MetaBench: Planning of Multiple APIs from Various APPs for Complex User Instruction

Anonymous ACL submission

Abstract

Large Language Models (LLMs) can interact with the real world by connecting with versatile external APIs, resulting in better problem-solving and task automation capabilities. Previous research primarily either focuses on APIs with limited arguments from a single source or overlooks the complex dependency relationship between different APIs. However, it is essential to utilize multiple APIs collaboratively from various sources, especially for complex user instructions. In this paper, we introduce `MetaBench`, the first benchmark to evaluate LLMs' ability to plan and execute multiple APIs from various sources in order to complete the user's task. Specifically, we consider two significant challenges in multiple APIs: *1) graph structures:* some APIs can be executed independently while others need to be executed one by one, resulting in graph-like execution order; and *2) permission constraints:* which source is authorized to execute the API call. We have experimental results on 9 distinct LLMs; e.g., GPT-4o achieves only a 2.0% success rate at the most complex instruction, revealing that the existing state-of-the-art LLMs still cannot perform well in this situation even with the help of in-context learning and fine-tuning. Our code and data will be publicly available at https://github.com/.

1 Introduction

Empowering Large Language Models (LLMs) (Zhao et al., 2023) with versatile tools such as retrievers (Wang et al., 2023), models (Shen et al., 2023), and even physical robots (Liang et al., 2023), holds significant promise in overcoming inherent limitations, such as hallucination (Ji et al., 2023) and outdated information (Nakano et al., 2021; Liu et al., 2023a), and unveils the immense potential for LLMs to tackle increasingly complex and interactive real-world tasks (Li et al., 2023; Lu et al., 2023). Over the past several months, lots of new benchmarks and datasets have been proposed

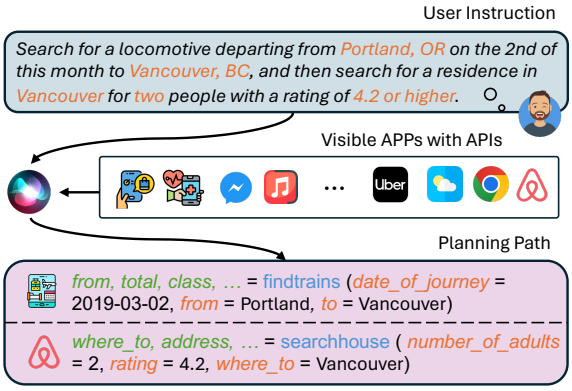

Figure 1: An example of one user instruction requires two independent APIs from different APPs since input arguments of both two APIs do not rely on each other. We use different icons to indicate different APPs, and color API, and returned arguments and input arguments.

to evaluate the performance of different LLMs to adeptly select and execute various tools (Li et al., 2023; Shen et al., 2023; Huang et al., 2024a), marking a pivotal milestone in their evolution. Out of plentiful tools in practice, APIs have become one of the fundamental and promising tools in today's digital world, due to greater flexibility and customizability with well-defined format and ease of execution (Qin et al., 2023).

Previous works have attempted to evaluate LLMs on their ability to call the correct API in multiple turn dialogues, such as API-Bank (Li et al., 2023) and ToolBench (Qin et al., 2024), or single turn instructions, like APIBench (Patil et al., 2023). However, most existing benchmarks focus either on a single API call in a single turn or on APIs with limited arguments. For instance, ToolBench and API-Bank evaluate one API call per turn in multi-turn dialogues, while APIBench considers only APIs with one or two arguments. Furthermore, the small number of arguments makes it difficult to fully explore the complex dependency relationships between multiple APIs. For instance, the input arguments for a current API may depend on the

return arguments of several previous APIs. These limitations highlight a gap in addressing complex user instructions when it is necessary to utilize multiple APIs in practice, underscoring the need for more comprehensive and practical evaluation benchmarks.

To bridge the gap, we introduce a new evaluation benchmark: `MetaBench`, representing the first effort to assess the aptitude of LLMs to function as the meta planner for multiple APIs from various sources for complex user instruction. Specifically, we simulate a situation in which the user instruction can be fulfilled through collaboratively API calls from various APPs in the mobile device. Figure 1 shows one typical example. Given the complex user instruction, the meta LLM, such as Apple's Siri and Google Assistant, need to plan an executable path according to user instruction and corresponding API descriptions. To fulfill this requirement, it is necessary not only to indicate which APP will distribute and execute each API but also to specify the execution order of the APIs, including all necessary inputs and returned arguments.

Therefore, it offers two significant challenges: *graph structure* and *permission isolation*. Firstly, the inter-dependency between multiple APIs creates a more complex execution structure. Some APIs can be executed independently, while others are dependent and must be executed sequentially, resulting in a graph-like structure. Secondly, these APIs may originate from different sources, and the LLM might not have permission to call them directly. This necessitates identifying the authorized source for each API. For instance, APIs from one company may only be executed by an LLM within the same company. In doing so, we aim to chart a path towards realizing the vision of an intelligent assistant capable of seamlessly navigating and interfacing with the myriad APPs and APIs pervasive in contemporary digital ecosystems. To conclude, our contribution can be summarized in three folds:

- To the best of our knowledge, we are the first to identify graph structure and permission isolation issues of multiple API calls when addressing complex user instructions.
- We propose `MetaBench`, serving as an important complementary evaluation benchmark to assess the planning capabilities of different LLMs as meta planner for these APIs. Additionally, we introduce an automatic data collection pipeline, which can be used to gather data efficiently and effectively.
- Our experimental results on 9 distinct LLMs demonstrate almost all models, including the latest GPT-4o, fall short in this setting, particularly when dealing with complex graph planning structures. Further analysis shows that simple in-context learning and fine-tuning do not significantly improve performance.

2 Related Work

Tool Benchmarks. The complexity of real-world tasks necessitates the integration of diverse tools and services, consisting of three types of tools (Qin et al., 2023): 1) physical interaction-based tools (Liang et al., 2023); 2) GUI-based tools (Wang et al., 2024); and 3) program-based tools (Wang et al., 2023; Li et al., 2023). On the one hand, some work focuses on models, retrievers, or calculators to address the intrinsic limitations of LLMs, such as ToolQA (Zhuang et al., 2023) and ToolBench (Qin et al., 2024). On the other hand, another line of work targets APIs since they are particularly crucial for bridging smooth interaction between humans and the digital realm (Li et al., 2023; Qin et al., 2024; Huang et al., 2024a). Most previous works formulate this as an API selection task given all related information about each API and current input, which overlooks the nuanced dependencies and permission constraints between different APIs, such as APIBench (Patil et al., 2023) and API-Bank (Li et al., 2023). Nevertheless, the successful execution of APIs in the real world necessitates meeting requirements fulfilled (either the value is provided by the user or previous APIs) and obtaining permission from trusted agents beyond just knowing API names and a few arguments.

Language Agent. Existing frameworks for language agents have made notable strides in facilitating interaction with external tools (Shen et al., 2023; Li et al., 2023; Huang et al., 2024a) and environment (Puig et al., 2018a; Wang et al., 2022). They usually follow the single-agent paradigm to access different tools or services sequentially (Lu et al., 2023; Li et al., 2023). For example, Lu et al. (2023) propose Chameleon which utilizes one agent to plan the execution order of different services by outputs a sequence of names of tools, which assume that the agents to call these tools are

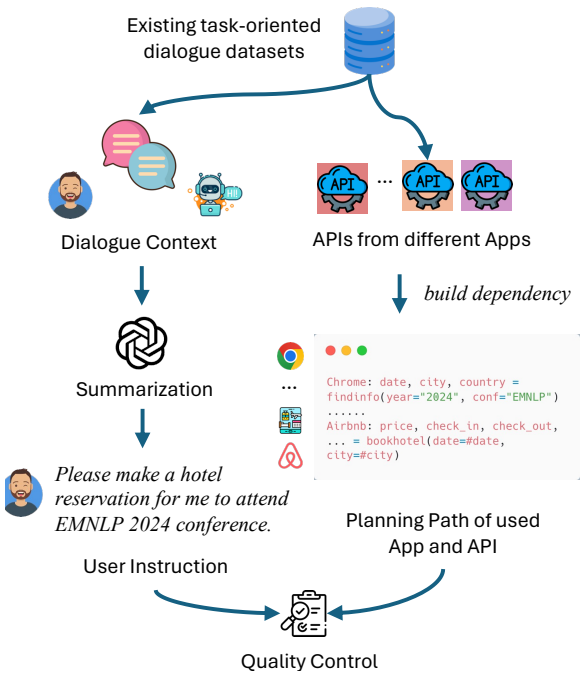

Figure 2: A high-level processing to collect the MetaBench, taking advantages of existing task-oriented dialogue datasets.

already known, and lots of works follow (Xu et al., 2023; Huang et al., 2024a). Furthermore, various benchmarks are proposed to evaluate the abilities of LLMs serving as agents in different situations (Li et al., 2023; Liu et al., 2023b; Ma et al., 2024). For instance, (Yao et al., 2023) proposes WebShop to evaluate whether LLMs are capable of interacting with the Web. Similarly, (Puig et al., 2018b) simulates household activities through programs, and many works use this as a testbed for embodied agents (Hao et al., 2024).

3 MetaBench Construction

In this section, we start with a formal task definition and then provide a detailed explanation of how we efficiently and effectively built our MetaBench by leveraging existing datasets.

3.1 Task Definition

Given the user instruction u and a virtual mobile environment with an APP family, $\mathcal{E} = \{APP_1, APP_2, ..., APP_n\}$ where each APP contains several APIs $\{p_i^1, ..p_i^j\}$ where i stands for i_{th} APP and j means j_{th} API inside this APP, the meta agent need to decide an executable path to call different APIs from various APPs to fulfill the instruction in the format of the list which each item in the list is $\{APP_i : r_1, r_2, .., r_m = p_i^j(k_1 = v_1, ..., k_n = v_n)\}$. The APP_i and p_i^j denote the name of the APP and corresponding API of this APP, and the r_i and k_i mean the i_{th} returned and input arguments respectively. The v_i can be the actual value provided by the user or a returned argument by previous APIs.

Benchmark	SS	SM	MS	MM	DP
APIBench (Patil et al., 2023)	✓	✗	✗	✗	✗
API-Bank (Li et al., 2023)	✓	✗	✗	✗	✗
ToolQA (Zhuang et al., 2023)	✓	✗	✗	✗	✗
ToolBench (Qin et al., 2024)	✓	✓	✓	✓	✗
UltraTool (Huang et al., 2024a)	✓	✓	✗	✗	✓
MetaBench (Ours)	✓	✓	✓	✓	✓

Table 1: Comparison with existing evaluation benchmarks at the turn-level for a fair comparison. DP stands for Dependency.

3.2 Data Categorization

Based on the number of APPs and APIs utilized in each user instruction, the data can be categorized into four distinct types. Each category represents a typical use case in practical scenarios while most of existing benchmark only focus on part of these typical situations or overlook the complex dependency relationships (Table 1).

- ***Single APP Single API (SS)*** The instructions of the users only need to utilize one API from one APP.
- ***Single APP Multiple API (SM)*** The instructions of the users need to utilize multiple API from one APP. It is important to note that these APIs can be called either sequentially or concurrently, depending on whether there is a dependency between their arguments.
- ***Multiple APPs Single API (MS)*** The instructions of the users need to utilize multiple APIs and each of them belongs to one different APP. Also, there may exist dependency between APIs across different APPs.
- ***Multiple APPs Multiple API (MM)*** The instructions of the users need to utilize multiple APIs and multiple APPs. The difference with MS is there may exist multiple APIs come from the same APP. Furthermore, the dependency relationship between arguments can be the most complex when dealing with APIs from the same APP or from different APPs.

3.3 Data Collection

To maximize the authenticity of user instructions and minimize human efforts, we prioritize using

Data Type	Example	Structure
SS	Instruction: *Find a house with a rating of 4.6 or higher for a trip to Delhi for two people, inquire about laundry service availability* Output: House: address, phone_number, total_price, has_laundry_service, … = searchhouse(number_of_adults='2', rating='4.60', where_to='Delhi')	
SM	Instruction: *Please book a Hatchback car with insurance to be picked up from Warsaw Chopin Airport on March 7th at 1:30 pm, and returned on March 13th in Warsaw.* Output: Rents: pickup_location, price_per_day, ... = getcarsavailable(car_type='Hatchback', city='Warsaw', end_date='2019-03-13', pickup_time='13:30', start_date='2019-03-07') Rents: car_type, car_name, ... = reservecar(add_insurance='True', car_type=car_type, end_date=end_date, pickup_location=#**pickup_location**, pickup_time=pickup_time, start_date=start_date)	
MS	Instruction: *Search for a locomotive departing from Portland, OR on the 2nd of this month to Vancouver, BC, and then search for a residence in Vancouver for two people with a rating of 4.2 or higher.* Output: Train: from, total, class, … = findtrains (date_of_journey = 2019-03-02, from = Portland, to = Vancouver) House: address, phone_number, total_price, has_laundry_service, … = searchhouse(number_of_adults='2', rating='4.2', where_to= Vancouver')	
MM	Instruction: *Please make a reservation for 3 people at one Korean restaurant in San Francisco at 1:30 pm on March 12th, and also book a Luxury taxi for 3 to 4 Embarcadero Center.* Output: Restaurant: restaurant_name, has_vegetarian_options, phone_number, rating, address, price_range, category, ... = findrestaurants (category ='Korean', has_seating_outdoors='True', location='San Francisco') Restaurant: date, time, location, ... = reserverestaurant (date='2019-03-12', location=location, number_of_seats='3', restaurant_name = #**restaurant_name**, time='13:30') Rents: destination, ride_type, ride_fare, wait_time, number_of_seats = getride(destination='4 Embarcadero Center', number_of_seats='3', ride_type='Luxury')	

Figure 3: An example of different types of samples in MetaBench. We color APP, API, and returned arguments and input arguments. We also present the structure of the example using grey nodes and colorful nodes to indicate user instruction and APIs from different APPs, respectively. We bold the **argument** which is returned by the previous API call (a.k.a., dependency relationship). Para. and Seq. represents the parallel and sequential size of the corresponding data sample. We emphasize we only choose the simplest examples in each type for better understanding, there are data samples with much more complex logic structures in the original dataset.

existing task-oriented dialogue datasets (Rastogi et al., 2020; Budzianowski et al., 2018). These datasets are typically collected through human-to-human interactions in real-world scenarios and contain a wide range of APIs across numerous domains and services. Specifically, we selected the SGD (Rastogi et al., 2020) dataset as the seed dataset because it encompasses most of the domains and APIs. We then utilized LLMs and Python scripts to generate the desired inputs and outputs, respectively. Figure 2 illustrates the detailed procedures.

Instruction Acquisition. Firstly, we extract the utterances of the user and system in the task-oriented dialogue and feed it into the LLM[1] to summarize the user's requirements in one instruction. For example, the user may want to know the city and date of EMNLP 2024, and book a hotel according to the city and date. In the previous task-oriented dialogue, this is achieved by multi-turn interactions. In contrast, we summarize the whole dialogue into one complex user instruction to mimic more natural and complex cases in practice. To ensure that the values of certain intermediate arguments (such as *date* and *city*) are not disclosed at the instruction, we require the LLM to avoid outputting the actual values of other arguments, except for those that are explicitly provided in the prompts, such as user-aware arguments. The prompt details can be found in Appendix A [2].

Planning Path. Besides the instruction part, we write a Python script to automatically parse the API calls at different system turns in the multi-turn dialogue to form the planning path as the output. Specifically, we regard different domains (a.k.a., services) in task-oriented dialogue as different APPs such as restaurants and hotels, and extract the name of the domain and API first to locate which APP should invoke to call the API, and then we follow the execution order of different APIs to build the dependency between various arguments. For example, if the returned arguments from the previous API are required in the current API, we use #name to indicate it such as #date and #city in the Figure 2. In this way, we can get an executable and unique path to execute APIs from different APPs.

Quality Assessment To ensure the quality of data, we utilize a Python script to validate whether or not all actual values are provided from the user side, and none of values are provided from the system side. Furthermore, we adapt GPT-4o to score each instruction in terms of fluency and diversity from 1 to 10, and then remove cases whose score is lower than 6. Approximately 20% of the samples were removed, and the average score of the remain-

[1] GPT-4o during the collection

[2] All prompts can be found in Appendix if not stated.

Statistics	SS	SM	MS	MM
# Samples	200	200	200	200
# Apps	9	11	10	11
# APIs	11	22	12	23
Avg. Apps	1.0	1.0	2.7	2.2
Avg. APIs	1.0	2.2	2.7	3.3
Avg. arguments	4.0	4.5	3.8	4.4
Max. Seq.	1	4	4	8
Max. Para.	1	1	4	3
Avg. Seq.	1	2.2	1.2	1.9
Avg. Para.	1	1.0	2.2	1.8

Table 2: The data statistics of our proposed `MetaBench`.

ing samples is around 8.05. We finally manually check each instruction-path pair, and remove some mismatch pairs such as the instruction is simple or API calls can not complete the user instructions, resulting in 200 high-quality samples for each category.

3.4 Data Statistic

Table 2 illustrates the statistics of `MetaBench`. Specifically, there are approximately 10 different APPs for each type and over 20 various APIs in both MS and MM. We provide the list of all APP and API in Table 8. Secondly, the average number of APIs increases from SS, SM to MS, MM, revealing the complex relationship. We also emphasize that the higher number of arguments for each API aligns with the complicated nature of tool execution in practice, as there may be multiple input and returned arguments for one API. Furthermore, we provide statistics about sequential and parallel relationships in each category (Seq. and Para.), revealing the complex graph structure in the dataset. Figure 3 presents one example for each category for better understanding. More analysis can be found in the Appendix A.2.

4 Experiments

4.1 Setup

Models. We choose several LLMs from both open- and closed-source models, aiming to provide a comprehensive evaluation, following (Huang et al., 2024a; Zhuang et al., 2023). Specifically, we choose Mistral-7B (`Mistral-7B-v0.2`) (Jiang et al., 2023), the LLaMa3 series (AI@Meta, 2024) (`Meta-Llama-3-8B/70B-Instruct`), and the Qwen series (Bai et al., 2023) (`Qwen1.5-7B/14B/72B-Chat`) from open-source LLMs. Besides that, we also select GPT3.5 (`gpt-3.5-turbo`) and GPT4 (`gpt-4o`) from closed-source LLMs. We also tried other models such as LLaMA2-7B or Vicuna but we find it difficult for them to output in the required format.

Implementation Details. We set the temperature and top p as 0.1 to reduce randomness. The experiments of open-source models are run on NVIDIA A100 GPUs and those of closed-source models are fulfilled by APIs of OpenAI. To address the limitations imposed by the varying context windows of different LLMs, we adopt a *hierarchical* prompting approach. First, we prompt the LLMs to identify the relevant APP. Once the appropriate APP is determined, we then provide the LLMs with only the API descriptions of these specific APPs.

4.2 Evaluation Metrics

In order to evaluate the LLMs' capabilities of selecting proper APPs, choosing APIs, and fulfilling all arguments to execute the API based on the users' instruction, we carefully design two F1 scores for APP and API, and one overall success rate considering the complexity of the task. We also provide the results of EM metrics in the Appendix.

F1 of App. We first get the precision P_{app} as the number of correctly predicted APPs divided by the total number of APPs predicted by the model:

$$P_{app} = \frac{app\_hit\_num}{app\_pred\_num} \quad (1)$$

and recall R_{app} as the number of correctly predicted APPs divided by the total number of APPs that are in the ground truth as follows.

$$R_{app} = \frac{app\_hit\_num}{app\_ground\_truth\_num} \quad (2)$$

The F1 of App score is 2PR / (P+R), as usual.

F1 of API. Similarly, the metrics of API predictions can be evaluated using $F1_{api}$. Note that we only consider the name of the API here to determine LLM whether or not to choose the right API, and the performance of arguments of APIs is evaluated in the next metric.

Success Rate (Succ): This metric evaluates whether the LLMs can fully execute the user's instruction by correctly identifying all required APPs, APIs, and arguments. It is defined as the

Models	SS			SM			MS			MM		
	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ
Mistral-7B	55.97	16.31	0.51	36.59	15.09	0.50	33.72	6.42	0.00	28.92	7.56	0.00
Vicuna-13B	43.20	3.70	2.00	34.71	4.63	0.50	20.43	3.10	0.00	21.05	2.52	0.00
LLaMA3-8B	63.04	42.67	23.23	37.20	25.33	0.50	30.65	19.52	0.10	26.39	17.80	0.05
LLaMA3-70B	71.20	70.00	50.00	46.48	46.96	10.50	32.61	32.96	2.50	28.97	28.53	0.50
QWen1.5-7B	48.14	19.54	0.00	30.13	16.71	0.00	23.24	10.11	0.00	23.76	11.55	0.00
QWen1.5-14B	72.89	28.41	10.10	41.89	25.51	1.50	42.22	21.98	0.80	32.36	15.07	0.00
QWen1.5-72B	81.23	24.28	12.50	**51.89**	25.27	1.00	**45.94**	13.42	0.62	**38.53**	11.51	0.00
GPT-3.5	63.60	57.95	30.81	41.49	43.65	6.50	33.17	34.53	7.00	27.79	28.09	1.00
GPT-4o	**88.31**	**86.87**	**70.92**	50.83	**50.57**	**20.50**	39.39	**39.14**	**11.00**	32.62	**32.35**	**2.00**

Table 3: The main results of different LLMs on MetaBench. Bold highlights the best score among all models, and underline underscores the best score under the same model scale

proportion of instances where all elements—APP, API, and arguments—are in perfect alignment with the ground truth, considering the complex dependency relationship between different APIs across APPs, resulting in a direct measure of model capability in full instruction fulfillment. Since there may exist different output orders, we calculate this at the structure level since the execution structure is unique.

4.3 Main Results

Table 3 shows the results of different LLMs for different types of user instructions on MetaBench, respectively. Several conclusions can be drawn from the results [3].

Overall, GPT-4o achieves the best overall performance, while LLaMA3-70B sometimes outperforms GPT-3.5, mostly in scenarios only involving single APP. In general, other models significantly lag behind GPT-4o in all types of instructions, and only QWen1.5-72B or LLaMA3-70B achieves better or competitive performance compared with GPT-4o. Despite significant advancements in LLMs, the existing models still fall short in addressing the complexities of planning cases such as multiple APPs and multiple APIs. One fact is that all LLMs only get less than 3% Succ in MM situations.

As the size of the model increases, the performance can get further improved regardless of the type of instructions and the improvement becomes less significant with multiple APPs. As evidenced by LLaMA3 and QWen1.5 series models, we can find that large models mostly lead to better performance. However, when the instruction requires coordination between multiple APPs, most models show a significant drop in performance and some models even get 0 at Succ, such as QWen1.5-7B and 14B. Moreover, the $F1_{app}$ can get around 10% improvement in a single APP while only less than 5% in LLaMA3 series models.

The complexity of planning highly impacts the performance of these models. From the varying scores of different LLMs across different scenarios, a trend in performance emerges: the observed order of performance is approximately: MM < MS < SM < SS. This trend exists in most LLMs such as GPT-4o, QWen1.5-14B, LLaMA3-8B, and LLaMA3-70B. The slight difference between SM and MS can be attributed to different percentages of specific data examples such as the number of APPs and APIs. This kind of trend also aligns well with our intuition that the MM scenario is the most complicated, followed by MS and SM, and SS is the simplest.

5 Analysis

In this section, we conduct a comprehensive analysis, aiming to answer three research questions. **RQ1**: *How do the parallel and sequential dependencies influence the model performance?* (Sec 5.1) **RQ2**: *Is it necessary to identify APP first to reduce the context window?* (Sec 5.2) and **RQ3:** *What is the major bottleneck of current LLMs* (Sec 5.3), *and can fine-tuning or in-context learning alleviate it?* (Sec 5.4, 5.5).

5.1 The Effects of Dependency Structures

We classify the dependency structures among APIs as twofold: parallel execution and sequential execution. For each data sample, we measure the parallel execution scale by the number of connected components of APIs and use the average size of these API-connected components as the sequential

[3]The conclusions are consistent with EM results at Table 11.

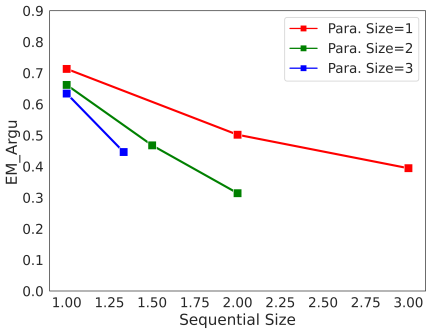

Figure 4: The relationship between GPT-4o's performance with parallel and sequential scaling. Both parallel and sequential scaling cause challenges for model performance.

execution scale. The data sample with a sequential scale of 1 means no sequential dependencies among APIs. All of the APIs can be finished in a parallel way. Then, we classify the data samples of `MetaBench` based on the above criteria and discard the categories with less than 10 samples.

We illustrate the Exact Match (EM) of Arguments of GPT-4o in Figure 4 since arguments are directly related to the dependency relationship. First of all, when the parallel scale is fixed, an increased sequential scale becomes more challenging for GPT-4o, and vice versa. Secondly, GPT-4o appears to struggle more with sequential-complex data than parallel-complex samples. The gap between different para. size (i.e., when seq. size is fixed) is much smaller than the gap between different seq. size (i.e., when para. size is fixed).

5.2 The Effects of Different Prompting

In the main experiments, we initially required LLMs to select candidate APPs based on user input and the APP's descriptions, and then generate API calls, resulting in *hierarchical* prompting. Recently, many studies have expanded the context of LLMs to 200K or more (Huang et al., 2024b). Many of these works proposed LLMs with a context window that is sufficient to accommodate all the descriptions of APPs and APIs at once (*flat* prompting). Therefore, this section explores how the model would perform if we directly provided all apps and APIs to the model. We test GPT-3.5 and GPT-4o and compare the results in Figure 5.

We can observe that flat prompting has impacted the performance of the GPT-3.5, with obvious declines in metrics such as $F1_{app}$ scores across data types. We attribute this to the introduction of a large amount of irrelevant information, which affects the model's understanding and extraction of

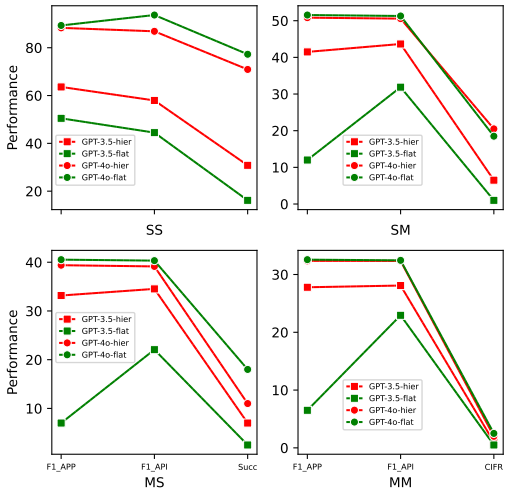

Figure 5: The performance gap between hierarchical and flat prompting on GPT-3.5 and GPT-4o.

Category	Keys		Values		
	I	D	I	D	T/S
SS	6.1	-	6.6	-	42.1/26.3
SM	5.5	8.0	2.5	75.5	27.1/15.2
MS	6.0	1.0	6.0	30.0	45.1/8.8
MM	19.0	15.0	6.0	82.0	36.8/24.6

Table 4: Error analysis of GPT-4o on `MetaBench`. I and D stand for independent and dependent variables or values, respectively, between multiple APIs. T/S refers to time-related or space-related values, such as start date and location.

useful APPs and APIs. Surprisingly, the GPT-4o model achieved better performance using flat prompting. We believe this is due to the GPT-4o's more powerful long-context understanding capabilities, which allow it to accurately identify the required APP and API. Moreover, the absence of the error propagation effect that occurs during the first APP selection step of hierarchical prompting, has led to a clear improvement in performance. However, flat prompting requires a strong contextual capability that few models possess, and it necessitates the input of a large number of irrelevant tokens, which incurs additional computational power consumption.

5.3 Error Analysis

We further conduct error analysis at the argument level since it is directly related to different relationships between multiple APIs, to identify potential bottlenecks of the current best model: GPT-4o. Specifically, there are two main categories of errors to consider: *1) key error.* It occurs when the model predicts fewer keys than expected to successfully execute the API call, and it can be further divided

Settings	SS			SM			MS			MM		
	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ	$F1_{app}$	$F1_{api}$	Succ
GPT-4o	88.31	86.87	70.92	50.83	50.57	20.50	39.39	39.14	11.00	32.36	32.35	2.00
3-shot	93.73	90.73	81.63	51.16	50.90	13.50	40.12	39.92	12.50	32.72	32.73	2.50
4-shot	93.23	89.72	79.59	50.96	50.70	14.00	40.29	40.29	10.50	32.44	32.44	3.00
5-shot	93.70	91.18	79.59	50.32	50.06	14.00	40.33	40.12	12.50	32.36	32.36	2.50

Table 5: In-context learning results of GPT-4o on `MetaBench`.

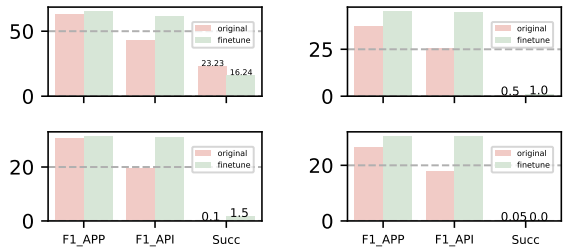

Figure 6: The performance gap between original LLaMA3-8B and finetuned one.

into two types: **I**ndependent: The missing or incorrect keys are from the independent variables or arguments and **D**ependent: The missing or incorrect keys are from the dependent variables or arguments; and *2) value error*. it occurs when the model predicts values that do not match the ground truth values, given the name of the key. Value errors can also be divided into I and D types.

Table 4 presents the results. It can be found that as complexity increases, errors also increase. The lower D-key error and D-value error in MS can be attributed to a smaller percentage of dependency cases in this category. Out of all types of errors, the D-value error appears to be the biggest bottleneck or challenge for the LLM. Further analysis reveals that the value errors are particularly prevalent for time and space-related keys. For example, the language models may struggle to accurately recognize or reason about date/time expressions used in the user's input, such as "next Monday".

5.4 The Effects of Fine-tuning

We additionally collected around 1,000 samples for each category from the training dataset of SGD, resulting in 4,000 samples total. We then used this mixed dataset to fine-tune the LLaMA3-8B model. Figure 6 shows the final results. Further fine-tuning on in-domain data did bring some improvement in the F1 score of the APP and API, but can not boost the performance for the Succ. Upon closer inspection, we found that the major reason for the lower performance on Succ was due to issues with recognizing or matching the keys and values in the input arguments. The model sometimes failed to recognize all the necessary input keys and values, or mistakenly used keys from other APIs. This appears to be strongly related to the complexity of the task. Factors like the dependency relationships between multiple APIs, as well as the lengthy API descriptions, made it challenging for the LMs to fully capture the necessary patterns and logic.

5.5 The Effects of In-context Learning

Table 5 shows the in-context performance of GPT-4o when using different shots at the demonstrations. Specifically, we randomly sample (instruction, outputs) from the same training set created during fine-tuning according to the used APP in the current instruction. For example, if the used APP in current instruction is Hotel, we sample the first 3 appeared samples with the same APP in the training set to form 3-shot demonstrations, aiming to save the space of additional API descriptions and make the agent familiar the utilization of current API. From the table, we find that in-context learning shows some improvement in simpler cases, such as SS ($\approx$ 10 points increase on Succ). However, the performance does not improve further as situations become more complex and even decreases in scenarios like SM or MS, highlighting the challenges of complex planning. The worst performance in SM may be strongly related to our sampling strategy, as we only consider the APP level rather than different APIs within the same APP. More effective in-context learning for complex planning is desired and warrants further exploration and attention.

6 Conclusion

In this paper, we introduce a new benchmark, `MetaBench`, addressing the challenge of complex user instructions that require the involvement of multiple APIs. These scenarios demand advanced planning capabilities from LLMs to effectively handle graph structures and ensure permission isolation in practical applications. We left the self-evolving or more effective fine-tuning framework in our future work.

7 Limitations

We acknowledge the following limitations in terms of the evaluations and benchmarks.

Evaluations. We do not consider the existing agent framework since we mainly focus on the base capabilities of various LLMs on this new benchmark. We anticipate that introducing additional reflection or a carefully designed agent framework may further boost the performance of original LLMs.

Benchmarks. We mainly take advantage of existing task-oriented datasets to build our benchmark, which brings two limitations: 1) We main focus on text-based natural language interactions while the API also works at different modalities. We leave this in future work, and 2) We do not consider APPs with overlap or similar functions, but they exist in practice such as different platforms to buy tickets. We argue that these apps can often be distinguished through minor modifications to the app names and APIs. The specific choice of which app a user selects ultimately comes down to individual user preferences, which is outside the scope of this paper.

8 Ethical Considerations

In conducting our research, we have thoroughly reviewed and ensured compliance with ethical standards. Our study utilizes existing datasets, which have been publicly available and previously vetted for ethical use. These datasets have been carefully selected to avoid any form of offensive or biased content. Therefore, we consider that our research does not present any ethical issues. The data used is ethically sourced, the analysis is unbiased, and all procedures align with established ethical guidelines.

References

AI@Meta. 2024. Llama 3 model card.

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*.

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Inigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. Multiwoz–a large-scale multi-domain wizard-of-oz dataset for task-oriented dialogue modelling. *arXiv preprint arXiv:1810.00278*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2024. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, Xin Jiang, Ruifeng Xu, and Qun Liu. 2024a. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios.

Yunpeng Huang, Jingwei Xu, Junyu Lai, Zixu Jiang, Taolue Chen, Zenan Li, Yuan Yao, Xiaoxing Ma, Lijuan Yang, Hao Chen, Shupeng Li, and Penghao Zhao. 2024b. Advancing transformer architecture in long-context large language models: A comprehensive survey.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control.

Xiao Liu, Hanyu Lai, Hao Yu, Yifan Xu, Aohan Zeng, Zhengxiao Du, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023a. Webglm: Towards an efficient web-enhanced question answering system with human preferences. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4549–4560.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2023b. Agentbench: Evaluating llms as agents.

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

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.

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

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

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018a. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018b. Virtualhome: Simulating household activities via programs.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023. Tool learning with foundation models.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8689–8696.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face.

Hongru Wang, Minda Hu, Yang Deng, Rui Wang, Fei Mi, Weichao Wang, Yasheng Wang, Wai-Chung Kwan, Irwin King, and Kam-Fai Wong. 2023. Large language models as source planner for personalized knowledge-grounded dialogues. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9556–9569, Singapore. Association for Computational Linguistics.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. 2022. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11298, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. Rewoo: Decoupling reasoning from observations for efficient augmented language models.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2023. Webshop: Towards scalable real-world web interaction with grounded language agents.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools.

A Data Collection

A.1 Prompt Details

Your task is to generate a complex instruction in one sentence which exactly reflect what user want to do during the dialogue with the dialogue system as follows.

Please give all specific values of user requirements in user aware arguments {`user_aware_arguments`}.

You should not know any values of other arguments specified by the system side.

Table 6: The prompt used to prompt LLM to generate the summarized instruction

Please evaluate the given instruction based on the following criteria:

Fluency:
– Evaluate the prompt's clarity, coherence, and ease of understanding.
– Consider factors such as the organization, language flow, and presentation of the prompt.

Diversity:
– Evaluate the range of topics, perspectives, and related APP and APIs covered by the prompt.

Please only output the overall score considering both of fluency and diversity. The overall score should be a value between 1 and 10, with 10 representing the best.

Table 7: Prompts to evaluate the quality of generated instructions.

A.2 Data Statistics

Definition of Parallel and Sequential In this section, we delve into the execution logical structure inherent in each data sample to have a better understanding of task complexity. We conceptualize the APIs used within a single data instance as nodes within a directed graph and the dependency among them as the directed edges. Consequently, we analyze the interrelations among all APIs within the data sample and construct a corresponding graph for each of them. It is important to notice that not all APIs within a sample are interdependent, and some may operate independently. As a result, the APIs within the same data sample generally form several distinct **components**. We treat each component as a unit to perform topological sorting. The execution process of each unit can be parallel, while the procedure within the component is sequential.

As shown in Figure 3, the illustrated MM sample needs to leverage 2 APIs from APP-1 and 3 APIs from APP-2 to fulfill the user instruction. Though APIs of APP-1 are interdependent, they do not need results from APIs of APP-2. Hence the formed graph of this sample has 2 components, with a size of 2 and 1. These two components can be fulfilled simultaneously, but the results from APIs of APP-1 or APP-2 need to be executed one by one.

Above all, to quantify the complexity inherent in these interactions, we compute both the average and maximum **sizes** of these components as sequential scale(*Max. and Avg. Seq.* in Table 2), which reflect the complexity of sequential dependency among the APIs. Additionally, we measure both the average and the maximum **number** of components within each sample (*Max. and Avg. # Para.* as the parallel scale, providing insight into the level of parallelism among the APIs or components.

As shown in Table 2, the instances of SS and SM are relatively simple. Since the samples of MS have the most complex sequential scales. The samples of MM are the most complex since their parallel and sequential scales are relatively larger than the others.

B Experimental Details

App	APIs
Rents	*getcarsavailable*, *reservecar*, *getride*
Hotels	*searchhouse*, *bookhouse*
Services	*book_stylist_appointment*, *find_stylist_provider*, *book_therapist_appointment*, *find_therapist_appointment*
Restaurant	*reserverestaurant*, *findrestaurants*
Movies	*buymovietickets*, *findmovies*, *gettimesformovie*, *reviewmovies*
Trains	*gettraintickets*, *findtrains*
Events	*findevents*, *buyeventtickets*
Travel	*findattractions*
Buses	*findbus*, *buybusticket*
Flights	*searchonewayflight*, *searchroundtripflights*
Payment	*requestpayment*, *makepayment*
Music	*playmedia*, *lookupmusic*
Weather	*getweather*

Table 8: List of All Apps and their corresponding APIs in the `MetaBench`.

Your task is to determine the required App list according the description of each App and user requirements.

Here is the information about all accessible Apps: {`app_desc`}

Make your response short and concise. Your **ONLY** need to return needed app names and your output MUST follow this format: [app1, app2, ...]

User Instruction: {`user_instruction`}

Table 9: Prompts to select APP first.

Your task is to generate App name and corresponding API calls to complete the user requirements according to given descriptions of all Apps and APIs.

Here is the information about all accessible Apps and corresponding APIs. {app_api_list}

Your output should follow the format as follows:
app1: [returned_argument1, returned_argument2, ... = app1_api1(#argument1=value1, #argument2=value2, ...)]
app1: [returned_argument1, returned_argument2, ... = app1_api2(#argument1=value1, #argument2=value2, ...)]
app2: [returned_argument1, returned_argument2, ... = app2_api1(#argument1=value1, #argument2=value2, ...)]

Here are explanations:
1. API Naming Convention
– The API call format is [returned_argument1, returned_argument2, ... = app1_api1(#argument1=value1, #argument2=value2, ...)].
– app1 signifies the name of app1, and app1_api1 signifies the name of api1 in the app1.

2. Arguments
– argument1 is the first input arguments for the corresponding api, and so on.
– returned_argument1 is the first output arguments from the corresponding api, and so on.
– Input arguments include both required and optional arguments as described in the corresponding API description of App.
– The order and names of input and returned arguments must exactly match the given description.

3. Values of Input Arguments
– If specified by the user, replace the placeholder with the actual value.
– If not specified by the user, omit the optional arguments from the API call.
– If an argument value is dependent on another API's output, use the name of the returned argument as the value.
– There are no default values for any arguments. All required arguments must be provided by the user or through dependencies on other APIs' outputs.
– You should be careful about the date value, you need to infer it based on current date "2019-03-01".

4. Order of Execution:
– Execute APIs in a sequence that respects their dependencies. For example, if api2 requires an output from api1, ensure api1 is executed before api2.
– Handle cases where multiple APIs' outputs are required for a single API's input by waiting for all dependent APIs to execute before calling the dependent API.

Example:
If api2 in app1 depends on the output of api1 in app1 and an optional argument is not provided by the user:
app1: [output1 = app1_api1(#argument1=value1)]
app1: [output2 = app1_api2(#argument2=output1)]

If api3 in app2 requires outputs from both api1 in app1 and api2 in app1:
app1: [output1 = app1_api1(#argument1=value1)]
app1: [output2 = app1_api2(#argument2=output1)]
app2: [output3 = app2_api3(#argument3=output1, #argument4=output2)]

User Instruction: {user_instruction}

Table 10: Prompts to generate the final planning path to fulfill the user instruction.

Models	SS		SM		MS		MM	
	APP	API	APP	API	APP	API	APP	API
Mistral-7B	27.27	14.14	19.50	4.50	1.50	1.00	2.00	0.00
Vicuna-13B	31.82	21.21	7.00	3.00	1.50	0.50	0.00	0.00
LLaMA3-8B	47.98	47.47	19.00	17.50	12.50	9.50	4.50	5.50
LLaMA3-70B	60.94	<u>58.33</u>	<u>51.00</u>	<u>49.00</u>	12.00	6.50	16.00	8.50
QWen1.5-7B	28.28	12.63	11.50	4.00	2.50	0.50	4.00	1.50
QWen1.5-14B	56.57	41.92	10.50	10.00	5.60	4.00	1.50	1.50
QWen1.5-72B	<u>71.88</u>	32.29	38.50	9.50	2.47	1.85	4.50	3.50
GPT-3.5	44.44	52.02	30.50	31.00	<u>31.00</u>	<u>19.00</u>	<u>18.50</u>	<u>19.50</u>
GPT-4o	**79.59**	**78.06**	**55.50**	**51.50**	**35.50**	**26.50**	**29.50**	**24.00**

Table 11: The Exact Match (EM) results of different LLMs on `MetaBench`. Bold highlights the best score among all models, and underline underscores the best score under the same model scale