly extend the abilities of LLMs and thus attract sharply increased studies. An ambitious vision – serving users by manipulating massive API-based tools – has been proposed and explored. However, we find a widely accepted evaluation mechanism for generic agents is still missing. This work aims to fill this gap. We decompose tool use capability into seven aspects and form a thorough evaluation schema. In addition, we design and release an instruction dataset and a toolset – the two sides that the agents bridge between – following the principle of reflecting real-world challenges. Furthermore, we evaluate multiple generic agents. Our findings can inspire future research in improving LLM-based agents and rethink the philosophy of API design.

1 Introduction

Large Language Models (LLMs) exhibit remarkable capabilities across a variety of tasks, such as language, mathematics, coding, and etc (Bubeck et al., 2023). However, they still face some limitations, such as having frozen knowledge, being bad at some specialized tasks like calculation, and not being able to ground their generated solution outlines to the real world. Meanwhile, there are existing systems or models that can perform very well on domain-specific tasks. Therefore, a mechanism that links LLMs with the existing ecosystem of tools can bring the ability of LLM-based AI to another level.

To stretch the ability of LLMs, a sharply increasing number of works have studied LLM-based agents [1] which can manipulate API-based tools. A very ambitious vision is to build a new AI ecosystem that connects LLMs with millions of APIs, assessed via an API platform, for task completion. As shown in Fig. 1, the agent acts like a super-APP. It eases user interaction with language rather than GUI, manipulates massive API-based tools and thus supports a mass of functionalities. This requires the agent to have, on one hand, rich knowledge to deal with different user needs, and on the other hand developer's skills for manipulating APIs given documentation and understanding the results of API execution.

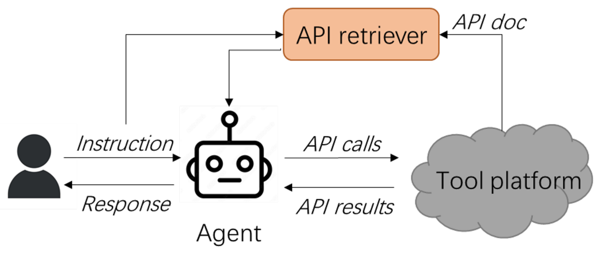

Figure 1: Agent connecting a user with massive APIs.

Though the research community has made an effort to build up generic LLM-based agents, we found that a widely accepted evaluation mechanism for LLM-based agents on tool use is still lacking. This prevents researchers from making fair comparisons between different agent systems, as well as gaining insights into the challenges of designing agents. In this work, we aim to narrow this gap by providing an evaluation mechanism, putting special emphasis on discovering the limitations of existing agents as well as the problems of current API design philosophy.

It is very challenging to evaluate an agent because of the complex process of an agent performing user tasks. Once a user issues an instruction, the agent first decomposes it into solvable subtasks with available tools, depending on the complexity of the task. Then, it may need to collect user needs by interacting with the user and call multiple APIs. Eventually, it responds to the user as per

[*]Corresponding authors.
[†]Work was done when he interned at OPPO.
[1]The term *agent* denotes *LLM-based agent* by default in this work.

the results of APIs. The problems this complex process brings to agent evaluation are: (1) the possible involvement of users makes the evaluation very hard. (2) the agent may fail in any step of the sequential process, making the samples in the later stage rare. (3) executing a complex process may involve multiple aspects of capability. However, an end-to-end performance cannot help locate the weaknesses of agents and gain more insights. To solve these problems, we dissect the whole process into intermediate decision behaviors to get a thorough view of the involved capabilities. According to this anatomy, we include 7 aspects of tool use capability in our evaluation schema. Each aspect corresponds to one separate evaluation subtask without involving human users.

Under the guide of our evaluation schema, we build a toolset and a dataset of instructions, following the principle of reflecting real-world challenges. We construct our toolset by addressing a series of concerns as demonstrated below rather than assembling some random tools. (1) To ensure reflection of real-world challenges, we collect tools from the real API platform. It matters to reflect the properties of API design and documentation. (2) We intentionally control the diversity of tools regarding functionalities and API structures. According to our observation, a tool may have a single API, a list of APIs, or several collections of APIs. These organizational structures can indicate the variety of functionality and expose different difficulties in calling. (3) We find some APIs depend on other APIs [2]. We include this kind of dependency relationship in our toolset. (4) To enable high-level tasks applied to the toolset, we take measures to increase the coherence of tools regarding application scenarios. (5) We noticed that the affordability of toolset can be one potential problem for individual researchers. To avoid this problem, we devoted lots of engineering work to make the toolset usable at low or even no cost.

On top of our toolset, we construct a set of user instructions that may use the contained tools to solve. We analyze how humans ask questions and summarize five types of instructions, varying in user intentions and complexities. These different types of instructions can be used to produce evaluation data required by our evaluation schema. Additionally, we emphasize that the instructions should be in the real way of user expression. Only by this can the evaluation data imply the mismatch between user expression and the form of information required by APIs.

To conclude, we contribute an evaluation mechanism of LLM-based agents on API manipulation. It is composed of several evaluation subtasks supported by one toolset and one dataset. We make these resources publicly available at `https://github.com/OPPO-Mente-Lab/agent_eval`.

2 Related Works

2.1 LLM-based Agents

LLM is the core component of an LLM-based agent. In the LLM domain, ChatGPT (Ouyang et al., 2022) is the most typical proprietary LLM and represents the SOTA LLM while many open-sourced LLMs like LLaMA (Touvron et al., 2023) are also very competitive. These LLMs have shown impressive language ability, rich knowledge, great potential in reasoning, and unbelievable generality in Question Answering tasks (Bubeck et al., 2023).

These characteristics of LLMs naturally inspire researchers to use LLMs as the brain of agents, which are designed to interact with complex environments and are closer to general AI.

In the surging literature of LLM-based agents, the agents have been explored to (1) manipulate external tools to solve more complex tasks (Nakano et al., 2021; Song et al., 2023; Shen et al., 2023); (2) play games (Zhu et al., 2023; Xu et al., 2023b); (3) form a multi-agent system which can do big projects collaboratively (Qian et al., 2023; Talebirad and Nadiri, 2023); (4) and even be embedded in robots to interact with the physical world (Wang et al., 2023; Ichter et al., 2022). This work focuses on the tool use ability of agents.

Some works have explored connecting LLMs with a pre-specified set of tools. By enabling LLMs to manipulate tools, the LLMs can access more information than that frozen in the weights (Nakano et al., 2021), overcome the shortcomings of LLMs like calculation (Schick et al., 2023), and complete more complex tasks than QA (Zhou et al., 2023; Shen et al., 2023). As proof of concept, these works demonstrate that equipping LLMs with tool use ability is a promising direction.

We distinguish between close-world settings and open-world settings. The close-world settings usually have a few special properties: (1) the number of tools is usually limited; (2) the design of APIs

[2]This is caused by the principle of API design – being simple and general.

tends to be simplified to ease the calling by LLMs. (3) considering the toolset will not be updated frequently, optimizing LLMs for the toolset is feasible, for example, by constructing toolset-specific training data to fine-tune the LLMs.

On the opposite, in the open-world settings (Liang et al., 2023; Patil et al., 2023), (1) the number of APIs API platform can be massive and may keep increasing; (2) the APIs are designed in an LLM-agnostic way and documented by following a unified schema required by the API platform. (3) the tools available in the API platform always keep changing. Our work is for evaluating agents designed for the second setting.

Different forms of tools have been considered in the literature, such as APIs (Patil et al., 2023; Qin et al., 2023), websites (Deng et al., 2023; Yao et al., 2022) and mobile APPs (Zhang et al., 2023; Hong et al., 2023; Rawles et al., 2023). We divide these tools into two categories: API-based tools and UI-based APPs (e.g. websites, desktop software and mobile APPs), as per the different challenges they pose to the LLMs. This work aims to serve the investigation of agents on manipulating API-based tools.

An agent system basically consists of an LLM and an inference pipeline. The LLM is injected with the ability of manipulating tools by fine-tuning (Tang et al., 2023; Qin et al., 2023; Patil et al., 2023) or in-context learning (Shen et al., 2023; Xu et al., 2023a). In addition, considering the complexity of tool use tasks, the inference process is usually enhanced with more sophisticated reasoning (Yao et al., 2023), searching method of solution path (Qin et al., 2023), and etc. To facilitate the creation of agent systems, a few open-sourced frameworks have been released (Li et al., 2023a; Qin et al., 2023).

2.2 Evaluation of LLM-based Agents

Many early works evaluated their agent systems with their own evaluation suites, making it hard to compare different agents. In addition, these works usually only report the end-to-end performance, which is inadequate for gaining insight into the tool use ability. To address these problems, more and more effort has been put into benchmarking the existing agents.

T-Eval (Chen et al., 2023) is close to our work. they designed an inference process composed of several steps and compared the overall and step-wise performance of several LLMs as the backbones. Our goal is to design a universal evaluation mechanism which is not coupled with the design of agents.

AgentBoard (Ma et al., 2024) created one evaluation toolkit to test the generality of agents across different types of environments, such as Embodied environments, game environments, and tool environments. It is not dedicated to in-depth evaluation of tool use ability.

3 Methodology

We make our evaluation schema first to make clear our targets. Then, we take measures to construct a toolset inheriting real-world challenges and supporting tasks with varying complexities. Extra engineering work makes its usage affordable. Following this, we design five types of instruction data and align them to the evaluation schema. Finally, we determine the metrics of each evaluation sub-task.

3.1 Evaluation Schema

We go through the process of an agent performing a task and point out the capabilities involved in different stages. As shown in Fig. 2, the process starts with a user sending an instruction. Depending on its complexity [3], the agent may need to make a solution outline via planning.

- **Planning**, i.e. decomposing a complex task into several simple subtasks, each of which is solvable with a single API. Considering the varying complexity of user instructions as well as the mismatch between user needs and the design of APIs, planning would be very commonly used by agents.

When dealing with a simple task, the agent first needs to decide whether to use external tools. If yes, it then retrieves a few candidates of potentially useful tools and selects one of them according to the documentation of tools. Otherwise, it replies to the user directly.

- **Deciding whether to use tools**. Failing to trigger tool use when needed makes tasks not solved, while misusing tools can hurt LLM's performance in ordinary QA tasks.

- **Selecting useful tools**. If tools are required, the agent should be able to figure out the useful ones.

[3]We measure the complexity of a certain task with the number of tools required to complete this task.

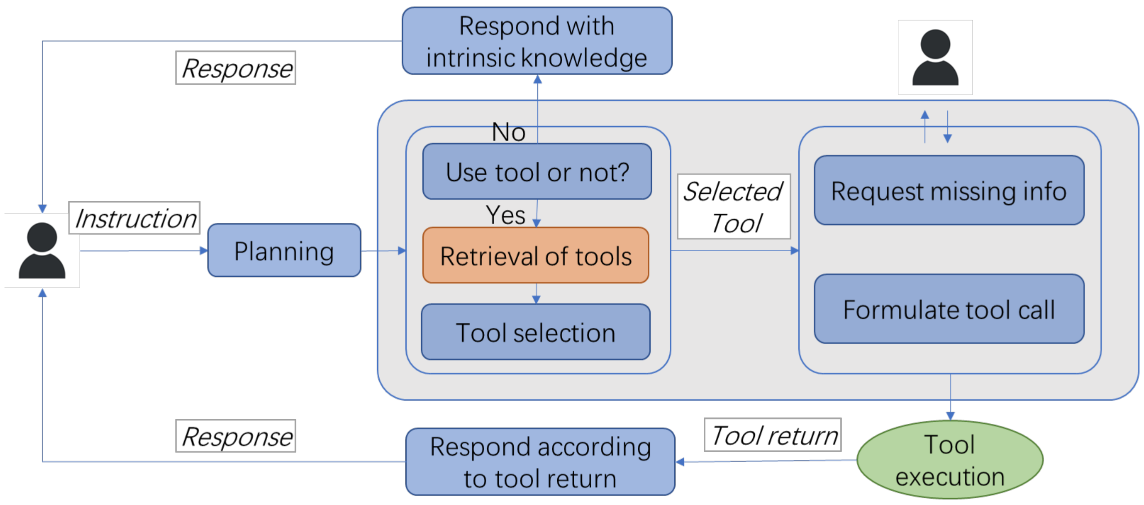

Figure 2: The process of an agent performing a task. Seven decision behaviors can potentially be involved. We examine the performance of an agent in each of them to gain a thorough understanding of its tool-use capability.

- **Responding with intrinsic knowledge**. There is a risk that fine-tuning an LLM with tool-use data makes it lose its original ability. Therefore, we also check whether LLMs still retain their intrinsic knowledge.

To call a certain tool, the agent needs to parse required parameter information from the context (i.e. conversation). If required information is not provided, the agent should ask the user to make clarifications.

- **Requesting missing parameter information**. It's common for users to initiate a dialogue with partial information. In this case, the agent should have the consciousness of requesting clarification rather than hallucinating.

- **Formulating tool calls**. When sufficient information is provided, the agent should be able to parse it and convert it to a valid format as per the specification of APIs. Here, one challenge to overcome is the mismatch between the user expression and the required format of parameters. Sometimes, commonsense reasoning is required.

Eventually, after receiving the execution results of tools, the agent synthesizes a final response to the user.

- **Responding according to the tool returns**. The variety of tools demands the agent to have great generality so that it can interpret the results of APIs, which are usually in JSON instead of natural language, and eventually generate a concise answer.

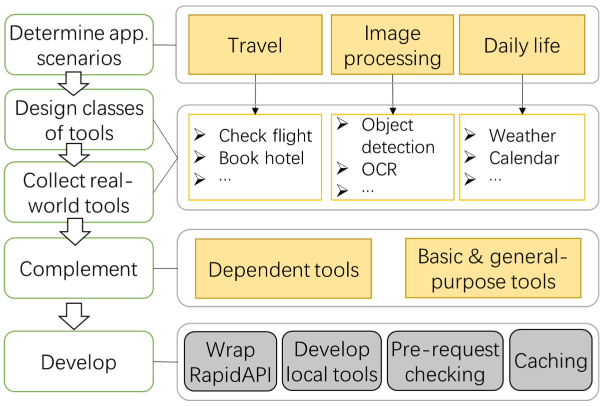

Figure 3: The process of building our toolset. The left side shows the steps while the right side illustrates their corresponding details.

In summary, our evaluation schema includes seven types of capabilities potentially involved in the process of tool use.

3.2 Toolset

Fig. 3 illustrates the process of developing our toolset, as detailed below.

Determining application scenarios. To achieve high **coherence** of tools, we start with selecting a few application scenarios (e.g. travel, image processing) of agents. Within each scenario, the tools have a relatively high chance of being combined to solve user's needs. In addition, having different scenarios helps ensure the **diversity** of tools.

Designing classes of tools. For each scenario, we think about the potentially useful tools. Then, instead of collecting the tools directly, we design the classes of tools to provide an umbrella, under which the tools from different sources and implemented by different people can be integrated. Here, each tool class is defined with a set of main functionalities.

Collecting real-world tools with documentation. For each tool class, we look for its real-world implementation from a well-known API platform RapidAPI [4], where massive APIs are deployed and documented with a unified schema. This is to make sure the design and documentation of APIs can reflect **real-world problems**.

In RapidAPI, most tools do not only have a single API (very typical in existing works) but multiple ones organized with a list or multiple collections. These multi-API tools may not only make the documentation of different APIs entangled but also comprise **dependencies** between APIs. For example, an API for checking flights takes airport codes as input, while checking these codes given airport names need to use another API (see Appendix A for a detailed example). This raises the difficulty level of manipulating APIs for agents. Therefore, we intentionally include tools with different API structures.

Complementing tools. The **dependencies** between APIs occur not only within a single tool but also across the boundary of tools. This underlying reason is that the philosophy of designing APIs is to make them simple and general utilities of many different APPs. However, for a toolset, a certain API's functionality will not be usable unless its dependent APIs are also included. To avoid this problem, we analyze the dependent APIs of already collected APIs and collect them in our toolset.
Additionally, we add a few **basic and general-purpose tools** (e.g. calculator, search engine, code interpreter).

Development. We first wrap remote tool services deployed on RapidAPI and develop a few local tools, forming an initial toolset. The tool services deployed on API platforms are usually not free. Even though we have tried to select the tool with the most free quota when collecting tools, a portion of them provide very limited free requests. Frequent access to the tool services can cause high subscription fees – an obstacle for research. To address these problems, we take the following measures: (1) developing **free alternatives** to some tools while reusing their documentation and API designs. (2) adding a **caching** mechanism to avoid repeated requests. (3) **check the validity** of API calls before sending them to the remote services.

[4] https://rapidapi.com/hub

3.3 Instruction Data

We design five types of instructions that can be used to evaluate all aspects of tool use capability in our evaluation schema.

Types of Instructions. Our first three types are low-level instructions, which can be solved mainly with the functionality of a certain API. We construct these instructions for each API in turn.

- Type-I: Instructions that do not need tools to solve but may mislead agents to call tools. For example, for the question "What's the weather **usually** like in London", one agent may call real-time weather API if they cannot understand the nuance caused by "usually". This type of instruction can be used to test two abilities: *deciding whether to use tool*, and *Responding with intrinsic knowledge*.

- Type-II: Instructions that need to use tools and provide sufficient information for formulating function calls. With this type of instruction, we can check whether an agent can parse or infer parameter information correctly from user questions. Also, because this type of instruction is relatively easy, an agent has a higher chance of getting a final assistant response (rather than being interrupted by invalid function calls). We thus can check whether an agent can make a proper response according to the return of a tool.

- Type-III: Instructions that need to use tools but provide insufficient information for filling parameters. For example, "Can you check the weather for me?". The agents would need to ask the user for his location. This type of instruction is very common and thus very important for evaluating agents. It can be used to check whether an agent can make multi-round interactions with the user consciously to solve the user's need.

From the Type-II and Type-III instructions, we filter out the data produced for APIs having dependencies. Then, they are used to form Type-IV instructions.

Table 1: Eight evaluation tasks and their used instruction data.

	Type-I	Type-II	Type-III	Type-IV	Type-V
Task-1: Deciding whether to use tools	✓	✓			
Task-2: Tool selection		✓			
Task-3: Requesting user to clarify missing info			✓		
Task-4: Filling parameter values		✓			
Task-5: Responding with intrinsic knowledge	✓				
Task-6: Responding according to tool returns		✓			
Task-7: Planning for resolving dependency				✓	
Task-8: Planning for high-level task					✓

Type-IV: instructions that are not complex but still need to use multiple APIs to solve because of the dependencies of APIs.

Lastly, we create high-level instructions issuing complex tasks:

Type-V: instructions that require to be decomposed into solvable sub-tasks by APIs. For example, "Plan a seven-day trip in Dubai for me". To complete this task, an agent would, for example, check the weather and search for interesting spots to visit.

Generating Instructions. We generate initial instruction data by prompting GPT4. Apart from the special requirements for each instruction type, we include the following general rules: (1) the instructions should be asked in the real way of human speaking. (2) human users do not mention API in their questions.
For each type of instruction, we use one generator to generate instructions first and then use one discriminator to filter out invalid ones. Additionally, human annotators double-check the instructions to ensure high quality and avoid ethical issues [5].

Aligning instructions to evaluation schema. In Table 1, we enumerate the evaluation tasks and the corresponding types of instructions to use. In tasks-1,3,4,5, the agent is given an instruction and its corresponding API specification. In task-2, the agent is given an instruction, a correct API along with a few perturbing APIs. In task-6, the agent is given a conversation history including a user message, an assistant message containing a function call and a tool result. In task-7 and 8, the agent is given an instruction and multiple APIs, expecting tool use response and chat response respectively.

[5]Two of our authors participate in data annotation.

See Appendix C for more details on the construction of our dataset.

3.4 Assessment & Metrics

We assess the performance of an agent for each data instance as below:

- Task-1: whether correct decision has been made for the two types of instruction. Overall precision, recall and macro-F1 score can be computed.
- Task-2: whether the right tool is chosen from the given candidates.
- Task-3: whether the response is to request clarification of missing information.
- Task-4: percentage of correct function call.
- Task-5: whether the response is related to the question. Answer quality is our concern.
- Task-6: whether the response is based on the tool results and whether desired information in the results is interpreted precisely.
- Task-7: the chain of function calls is compared with a ground-truth order of actions. The rate of progress is used as metrics.
- Task-8: whether the solution outline is sound – the coverage of provided APIs and whether the functionality of each API is correctly understood.

The assessing scripts, implemented by mixing rules and GPT-4 usage, are included in our evaluation mechanism too.

4 Experiment

In this section, we first demonstrate more details of our dataset and toolset. Then, we apply our evaluation mechanism to examine several well-known LLMs equipped with generic tool use ability, including ChatGPT series – GPT-3.5-turbo and GPT-4-8k (abbreviated as GPT-3.5 and GPT-4 below), and Qwen1.5 series with sizes 7b, 14b and 72b (abbreviated as Qwen-7b, Qwen-14b and Qwen-72b) (Bai et al., 2023). The new findings can show the value of our evaluation mechanism.

4.1 Dataset & Toolset

Dataset. The numbers of different types of instructions are shown in Tab. 2, while the size of data for each evaluation task is shown in Tab. 3.

Table 2: Number of each type of instruction.

Type	I	II	III	IV	V
Num	372	326	195	85	50

Table 3: Data number of each evaluation task.

Task	1	2	3	4	5	6	7	8
Num	698	326	195	326	372	311	85	50

Toolset. Our toolset consists of 66 APIs organized into 27 tools. 83% of these APIs are implemented based on API services from RapidAPI, while the other 17% are developed from scratch. We recognize 28 pairs of (API, dependent APIs). In addition, we combine 9 groups of coherent APIs for supporting high-level tasks. See Appendix B for concrete tool classes, and functionalities.

4.2 Evaluation of Generic LLM-based Agents

In Fig. 4, we compare the performance of agents in 8 evaluation tasks.

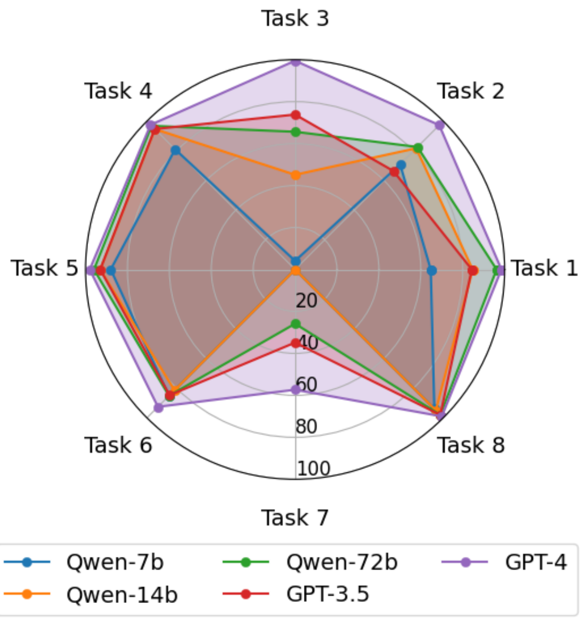

Figure 4: Comparing the performance of five generic agents in eight evaluation tasks. Metrics values can be found in Tab. 5.

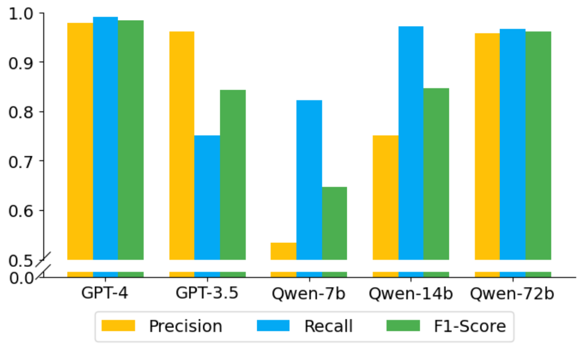

Figure 5: Performance of agents in tool-use decision: precision, recall and macro-F1.

Task-1: On the decision of tool utilization. (1) We found Qwen-7b and Qwen-14b have the problem of misusing tools – tending to use tools once given. This leads to relatively low precision in their tool-use decision. (2) On the contrary, GPT-3.5 is conservative in tool use – tending not to use tools even needed – resulting in a low recall. (3) GPT-4 and Qwen-72b can make proper decisions, above 0.96 in macro-F1 scores.

Task-2: On tool selection. To check whether the agents can figure out the right API to use, we provide the agents with one correct API along with four perturbing ones [6].

GPT-3.5 and Qwen-7b perform the worst in tool selection, however, for different reasons. Among the mistakes made by GPT-3.5, 77% is because of its conservativeness again – did not call any API, while only 23% are incorrect selections. Qwen-7b always selects the wrong tools, showing its weakness in understanding API specifications.

Compared with GPT-4, the Qwen-14b and Qwen-72b achieved accuracies less than 83%, having a big gap from GPT-4.

Task-3: On the awareness of requesting clarification. It is very often that one user initiates a dialogue with partial information. This requires the agent to figure out the missing parameter information for calling a certain tool and ask the user to clarify. However, we found that the three open-sourced LLMs – Qwen series – are bad at this, worse than both GPT-3.5 and GPT-4. GPT-4

[6]We select the perturbing APIs which have top-4 highest similarities with the target API regarding sentence embeddings encoded with `gte-base-en-v1.5` (Li et al., 2023b).

performs almost perfectly while GPT-3.5 still has big space for improvement.

In this case, some typical mistakes include: (1) hallucinating parameter values (no evidence can be found from the user's questions). (2) using a placeholder-like value (e.g. /path/to/image) instead of a real value. (3) imprecise parameter values (e.g. a location parameter requires a city name but is given a country name) are used, leading to exceptions in executing APIs. (4) required parameters are missing in the function calls.

Task-4: On the correctness of function calls. When sufficient information is contained by the user's questions, most LLMs including Qwen-14b have over 94% correct function calls, except Qwen-7b achieving around 80%. These numbers are pretty decent. We reckon the reason is most existing works focus on this setting while neglecting the others.

Task-5: On the utilization of intrinsic knowledge. We empirically notice that, in some existing works (e.g. Qin et al., 2023), fine-tuning LLMs with tool use data makes the LLMs forget their original capabilities in ordinary Question Answering (QA) tasks. Fortunately, this did not happen in the generic agents we evaluated. Intuitively, it is not hard to achieve since QA is a more basic task for generic LLMs. Even though, we still consider keeping this aspect in our evaluation schema to remind the phenomenon of over-fitting.

Task-6: On interpreting tool results. Overall, these agents are good at interpreting tool results. However, we still noticed a few typical errors by them. In some cases, the LLMs fail to locate the desired information, to some extent because of poor readability of results. In addition, we find LLMs have shortcomings in a kind of copy&paste capability of target information. This makes some information that is sensitive to character-level precision (e.g. URLs, longitude and latitude, long decimal values, etc.) not useful anymore. Furthermore, we find some APIs, e.g. searching APIs, return very long results exceeding the max context length of LLMs.

Task-7&8: On planning capability. We examine the planning ability of agents with two folds of experiments. In the first fold of experiments, we examine whether an agent can complete a low-level task by manipulating APIs with dependencies. All the evaluated agents have poor performance – even GPT-4 has a success rate lower than 60%. We find the Qwen models, even Qwen-72b, rarely have the sense of starting with more basic APIs. It is very challenging for the agents to manipulate APIs with dependencies.

In the second fold of experiments, we check whether an agent can outline a solid plan involving tool use for a high-level task. We find these LLMs' performance in decomposing a high-level task is always decent. Though both settings require the planning capability of LLMs, they impose very different challenges. For LLM-based agents, talking is much easier than doing.

Next, we discuss a bit more from other angles.

On scaling law. Though not a new finding anymore, the scaling law still applies in API manipulation scenarios. The performance of Qwen series reveals larger LLMs have better performance regarding almost every aspect of tool use capability.

On API design and quality of documentation. The effect of API design and documentation quality deserves more attention. A few concrete examples are: (1) A tool with multiple APIs may introduce its functionalities in its tool-level description while giving very unclear API-level descriptions. (2) The execution results of APIs have poor readability or are too verbose. (3) The APIs with dependencies seem too complex for the LLMs to use.
Despite getting some insights, we believe more research on the API side needs to be done. One question already inspired by our observations is: in the era of LLMs, should we design new standards for API design and documentation? It is a complex problem and deserves dedicated research. We treat it as future work.

5 Conclusion

The LLM community is driving towards an ambitious vision: building a new AI ecosystem in which LLM-based agents serve users by manipulating millions of APIs. However, an evaluation mechanism for such agents is still missing, preventing studies from proceeding in this area. In this work, we narrow this gap by proposing a new evaluation mechanism for generic LLM-based agents. We designed a thorough evaluation mechanism schema aiming to examine seven different aspects of tool use capability. Also, we release one dataset and one toolset, both designed to reflect real-world challenges. These resources can support the studies on improving LLM-based agents as well as a new philosophy of API design. We evaluated five influential LLMs and shared findings and insights into their tool use capability. The found weaknesses of LLMs can indicate the future directions to go.

6 Limitations

When designing evaluation tasks, we did not include data involving multi-turn interactions with users. Can the agents still formulate correct function calls by parsing information from multi-turn dialogue? What the performance will be like if the agents need to continuously request clarification from the users? These problems cannot be answered by our evaluation mechanism. Our evaluation mechanism does not provide an end-to-end performance or an overall performance score.

In addition, in the landscape of generic agent research, API retriever is critical. We assume the existence of a good third-party API retriever. More studies dedicated to API retrievers are suggested to be done so that we can be closer to estimating the overall performance of LLM-based agent systems.

7 Disclaimer

The toolset released by us is only for research purposes. It is not for the usage of solving real-life problems (e.g. checking flight prices).

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. 2023. T-eval: Evaluating the tool utilization capability step by step. *CoRR*, abs/2312.14033.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *CoRR*, abs/2306.06070.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. 2023. Cogagent: A visual language model for GUI agents. *CoRR*, abs/2312.08914.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. 2022. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning, CoRL 2022, 14-18 December 2022, Auckland, New Zealand*, volume 205 of *Proceedings of Machine Learning Research*, pages 287–318. PMLR.

Chenliang Li, He Chen, Ming Yan, Weizhou Shen, Haiyang Xu, Zhikai Wu, Zhicheng Zhang, Wenmeng Zhou, Yingda Chen, Chen Cheng, Hongzhu Shi, Ji Zhang, Fei Huang, and Jingren Zhou. 2023a. Modelscope-agent: Building your customizable agent system with open-source large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, pages 566–578. Association for Computational Linguistics.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023b. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. 2023. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis. *CoRR*, abs/2303.16434.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn LLM agents. *CoRR*, abs/2401.13178.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. Webgpt: Browser-assisted question-answering with human feedback. *CoRR*, abs/2112.09332.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

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

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *CoRR*, abs/2307.07924.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *CoRR*, abs/2307.16789.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy P. Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. *CoRR*, abs/2307.10088.

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

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving AI tasks with chatgpt and its friends in huggingface. *CoRR*, abs/2303.17580.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world applications via restful apis. *CoRR*, abs/2306.06624.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent LLM agents. *CoRR*, abs/2306.03314.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *CoRR*, abs/2306.05301.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *CoRR*, abs/2305.16291.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023a. On the tool manipulation capability of open-source large language models. *CoRR*, abs/2305.16504.

Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. 2023b. Exploring large language models for communication games: An empirical study on werewolf. *CoRR*, abs/2309.04658.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. Appagent: Multimodal agents as smartphone users. *CoRR*, abs/2312.13771.

Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. 2023. LLM as DBA. *CoRR*, abs/2308.05481.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. 2023. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *CoRR*, abs/2305.17144.

A An Example Illustrating Dependency between APIs

Here, we show the specifications of three APIs from RapidAPI - skyscanner80[7] in Listing 1,2 and 3. Only necessary information for showing the dependencies between APIs is kept in these doc examples. To use API `flights_search_one_way`, we need to first check API `flights_auto_complete` for the IDs of the origin and destination. Afterwards, because `flights_search_one_way` may not be able to return all the results at one time. More requests to API `flights_search_incomplete` need to be done to finalize fetching all flight data.

B Details of Toolset

The design of application scenarios, tool classes and their functionalities can be found in Tab. 4.

C Details of Dataset Construction

C.1 Instruction data

In Fig. 6, we show the process of producing the five types of instruction data. Taking the specification of each API as input, the instruction data of type-I, II, and III are initially generated with prompted GPT4 respectively, and then manually checked and filtered. The type-I instructions belonging to APIs without dependencies form the final type-I instruction data.

More complexly, the type-II and III instructions are processed to, for example, replace placeholders of files and URLs with real values. The folds belonging to APIs without dependencies form the final type-II and type-III instruction data, while the other folds form the type-IV instructions after manual selection.

To produce type-V instructions, we manually combine some APIs that are coherent regarding potential application scenarios. Then, as done for the first three types of instructions, we prompt GPT4 to generate initial instructions and follow them with manual filtering to form the final type-V instructions.

C.2 Evaluation tasks and scoring methods

In Fig. 7, we show how the eight evaluation tasks are synthesized from the five types of instruction data. Some key points deserving sharing are: (1) Most tasks except task-5 couple each instruction with its corresponding API specification. (2) Task-2 also has a few confusing tools according to their semantic similarities of specifications with the correct tool. (3) Each data instance for Task 6 is a conversation history, consisting of an initial instruction, a function call produced with GPT4 and double-checked by humans, and the real execution result. (4) Task-7 and 8 have multiple tools for each instruction. The dependency relationships and combinations are both manually crafted by analyzing the API designs and functionalities of tools in our toolset.

It is easy to derive the ground-truth labels for task-1,2,3 and check the predictions of agents with logical programs. Task-4,5,6,8 have flexible expected outputs. Thus, we prompt GPT4 to replace human annotators for the automation of evaluation. Task-7 is the most complex to examine. We prompt GPT4 to check the responses of agents first. For each failed answer, instead of simply assigning a zero score, we measure its progress in solving the task, by comparing the sequence of called tools with a reference sequence of tools (manually labeled).

D Experimental settings

We access GPT-4-8K and GPT-3.5-turbo via API and access Qwen1.5 series LLMs via local running.

We only ran the experiments of evaluating generic agents once. The metrics values are averaged within the evaluation data for each task.

E Performance of Agents

Tab. 5 contains the performance of two GPT versions and three Qwen1.5 versions in 8 evaluation tasks.

[7] https://rapidapi.com/datastore/api/skyscanner80

Listing 1: Documentation of `flights_auto_complete` API

```
{
  "name": "flights_auto_complete",
  "description": "This endpoint is resposible for providing a list of
      airports for the location",
  "parameters": {
    "query": {
      "type": "STRING",
      "description": "Name of the location where the Airport is
         situated. Ex: New York"
    }
  }
}
```

Listing 2: Documentation of `flights_search_one_way` API

```
{
  "name": "flights_search_one_way",
  "description": "This API helps to get the list of one-way flights.
     Note:- In the event that the status is incomplete (data->context
     ->status=incomplete), you must utilize the api/v1/flights/search
     -incomplete endpoint to retrieve the complete data until it's
     complete (data->context->status=complete).",
  "parameters": {
    "fromId": {
      "type": "STRING",
      "description": "`fromId` can be retrieved from `
         flights_auto_complete` (data->id) Ex:
         eyJzIjoiTllDQSIsImUiOiIyNzUzNzU0MiIsImgiOiIyNzUzNzU0MiJ9 (
         New York)"
    },
    "toId": {
      "type": "STRING",
      "description": "`toId` can be retrieved from `
         flights_auto_complete` (data->id) Ex: eyJzIjoiTE9
         ORCIsImUiOiIyNzU0NDAwOCIsImgiOiIyNzU0NDAwOCJ9 (London)"
    },
    "departDate": {
      "type": "Date",
      "description": "Format: YYYY-MM-DD. Ex: 2024-06-01"
    }
  }
}
```

Listing 3: Documentation of `flights_search_incomplete` API

```
{
  "name": "flights_search_incomplete",
  "description": "Obtain complete data for the endpoint of
     flights_search_one_way, flights_search_roundtrip. Until the item
     's status is complete (data->context->status=complete), you must
      call the API multiple times",
  "parameters": {
    "sessionId": {
      "type": "STRING",
      "description": "sessionId can be retrieved from
         flights_search_one_way or flights_search_roundtrip (data->
         context->sessionId)"
    }
  }
}
```

Table 4: Design of scenarios, tool classes, and API functionalities.

Scenario	Tool Class	Functionalities
daily life	weather	realtime weather, weather forecast, astronomy info
	news	news search, headlines
	calendar	public holidays, check month calendar
	recipe	search recipe
image processing	object detection	recognize objects in image
	ocr	extract text in image
	image translation	translate text in image
	image file processing	compression, format conversion, resize
	removing background	remove background
	web capture	take image screenshot
travel	flight	search one-way flights, search round-way flights, check flight details and prices
	accommodation	search hotels, check hotel details, prices, and reviews
	tourist attraction	search attractions, check details, photos and reviews of attractions
	currency	exchange rate
	airport	check airport info
	check codes	language codes, country codes,
	geocoding	convert between address and coordinates
basic & general-purpose	search	web search, image search, video search, news search
	python interpreter	python interpreter
	calculator	math calculation
	translation	translation
	ip lookup	check ip address
	access user info	user profile, location
	agent equipments	get current time

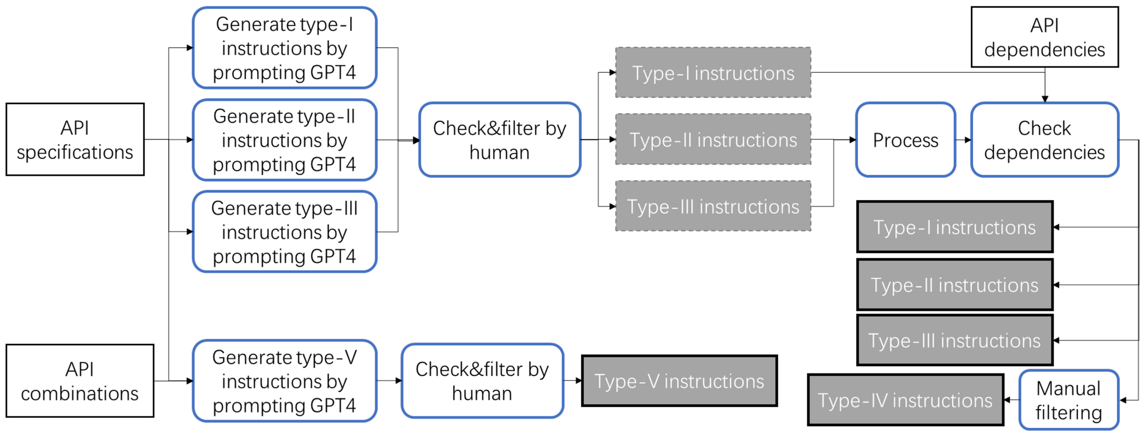

Figure 6: The process of producing instruction data.

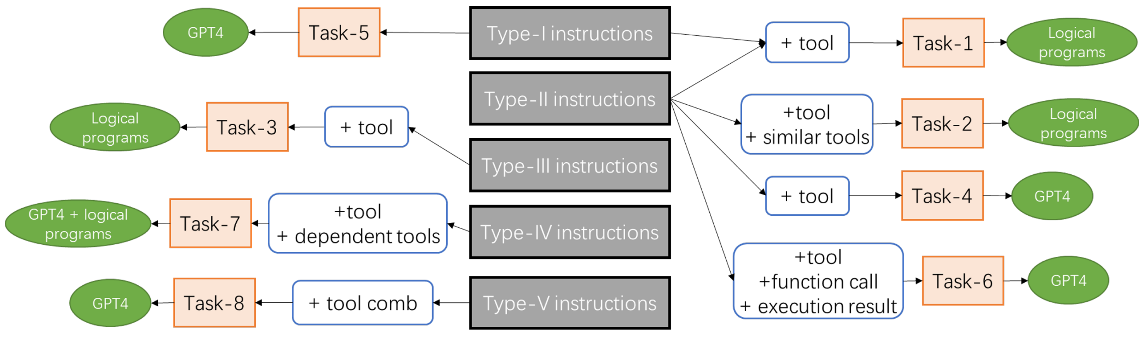

Figure 7: How the evaluation tasks are synthesized and their corresponding scoring methods.

Table 5: Performance of generic agents on eight evaluation tasks.

Tasks \Agents		GPT-4	GPT-3.5	Qwen-7b	Qwen-14b	Qwen-72b
Task 1	Precision	0.98	0.96	0.53	0.75	0.96
	Recall	0.99	0.75	0.82	0.97	0.97
	F1-score	0.98	0.84	0.65	0.85	0.96
Task 2	Accuracy	0.97	0.66	0.71	0.82	0.83
Task 3	Percentage	0.99	0.74	0.04	0.45	0.66
Task 4	Precision	0.98	0.95	0.81	0.95	0.97
Task 5	Relatedness	0.98	0.93	0.88	0.93	0.96
Task 6	Passing rate	0.93	0.85	0.85	0.81	0.85
Task 7	Progress	0.57	0.35	0.00	0.00	0.26
Task 8	Passing rate	0.99	0.98	0.94	0.95	0.98