StableToolBench-MirrorAPI: Modeling Tool Environments as Mirrors of 7,000+ Real-World APIs

Zhicheng Guo[1,2*†], **Sijie Cheng**[1,2,3*], **Yuchen Niu, Hao Wang**[4], **Sicheng Zhou**[2,5], **Wenbing Huang**[6], **Yang Liu**[1,2]
[1]Dept. of Comp. Sci. & Tech., Institute for AI, Tsinghua University, Beijing, China
[2]Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China
[3]RayNeo [4]Google [5]The University of Toronto, Canada
[6]Gaoling School of Artificial Intelligence, Renmin University of China
{guo-zc21, csj23}@mails.tsinghua.edu.cn

Abstract

The rapid advancement of large language models (LLMs) has spurred significant interest in tool learning, where LLMs are augmented with external tools to tackle complex tasks. However, existing tool environments face challenges in balancing stability, scalability, and realness, particularly for benchmarking purposes. To address this problem, we propose MirrorAPI, a novel framework that trains specialized LLMs to accurately simulate real API responses, effectively acting as "mirrors" to tool environments. Using a comprehensive dataset of request-response pairs from 7,000+ APIs, we employ supervised fine-tuning and chain-of-thought reasoning to enhance simulation fidelity. MirrorAPI achieves superior accuracy and stability compared to state-of-the-art methods, as demonstrated by its performance on the newly constructed MirrorAPI-Bench and its integration into StableToolBench.

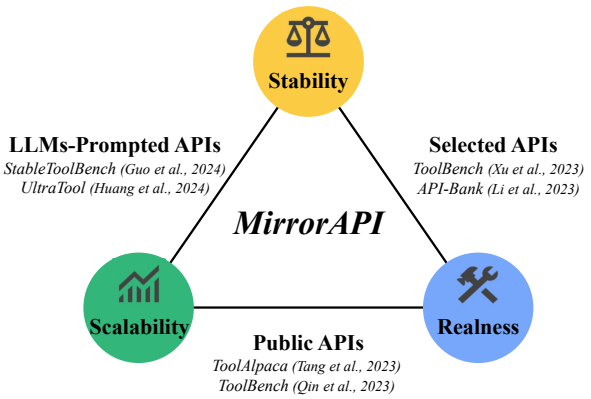


Figure 1: While existing API environments prioritise one or two of stability, scalability, or realness, MirrorAPI effectively harmonises all three aspects.

1 Introduction

Recently, the remarkable reasoning capabilities exhibited by large language models (LLMs) (Qwen Team, 2025; DeepSeek-AI, 2025; AI@Meta, 2024; Gemini Team, 2024) have catalyzed growing interest in tool learning (Yang et al., 2023; Ye et al., 2024; Wu et al., 2024). Analogous to the way humans leverage tools in the physical world to enhance their cognitive and physical capacities, LLMs augmented with external tools (Qu et al., 2025; Qin et al., 2024) demonstrate unprecedented potential in addressing diverse and complex challenges such as general task solving (Qin et al., 2023; Huang et al., 2024c), web search (Komeili et al., 2022; Gou et al., 2024), and multimodal scenarios (Ma et al., 2024; Wang et al., 2024b).

In a typical tool-using scenario, three main components are involved: (1) a user who provides instructions, (2) a tool-using model that performs actions, and (3) an environment containing tools that provides feedbacks in response to those actions. The environment is critical as it directly relates to the quantity, quality, and stability of available tools. Most environments (Tang et al., 2023; Chen et al., 2024; Basu et al., 2024; Farn and Shin, 2023) use publicly available real-world tools [1], such as RapidAPI [2]. Others use manually selected or created APIs (Li et al., 2023; Xu et al., 2023a), often prioritising stable APIs like Google services or Python libraries. Furthermore, recent studies further explore prompting LLMs to generate APIs and simulate their behaviours according to documentation. (Huang et al., 2024a; Wu et al., 2024).

However, existing environments struggle to balance stability, scalability and realness. Environments built on large-scale public APIs often exhibit instability, leading to inconsistent results over time (Guo et al., 2024). This stems from developer

[†]Project Leader [*]Equal contribution
GitHub: THUNLP-MT/StableToolBench
Dataset and Models: https://huggingface.co/stabletoolbench

[1]In this work, we use the terms APIs and tools interchangeably.
[2]https://rapidapi.com/

Findings of the Association for Computational Linguistics: ACL 2025, pages 5247–5270

updates, underlying behavioural modifications, and network connectivity fluctuations. On the other hand, environments relying on manually selected or created APIs lack scalability due to limited labour resources. While LLMs-simulated APIs offer greater stability compared to real-world APIs, there is a significant gap between simulated behaviours and actual API response, as demonstrated in our experiments in Section 4.3.

To balance stability, scalability and realness of tool-learning environments, we propose `MirrorAPI`, a novel framework that trains specialized LLMs to act as "mirrors" to tool environments by accurately replicating real API responses. A tool can be conceptualized as a wrapper around its underlying system, abstracted through manually crafted documentation. Given that real documentation data is both informative and practical to collect, it is natural to simulate tool behaviour based on documentation and user requests. However, the intricate mechanism of complex underlying systems cannot be fully captured in documentation alone. Therefore, it is crucial to align simulated behaviour not only with documentation but also with the practical workings of real-world APIs.

To train a specialised LLM, named `MirrorAPI`, we first collect a dataset of real request-response pairs spanning 49 categories and 7,000+ APIs with documentation from RapidAPI for supervised fine-tuning (SFT). To further capture the implicit factors in underlying real API systems, we incorporate reasoned explanations of API mechanisms into the training data. Specifically, given request-response pairs, we use OpenAI `o1-preview` to generate chain-of-thought (CoT) rationales that explain the underlying working mechanisms of APIs (Yang et al., 2024). The `MirrorAPI` is trained in both SFT and CoT modes, distinguished by special prompt tokens (Wang et al., 2023). During inference, the model defaults to SFT mode for consistent and better performance.

To evaluate the model, firstly, we construct `MirrorAPI-Bench` by defining an in-distribution (ID) and an out-of-distribution (OOD) sets based on real request-response pairs with APIs seen and unseen during training. Experiments demonstrate that `MirrorAPI` outperforms state-of-the-art LLM prompting methods in simulating real APIs, excelling in documentation and instruction following capabilities while achieving the highest similarity to real responses. Secondly, we integrate `MirrorAPI` as the tool environment into StableToolBench (Guo et al., 2024). Results show that `MirrorAPI` not only maintains full environment stability but also produces outcomes comparable to real environments. Beyond benchmarking, `MirrorAPI` can enhance tool-using models by providing step-wise feedback or expanding the size and diversity of training data, offering a promising direction for future research.

2 Methods

2.1 Request-Response Pairs Collection

To train specialised LLMs for simulating real APIs, we collect training data by calling the real RapidAPI server using LLM-generated requests. This process consists of three stages: (1) collecting real request-response pairs, (2) filtering request-response pairs, and (3) synthesising request-response pairs.

2.1.1 Collecting Real Request-Response Pairs

We first construct a large collection of APIs and call them uniformly to gather data. The data collection process is illustrated in Figure 2. Following ToolBench (Qin et al., 2023), we crawl the latest tools and their documentation from RapidAPI. Then, we exploit LLMs (`gpt-4o`) to generate API requests for the available APIs. When directly prompted to write requests based on API documentation, LLMs tend to produce simple calls with limited diversity. To enhance data complexity and diversity, we employ a two-stage scenario-based approach as shown in Appendix A. First, we prompt LLMs to generate realistic usage scenarios for a given API based on its documentation, setting the temperature to 1.0 to maximise diversity. Next, we prompt LLMs to write API requests that address the tasks described in these scenarios, setting the temperature to 0.1 to ensure stability and precision. Finally, we generate 200 calls for each available API using the predefined method and record the responses. Each request was attempted up to three times in case of failures.

2.1.2 Filtering Request-Response Pairs

First, real API calls are often unsuccessful due to various reasons, such as incorrect requests generated by LLMs, discrepancies between locally crawled API documentation and real-time API behaviour, and temporary API unavailability, which may harm model performance. To mitigate these issues, we filter out unsuccessful calls using the same rules outlined in StableToolBench as discussed in

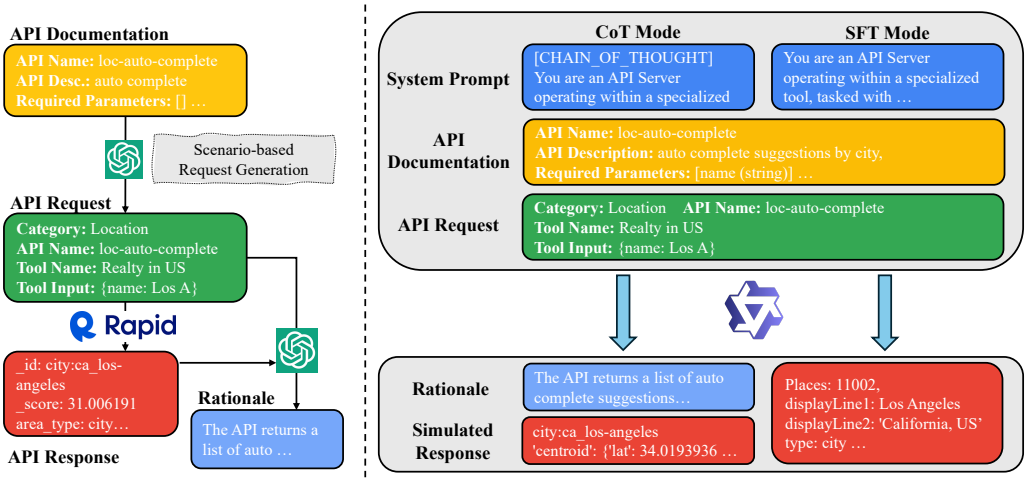


Figure 2: The phases of data construction (left) and the training of `MirrorAPI` (right).

Appendix D. Second, to ensure the trained simulator can return failure messages for requests that violate documentation requirements, we retain around 10,000 unsuccessful calls caused by parameter-related issues, such as missing required parameters or incorrect parameter values. Finally, due to the large number of poorly written documentation, some real API responses may deviate from their documentation, which can harm model performance. To overcome this, we elicit `gpt-4o` with the prompt as shown in Table 8. This effectively filters out request-response pairs that do not align with their documentation.

2.1.3 Synthesizing Request-Response Pairs

To further enlarge and balance the dataset, we use `gpt-4o` to generate synthetic request-response pairs based on existing examples. This serves two purposes: (1) to ensure sufficient data for robust model training, and (2) to mitigate the scarcity of data samples for long-tail APIs. For each generation, the prompt includes a list of 10 real API examples, comprising API documentation, LLM-generated requests, and real responses. By varying temperature and seed values, we encourage the model to produce a diverse range of request-response pairs. We then employ an exact match function to ensure that each generated pair is unique and does not already exist in the dataset. After generation, we conduct a quality check and filter out pairs with identical requests but differing responses, and ones with responses inconsistent with the documentation.

Dataset	Samples	APIs	Categories
Training	95,272	7,437	49
OOD Succ	200	38	21
OOD Fail	100	22	14
ID High	100	96	14
ID Medium	100	97	15
ID Low	100	81	14

Table 1: Dataset Statistics

2.1.4 Statistics and `MirrorAPI-Bench`

In total, we collect 95,872 API request-response pairs, spanning 7,500 APIs across all 49 categories in RapidAPI. Then, we partition the dataset into training and testing sets, named `MirrorAPI-Bench`. Specifically, the OOD testing set comprises 200 successful and 100 unsuccessful pairs from 49 APIs not included in the training data. Additionally, we construct three ID test sets, each consisting of 100 successful request-response pairs drawn from the training APIs, representing the high-resource, medium-resource, and low-resource groups. These sets correspond to the top third, middle third, and bottom third quantiles of APIs, respectively, as illustrated in Figure 4. The detailed statistics of the datasets are summarised in Table 1.

2.2 Chain-of-Thought Data Collection

Simply training models to follow API documentation does not fully close the gap between the simulated and real APIs as the model is ignorant of the working mechanism behind the real APIs. Therefore, we propose to improve the simulation model by augmenting reasoning API mechanism. The data collection process is illustrated in Figure 2.

To generate proper rationales, we fully utilise

Models	OOD Succ	OOD Fail	ID High	ID Medium	ID Low
GPT-4o	5.93	2.06	4.58	4.76	3.92
GPT-4o-mini	6.10	1.95	5.17	4.09	3.57
GPT-4o CoT	4.45	3.72	4.06	4.26	4.20
o1-preview	**7.67**	1.81	6.11	5.47	4.11
Llama-3.1-8B-Instruct	1.27	1.86	1.74	1.61	2.28
Qwen-2.5-7B-Instruct	2.99	1.71	3.33	2.89	2.27
Deepseek-R1-Distill-Qwen-32B	5.97	1.82	4.61	4.33	3.38
Deepseek-R1-Distill-Qwen-7B	3.03	1.83	2.42	3.12	2.56
`MirrorAPI` CoT	6.51	**8.64**	**8.91**	**8.79**	**9.01**
`MirrorAPI` SFT	<u>6.86</u>	<u>8.28</u>	<u>8.76</u>	<u>8.63</u>	<u>8.74</u>

Table 2: Performance of observation following on `MirrorAPI-Bench`. OOD Succ and OOD Fail stand for the test sets of OOD successful and unsuccessful calls respectively. The ID High/Medium/Low stands for the test sets of ID calls with a large/medium/low number of training samples. Tables below use the same notation.

the real responses in the data (Yang et al., 2024). Specifically, for a request-response pair (R, O), where R is a request to an API and O is the response from the API, we use a general reasoning LLM (OpenAI `o1-preview`) to reason the implicit working mechanism, which serves as a rationale for simulation. The prompt is shown in Table 9, i.e.,

$$R_{o1} = LLM_{o1}(R, O, Doc)$$

where Doc denotes the API documentation.

We randomly sample 43,597 successful call-response pairs and 2,000 unsuccessful pairs from the training set to generate rationales. To ensure the quality of the generated rationales and avoid excessively verbose explanations, we apply a length filter, restricting the maximum token count of the final outputs to 2,560. As a result, we obtain 42,465 successful and 1,975 unsuccessful request-response pairs each annotated with a generated rationale. Note that all CoT data is constructed based on the SFT training data, meaning no request-response data is created here.

2.3 Model Training

We train the environment model in both SFT and CoT modes simultaneously. Namely, we mix 95,272 SFT samples and 44,440 CoT samples in the final training data. This combination ensures that the `MirrorAPI` benefits from the extensive SFT data while also leveraging the performance-boosting advantages of CoT. Illustration of the two modes can be seen in Figure 2.

To distinguish the two modes during training and testing, following OpenChat (Wang et al., 2023), we introduce a special tag `[chain-of-thought]` at the beginning of the CoT mode system prompt. This approach allows the simulator to operate in two distinct modes by switching between system prompts. The prompts for the SFT and CoT cases are shown in Table 10 and 11, respectively.

During training, we finetune a `Qwen2.5-7B-Instruct` (Qwen Team, 2025) model to generate API responses in the SFT mode, given the user request and API documentation. In the CoT mode, the model is trained to generate both the rationale and the simulation output. Formally, these two modes can be expressed as:

$$S = LLM_{simu}(R, Doc|P_{SFT}) \quad \textit{(SFT mode)}$$
$$[R_{o1}, S] = LLM_{simu}(R, Doc|P_{CoT}) \quad \textit{(CoT mode)}$$

where S, R, Doc and LLM_{simu} represent the simulation output, the user request, the API documentation and the LLM to train, respectively. P_{SFT} and P_{CoT} are the system prompts used in the two modes. R_{o1} is the rationale generated by `o1-preview`.

At testing time, the model generates simulations based on user requests and API documentation in the SFT mode, while in the CoT mode, it additionally produces rationales before delivering final outputs. Mathematically, this can be represented as:

$$S = LLM_{simu}(R, Doc|P_{SFT}) \quad \textit{(SFT mode)}$$
$$[R_s, S] = LLM_{simu}(R, Doc|P_{CoT}) \quad \textit{(CoT mode)}$$

where S, R, Doc and LLM_{simu} represent the simulation output, the user request, the API documentation and the trained model respectively. P_{SFT} and P_{CoT} are the system prompts used in the two modes. R_s is the rationale the model generates before giving a simulation output. Note that during inference, we set the SFT mode as default.

Models	OOD Succ		OOD Fail		ID High		ID Medium		ID Low	
	BLEU	Cosine	BLEU	Cosine	BLEU	Cosine	BLEU	Cosine	BLEU	Cosine
GPT-4o	13.4	63.1	10.7	36.0	10.0	55.8	12.4	53.4	14.8	52.8
GPT-4o mini	14.7	62.3	11.3	36.6	10.3	56.2	12.5	53.1	14.5	53.6
GPT-4o CoT	13.4	60.3	13.6	42.8	11.7	55.0	14.2	52.8	16.5	54.5
o1-preview	19.2	65.6	8.4	37.2	12.1	57.5	14.6	55.7	15.9	54.3
Llama-3.1-8B-Instruct	5.3	49.6	11.2	43.7	6.3	47.2	9.6	46.3	9.9	47.8
Qwen-2.5-7B-Instruct	7.5	53.5	9.2	36.3	7.7	50.6	11.3	48.4	7.7	47.3
Deepseek-32B	12.6	63.6	9.7	38.1	9.2	55.5	13.4	53.8	15.9	53.9
Deepseek-7B	12.3	60.0	12.0	38.0	10.8	52.3	12.6	51.9	12.8	51.8
MirrorAPI CoT	31.6	66.7	84.9	90.7	73.6	87.6	77.5	88.0	84.7	93.4
MirrorAPI SFT	**35.6**	**69.9**	**89.9**	**94.1**	**74.2**	**88.7**	**80.0**	**89.8**	**86.3**	**94.2**

Table 3: Comparison of BLEU scores and LLM cosine similarity between real and simulated responses across different groups. BLEU and Cosine stand for the BLEU metric and LLM consine similarity metrics respectively. Deepseek-32B and Deepseek-7B are Deepseek-R1-Distill-Qwen-32B and Deepseek-R1-Distill-Qwen-7B.

3 Evaluation on MirrorAPI-Bench

3.1 Evaluation Metrics

Following Documentation and Instructions. To assess the documentation and instruction following capability of the simulation model, we adopt an LLM-as-a-judge framework, following FastChat (Zheng et al., 2023). The prompt is shown in Table 12. Specifically, we employ gpt-4o as the evaluator model, judging whether the simulated responses align with the documentation and user requests, rating from 1 to 10.

Similarity to Real Responses. In addition to evaluating the following capability, we directly compare the simulated responses with those real responses from real-world APIs. To do so, we adopt two metrics: BLEU (Papineni et al., 2002) and LLM cosine similarity (Zhang* et al., 2020). Specifically, we calculate BLEU-4 scores from NLTK toolkit (Bird et al., 2009). For the LLM cosine similarity, we first encode the real API responses and simulated responses with OpenAI text-embedding-3-small, respectively. Then we calculate the cosine similarity score between the real and simulated responses pairwise. Mathematically,

$$S = \frac{1}{N}\sum_{n=1}^{N}\cos(Enc(R_{real}), Enc(R_{simu}))$$

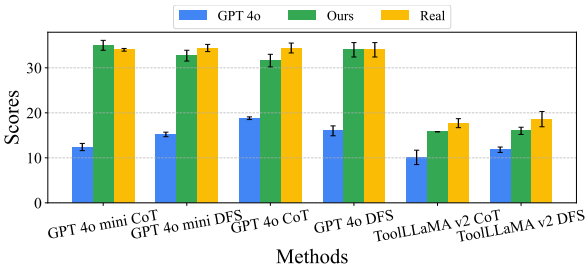
where S, N, R_{real}, R_{simu} and Enc are the similarity score, total number of samples in the test set, real responses, simulated responses, and the encoding LLMs, respectively.

3.2 Baseline

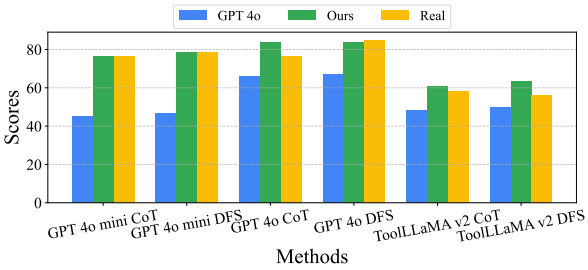
We evaluate several baselines on the simulation test data, including both general-purpose and reasoning-enhanced LLMs. These include gpt-4o and gpt-4o-mini (OpenAI, 2024a), which are API-based, general-purpose LLMs. Additionally, we consider gpt-4o CoT, a variant of gpt-4o integrated with chain-of-thought reasoning through the API mechanism. We also include o1-preview (OpenAI, 2024b), a general-purpose LLM known for its strong reasoning capabilities. For open-source alternatives, we evaluate Llama-3.1-8B-instruct (AI@Meta, 2024) and Qwen-2.5-7B-Intruct (Qwen Team, 2025), both of which are general-purpose LLMs. Finally, we assess the Deepseek-R1-Distill-Qwen-7B and Deepseek-R1-Distill-Qwen-32B (DeepSeek-AI, 2025), which are open-source models with enhanced reasoning abilities.

3.3 Results

We run our model with both CoT and SFT prompts, keeping the temperature at 0 during generation for all models. The results for the documentation and instruction-following metrics are presented in Table 2. BLEU scores and LLM Cosine Similarity scores are detailed in Table 3. As shown in the tables, our trained simulation model outperforms all baselines, except for o1-preview, on OOD successful tasks when evaluated using the documentation and instruction-following metrics, demonstrating the generalizability of our approach. Moreover, our model surpasses all other baselines, including o1-preview, on OOD failing tasks and all ID subtasks across all metrics. This result underscores the effectiveness of training on API calls in enabling accurate simulation of those APIs. Interestingly, o1-preview outperforms all prompting methods on the OOD successful tasks, showing the poten-



(a) Solvable pass rate scores. We run all models once, evaluate three times and take the average results. Error bars are the standard deviation during evaluation.



(b) Final Answer Completeness scores.

Figure 3: Comparison between the real environment, GPT 4o, and `MirrorAPI` simulated environments.

tial of reasoning for simulation. Additionally, our trained SFT model outperforms its CoT counterparts on OOD successful tasks. However, this does not diminish the importance of reasoning-based training. Further insights and supporting evidence are explored in Section 5.1. We further conducted an error analysis of `MirrorAPI` and typical errors can be seen in Appendix H.

4 Environment Simulation for StableToolBench

An important application of the simulation model is its role as an environment model in a tool learning benchmark. To improve the reliability and stability of simulation on StableToolBench, we further finetune `MirrorAPI` on the cache data, named `MirrorAPI`-Cache. To evaluate `MirrorAPI` and `MirrorAPI`-Cache on StableToolBench, we compare the real environment with our `MirrorAPI` in the StableToolBench setting, with both the Solvable Pass Rate (SoPR) and our proposed Final Answer Completeness (FAC) scores. Additionally, we present the performance of several baseline models when `MirrorAPI` is used as the environment model, demonstrating its reliability.

4.1 Training `MirrorAPI`-Cache

To better improve the performance of the model on benchmarks, we fine-tune the model using real API data accumulated from the StableToolBench. This dataset provides a rich collection of real-world API usage data, which is essential for evaluating the model's performance in practical scenarios. By leveraging this dataset, we aim to ensure the model is better aligned with actual user needs and behaviours, thereby improving its ability to generalize to real-world tasks. The dataset used for fine-tuning comprises 110,700 samples, in which 200 of the samples are used as the test set.

4.2 Evalution Metrics

Solvable Pass Rate. We use the SoPR metric in StableToolBench in this project. We evaluate all instances three times and calculate the average scores and standard deviation. We use `gpt-4o` as the evaluator model.[3].

Final Answer Completeness. Besides the Solvable Pass Rate scores in StableToolBench, we propose the new FAC score to measure whether the final answer completes the user query well. In tool learning, users ultimately need the final answer. How the tool-using model reaches that answer is less important. Therefore, it is more meaningful to measure whether the final answer matches the need of the user's request.

To ensure the stability of the metric, we decide to use performant closed-sourced models for annotation and train an offline evaluator. We first randomly sample 3,714 samples from ToolBench training data and let `gpt-4o`, `gpt-4o-mini` and `gpt-4-turbo` to annotate whether the answer completes the user request, given only these two pieces of information. The model is required to answer either solved or unsolved. The annotation prompt is shown in Table 13. We then sample 77 of these samples and invite two human annotators to annotate ground truth. Accuracy scores of these models are shown in Appendix F. Finally, we train a `Llama-3.1-8B-Instruct` model using the C-RLFT method (Wang et al., 2023). We use the `gpt-4-turbo` tag during inference because `gpt-4-turbo` performs best in the previous annotation process. Our model finally achieves 88.3% accuracy against both annotators.

[3]Evaluation models used in ToolBench and StableToolBench are `gpt-3.5-turbo` and `gpt-4-turbo`, both legacy models now

Method	I1 Inst	I1 Cat	I1 Tool	I2 Cat	I2 Inst	I3 Inst	Average
ToolLLaMA v2 CoT	28.0±1.9	30.5±0.8	21.5±0.9	19.9±1.0	22.3±0.4	19.1±0.8	22.8±0.8
ToolLLaMA v2 DFS	28.4±0.9	32.5±0.8	22.2±1.0	22.8±1.5	19.2±1.6	18.6±1.5	22.9±1.4
GPT 4o mini CoT	27.8±1.4	34.9±0.3	34.2±0.5	24.5±1.0	22.3±2.7	20.8±1.5	25.9±1.7
GPT 4o mini DFS	26.8±1.4	36.4±1.6	33.1±1.1	25.8±1.7	25.8±2.7	20.2±0.8	26.4±1.6
GPT 4o CoT	**33.3**±2.0	35.1±0.6	33.6±0.8	32.5±1.7	**29.6**±1.6	**27.9**±3.5	**32.0**±2.2
GPT 4o DFS	32.7±1.9	**42.3**±1.3	**34.6**±1.3	**32.8**±1.5	28.3±1.3	23.0±1.3	30.9±1.7

Table 4: Solvable pass rate scores. "Inst" and "Cat" stand for the Instruction and Category subsets. Tables below use the same abbreviation.

Method	I1 Inst	I1 Cat	I1 Tool	I2 Cat	I2 Inst	I3 Inst	Average
ToolLLaMA v2 CoT	45.4	38.6	34.2	40.3	37.7	31.1	37.9
ToolLLaMA v2 DFS	**47.9**	40.5	31.0	40.3	34.0	31.1	37.5
GPT 4o mini CoT	42.3	39.9	38.0	44.4	36.8	**36.1**	39.6
GPT 4o mini DFS	46.0	43.8	44.3	41.1	34.9	34.4	40.8
GPT 4o CoT	45.4	43.8	44.3	**54.0**	**45.3**	32.8	44.3
GPT 4o DFS	46.6	**53.6**	**44.9**	50.0	42.5	34.4	**45.3**

Table 5: FAC scores for different methods and conditions.

4.3 Compare Real and Simulated Environments

To compare the real and simulated environments, we first need to generate a set of queries. It is important to note that we cannot directly use the queries from the StableToolBench test set, as many of the ground-truth APIs are no longer available. In contrast, the simulated APIs are always accessible, which would introduce a bias between the two environments. To create new queries, we follow the approach used in ToolBench but without limiting the group or categories of APIs. Specifically, we sample 2-5 APIs from the entire set of available APIs and prompt `gpt-4o` to generate queries based on these APIs. The prompt is provided in Table 14. We then filter out unsolvable queries following the StableToolBench methodology. Next, we run `gpt-4o` and `gpt-4o-mini` using the CoT and Depth-First-Search (DFS) methods on these queries in the real environment. We ask `gpt-4o` to assess whether all APIs are functioning correctly in the response trajectory, using the prompt shown in Table 15. Any queries involving unavailable tools are filtered out. After this filtering process, 158 queries remain.

We run `gpt-4o`, `gpt-4o-mini`, and `ToolLLaMA -v2` using both CoT and DFS methods on these queries. The evaluation compares across three environments: the real environment, the environment simulated by `gpt-4o`, and the environment simulated by our `MirrorAPI`. The results are presented in Figure 3. As shown in the figures, our `MirrorAPI` closely matches the real environment across all tool-using methods, while the performance of `gpt-4o` in simulated environments is significantly lower. This demonstrates that `MirrorAPI` better simulates the tool environment compared to `gpt-4o`.

4.4 StableToolBench Performance with `MirrorAPI-Cache`

As an improved version for StableToolBench environments, we evaluate baseline model performance with `MirrorAPI`-Cache simulated environments.

First, to directly compare simulation outputs with real API calls, we evaluated the model on StableToolBench cache data containing actual calls from the test set. We randomly sampled 200 instances from the cache test set, stated in Section 4.1. The results demonstrate strong performance, with a BLEU-4 score of 84.9 and an observation and instruction following score of 8.82. In contrast, the general `MirrorAPI` achieved significantly lower scores of 27.1 and 4.76 respectively on the same metrics. This substantial improvement demonstrates the effectiveness of the finetuning approach on real-world API interaction data.

Then, we run `ToolLLaMA-v2`, `GPT-4o-mini`, and `GPT-4o` using both CoT and DFS inference methods on the simulated environments. The `single step max length` is set to 20, and the `max observation length` is set to 2048. The results are shown in Table 4 and Table 5. As observed in the tables, DFS methods generally outperform CoT methods, which aligns with the findings in ToolBench. Additionally, the latest `GPT-4o` model

outperforms the other models, as expected. However, the performance gap between DFS and CoT methods is narrower than anticipated. This is likely due to the reduced number of failing APIs during the inference stages, making the retry mechanism in DFS less advantageous. Moreover, the FAC scores are higher than the pass rate scores, which is consistent with expectations. Despite this, even `GPT-4o DFS` achieves only 45.3, which remains far from satisfactory.

5 Analysis

5.1 Ablation Studies

To evaluate the effectiveness of training in CoT mode, we compare several models on the OOD Succ test set using the observation-following metric. The models tested include: `MirrorAPI`, `MirrorAPI` without CoT training, `MirrorAPI` without augmented and CoT data, and `MirrorAPI` trained sequentially with SFT and CoT. Note that no additional data was created specifically for CoT compared to SFT. For inference, we use the direct SFT prompt for all models except the one trained sequentially with CoT, which uses the CoT prompt to achieve better performance.

The results are presented in Table 6. Incorporating augmented and CoT training data enhances model performance. Interestingly, generating outputs using the SFT prompt outperforms the CoT prompt, as detailed in Section 3.3. However, omitting CoT data during training negatively impacts performance. We hypothesize that adding CoT data lowers the training difficulty, as models may learn to reason implicitly during inference. This remains an open area for future research. Furthermore, the CoT training data is much smaller in size compared to the SFT data, which makes inference with CoT more challenging. Last, while sequential training with SFT and CoT yields worse performance than mixed training, it still outperforms training without CoT, supporting our hypothesis.

5.2 Cache Model on General Tasks

While `MirrorAPI-Cache` achieves satisfactory results on the StableToolBench test set, it is important to note that this post-training process does not degrade performance on OOD successful tasks. Specifically, the observation following metrics for CoT and SFT are 6.13 and 6.81, respectively. These results suggest that `MirrorAPI` has learned a robust representation that generalizes well across the benchmark through the diverse training datasets used. In this context, post-training fine-tunes the model on more task-specific features, but these adjustments do not significantly alter the model's core capabilities, allowing it to maintain strong performance across a variety of tasks.

Method	Score
`MirrorAPI`	6.86
w/ Seq CoT	6.33
w/o CoT	6.70
w/o A+CoT	6.26

Table 6: Ablation on the CoT Training on the OOD Succ test set. A and CoT represent data augmentation and CoT training data respectively. Seq CoT means the model is trained on SFT data and then CoT data.

5.3 Finetuning Reasoning Model

It is intriguing to examine the performance of fine-tuned reasoning models, especially given that test-time scaling has shown remarkable results on complex tasks (DeepSeek-AI, 2025). After fine-tuning the Deepseek-R1-Distill-Qwen model, we observed that its documentation and instruction following score on OOD successful tasks is significantly lower than that of the `MirrorAPI` and the o1-preview, with scores of 5.54 and 5.55 for CoT and SFT, respectively. Two potential explanations for this underperformance are as follows. First, the distilled models are heavily tuned for reasoning tasks, which may limit their ability in new domains. Second, the reasoning required for generating API responses can be subtle or implicit, leading to inefficiencies. In contrast, the normal LLM can more effectively handle tasks without unnecessarily forcing a reasoning process, which may explain its better performance in these scenarios.

6 Related Work

Tool Learning Benchmarks. LLMs augmented with external tools have demonstrated remarkable problem-solving capabilities, often exceeding their stand-alone performance (Li et al., 2023; Patil et al., 2023; Yang et al., 2023; Song et al., 2023; Tang et al., 2023; Ye et al., 2024; Xu et al., 2023a). Consequently, numerous benchmarks have been developed to evaluate proficiency in various aspects of tool learning, including tool selection (Wu et al., 2024; Xu et al., 2023b), task planning (Huang et al., 2024b; Shen et al., 2024), API stability (Guo et al., 2024) and query formulation (Shen et al., 2025).

Tool Environment Simulation. Prior studies (Lù et al., 2024; Yao et al., 2023; Rawles et al., 2024) have explored various real-world tool environment. Recent work has shifted toward simulated environments, such as web simulators (Shi et al., 2017; Yao et al., 2023; Zhou et al., 2024; Furuta et al., 2024; Chae et al., 2024), tool simulators for safety scenarios (Ruan et al., 2024), and multi-modal environment simulation (Zheng et al., 2024). However, existing approaches struggle to balance stability, scale, and realism. To address this, we propose a novel framework for training specialized LLMs to simulate tool environments effectively.

7 Conclusion

In this work, we present `MirrorAPI`, a novel framework that trains specialized LLMs to act as "mirrors" to tool environments by accurately replicating real API responses. The simulator is trained on request-response data pairs from over 7,000 APIs sourced from RapidAPI, with performance enhanced through a novel API mechanism reasoning framework. Experimental results demonstrate that `MirrorAPI` achieves strong documentation and instruction following capability. Responses provided by it are also the most similar to the real environment compared to baseline prompting methods. Furthermore, when deployed as an environment model in StableToolBench, `MirrorAPI` delivers reliable outcomes comparable to those obtained from real-world APIs, validating its practical utility.

Beyond the immediate scope of this work, general tool simulators may be beneficial to advancing LLMs' tool-using capabilities. First, such simulators can enable execution feedback during training or inference without costly real-environment interactions, paving the way for novel online training algorithms and adaptive inference strategies (Wang et al., 2024a; Renze, 2024; Yu et al., 2025). Second, they facilitate the generation of diverse training trajectories through the integration of synthetic APIs, addressing data scarcity challenges in tool-learning scenarios. Finally, while our current focus lies in simulating functional and reliable tools, the framework can be extended to simulate tools with operational failures, safety-critical vulnerabilities, or ethical concerns. This capability could significantly expand the scope of tool-learning research by enabling risk-aware training and robustness testing in controlled environments.

Acknowledgement

This work is supported by the National Key R&D Program of China (2022ZD0160502). We also extend our gratitude to Peng Li for his assistance with valuable suggestions.

Limitations

In this work, we propose a trained tool simulator specialized in mirroring real tool environments. However, our approach has certain limitations. Firstly, real API requests often fail in practical environments due to issues such as connectivity problems. While it is crucial for a tool-using model to effectively handle such failures, this aspect is not addressed in our project. Nevertheless, our system can be easily adapted to simulate these scenarios by modifying the system prompts or intentionally introducing failures manually. Secondly, as a general simulator, `MirrorAPI` has the potential to provide immediate feedback to tool-using models, thereby enhancing their performance. This capability could also be leveraged to evaluate the effectiveness of the models in greater detail. However, due to time and resource constraints, we have not conducted experiments in this direction, leaving it as an avenue for future work.

References

AI@Meta. 2024. Llama 3 model card.

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Vernon Austel, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis Lastras. 2024. API-BLEND: A comprehensive corpora for training and benchmarking API LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12859–12870, Bangkok, Thailand. Association for Computational Linguistics.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Hyungjoo Chae, Namyoung Kim, Kai Tzu iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. 2024. Web agents with world models: Learning and leveraging environment dynamics in web navigation. *Preprint*, arXiv:2410.13232.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng

Zhao. 2024. T-eval: Evaluating the tool utilization capability of large language models step by step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9510–9529, Bangkok, Thailand. Association for Computational Linguistics.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Nicholas Farn and Richard Shin. 2023. Tooltalk: Evaluating tool-usage in a conversation setting. *arXiv preprint arXiv:2311.10775*.

Hiroki Furuta, Yutaka Matsuo, Aleksandra Faust, and Izzeddin Gur. 2024. Exposing limitations of language model agents in sequential-task compositions on the web. *Preprint*, arXiv:2311.18751.

Gemini Team. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11143–11156, Bangkok, Thailand. Association for Computational Linguistics.

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. *Preprint*, arXiv:2401.17167.

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. 2024b. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *Preprint*, arXiv:2401.17167.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024c. Understanding the planning of llm agents: A survey. *Preprint*, arXiv:2402.02716.

Mojtaba Komeili, Kurt Shuster, and Jason Weston. 2022. Internet-augmented dialogue generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8460–8478, Dublin, Ireland. Association for Computational Linguistics.

Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Benchmark for Tool-Augmented LLMs. *Preprint*, arXiv:2304.08244.

Xing Han Lù, Zdeněk Kasner, and Siva Reddy. 2024. Weblinx: Real-world website navigation with multi-turn dialogue. *Preprint*, arXiv:2402.05930.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, and Aixin Sun. 2024. SciAgent: Tool-augmented language models for scientific reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15701–15736, Miami, Florida, USA. Association for Computational Linguistics.

OpenAI. 2024a. Gpt-4o system card. Accessed: 2025-02-16.

OpenAI. 2024b. Introducing openai o1-preview. Accessed: 2025-02-16.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. *ArXiv preprint*, abs/2305.15334.

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. 2024. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

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

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-rong Wen. 2025. Tool learning with large language models: a survey. *Frontiers of Computer Science*, 19(8).

Qwen Team. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, Miami, Florida, USA. Association for Computational Linguistics.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2024. Identifying the risks of lm agents with an lm-emulated sandbox. *Preprint*, arXiv:2309.15817.

Haiyang Shen, Yue Li, Desong Meng, Dongqi Cai, Sheng Qi, Li Zhang, Mengwei Xu, and Yun Ma. 2025. Shortcutsbench: A large-scale real-world benchmark for api-based agents. *Preprint*, arXiv:2407.00132.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. *Preprint*, arXiv:2311.18760.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. 2017. World of bits: An open-domain platform for web-based agents. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144. PMLR.

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. *ArXiv preprint*, abs/2306.06624.

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. *Preprint*, arXiv:2306.05301.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. 2024a. LLMs in the imaginarium: Tool learning through simulated trial and error. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10583–10604, Bangkok, Thailand. Association for Computational Linguistics.

Chenyu Wang, Weixin Luo, Qianyu Chen, Haonan Mai, Jindi Guo, Sixun Dong, Xiaohua, Xuan, Zhengxin Li, Lin Ma, and Shenghua Gao. 2024b. Mllm-tool: A multimodal large language model for tool agent learning. *Preprint*, arXiv:2401.10727.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023. Openchat: Advancing open-source language models with mixed-quality data. *arXiv preprint arXiv:2309.11235*.

Mengsong Wu, Tong Zhu, Han Han, Chuanyuan Tan, Xiang Zhang, and Wenliang Chen. 2024. Seal-tools: Self-instruct tool learning dataset for agent tuning and detailed benchmark. *Preprint*, arXiv:2405.08355.

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. *Preprint*, arXiv:2305.16504.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023b. On the tool manipulation capability of open-source large language models. *Preprint*, arXiv:2305.16504.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. GPT4Tools: Teaching Large Language Model to Use Tools via Self-instruction. *Preprint*, arXiv:2305.18752.

Zonghan Yang, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. 2024. React meets actre: When language agents enjoy training data autonomy. *Preprint*, arXiv:2403.14589.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2023. Webshop: Towards scalable real-world web interaction with grounded language agents. *Preprint*, arXiv:2207.01206.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. ToolEyes: Fine-Grained Evaluation for Tool Learning Capabilities of Large Language Models in Real-world Scenarios. *Preprint*, arXiv:2401.00741.

Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. 2025. Exact: Teaching ai agents to explore with reflective-mcts and exploratory learning. *Preprint*, arXiv:2410.02052.

Tianyi Zhang*, Varsha Kishore*, Felix Wu*, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Preprint*, arXiv:2306.05685.

Longtao Zheng, Zhiyuan Huang, Zhenghai Xue, Xinrun Wang, Bo An, and Shuicheng Yan. 2024. Agentstudio: A toolkit for building general virtual agents. *Preprint*, arXiv:2403.17918.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

A Request writing

An example of request writing is shown in Table 7

API Documentation
API Name: List Languages **API Desc.:** Get a list of currently supported languages. **Required Param:** N/A **Optional Param:** N/A **Tool Desc.:** This is a tool used to ... **Tool Name:** TTSKraken **Tool Category:** Artificial Intelligence
Scenario
You are a content creator tasked with producing an engaging educational video for young learners that helps them understand basic math concepts in multiple languages. To make the video more accessible, you want to incorporate voiceovers in various languages. First, you need to gather a list of the languages supported by the TTSKraken API. Once you have the language list, you will choose three different languages that can target international audiences. Next, you will convert your script, which explains addition and subtraction in a fun way, into audio using the TTSKraken service in those selected languages. Finally, you will integrate these audio files into your video, aiming to enhance the learning experience for children from different linguistic backgrounds.,
LLM-written request

`“category”: “Artificial_Intelligence”,`
`“tool_name”: “TTSKraken”,`
`“api_name”: “List Languages”,`
`“tool_input”: “{}”`

Table 7: An example of API documentation, scenario and LLM-written request.

B Distribution of Training Data

The distribution of the training set is shown in Figure 4.

C Prompts

D Request Error Identification and Request Filtering Rule

In this study, we classify request errors and filter out invalid requests to RapidAPI based on specific keyword occurrences within the error or response messages. The errors are categorized as follows:

- Not Connected Error: This error is identified when the error message includes terms such as `HTTP`, or when the response contains phrases like `HTTP error`, `connection`, `rate limit` or `timeout`.
- Not Found Error: This occurs when the error or response message includes terms such as `not found`, `not available`, `API doesn't exists`, `Service Not Found`, `internal error` or a 404 error message;
- Parameter Change: This category is triggered when the error message or response refers to issues with parameters, parsing, or undefined variables.
- Parsing Error: This error is detected when the error message begins with the phrase `Function executing from`.
- Not Authorised: Errors falling under this category are identified by terms such as `authorize`, `unauthorized`, `blocked user`, `unsubscribe`, `credential`, `disabled for your subscription`, or any mention of 401 or 403 error codes.
- Other Errors: This category includes errors that contain any non-empty error message not falling into the aforementioned categories.
- Success: All other requests that do not result in errors are classified as successful.

E Training Hyperparameters

We trained the `MirrorAPI` with the following hyperparameter settings: a learning rate of 2e-05, a train batch size per device of 2. The training process used a seed value of 42, with a multi-GPU distributed setup across 8 devices, each equipped with 40GB A100 GPUs. Gradient accumulation was applied with 8 steps, leading to a total effective train batch size of 128. The optimizer used was Adam, with betas set to (0.9, 0.999) and epsilon set to 1e-08. For the learning rate schedule, a cosine annealing approach was employed, with a warmup ratio of 0.04 and 100 warmup steps. The model was trained for a total of 5 epochs.

F Human Annotation on the FAC evaluator

Results of human annotation and model performance of the FAC evaluator are shown in Table 16.

Prompt for Checking Documentation Following	
System	We have a bunch of APIs at hand but their documentations may be out of date. Please help us evaluate the quality of an API response by assessing its adherence to API documentation. You will be given the API documentation, containing the API name, description, tool name, tool description, required parameters, and optional parameters. (A tool may contain several APIs and has an overall functionality) You will also receive a user request (in a JSON format) and the API response (in the format of "error": "error message if there is any, can be empty", "response": "the response content. may be empty if there is an error."). # Goals: Your primary goal is to determine how well the given API response adheres to the provided API documentation. # Notes: - If the user request is malformed, a correct response may include an appropriate error message. This is to say if a user request is malformed and the response points it out, then the response is a good one. - You need to only judge the API response, not the user request. - As long as the response is a possible valid response following the documentation, it should be considered correct. - Your analysis should remain neutral with an emphasis on objective evaluation. # Output: Provide a detailed evaluation that reflects adherence to the above criteria, clearly identifying any deviations from the expected results. Respond in the following JSON format: {"overall_eval":1/0 (1 means meet the goal and 0 otherwise), "reason":"xxx"}
User	# API Documentation: API Name: {api_name} API Description: {api_description} Tool (the API belongs to) Name: {tool_name} Tool (the API belongs to) Description: {tool_description} API Required Parameters: {required_parameters} API Optional Parameters: {optional_parameters} # User Request: {user_request} # API Response: {api_response}

Table 8: Prompt for checking documentation following.

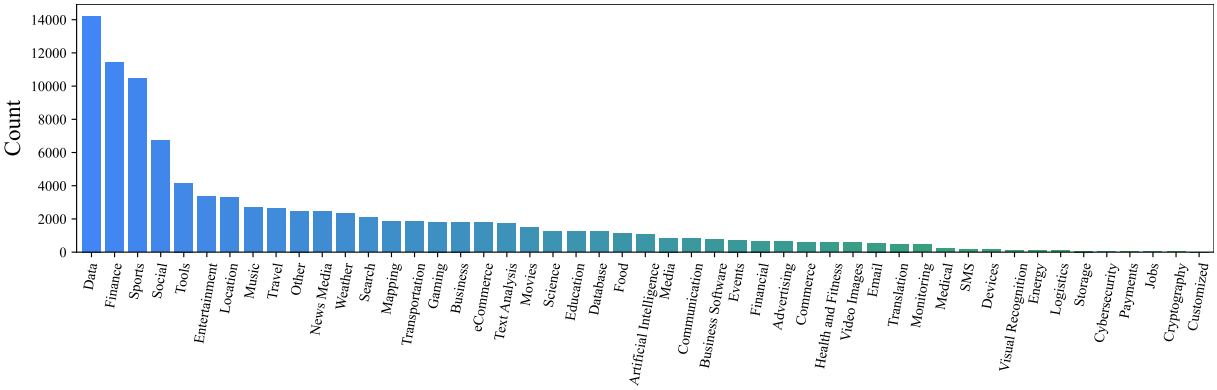


Figure 4: Data count in each tool category

API Simulation Prompt	
User	I have an API and call this API with a specific request. The API returned a response. Can you give some hints about the mechanism behind the API, i.e. how the API works? This will be used to help an API simulator that behaves like the API you are reasoning about. Note that the simulator will not be able to see the real response so your hint should be instructive and constructive but not leak any specific information about the real response. You will be given the API's documentation, parameters specification and the response returned by the API. Make sure to limit your reasoning in 300 words. Return only your reasoning of how the API works, not the response. API Name: `{api_name}` API Description: `{api_description}` Required Parameters: `{required_parameters}` Optional Parameters: `{optional_parameters}` Tool (The API belongs to) Name: `{tool_name}` Tool Description: `{tool_description}` API Request: `{api_request}` API Response: `{api_response}`

Table 9: Rationale generation prompt.

SFT Mode System Prompt	
System	Imagine you are an API Server operating within a specialized tool, which contains a collection of distinct APIs. Your role is to deeply understand the function of each API based on their descriptions in the API documentation. As you receive specific inputs for individual API calls within this tool, analyze these inputs to determine their intended purpose. Your task is to craft a JSON formatted response that aligns with the expected output of the API. The JSON scheme is: `{ "error": "", "response": "" }` The error field should remain empty, indicating no errors in processing. The response field should contain the content you formulate based on the API's functionality and the input provided. Ensure that your responses are meaningful, directly addressing the API's intended functionality. The key is to maintain the JSON format's integrity while ensuring that your response is an accurate reflection of the API's intended output within the tool. Please note that your answer should not contain anything other than a json format object, which should be parsable directly to json. Note that: - your response should contain rich information given the api input parameters. - your response must be effective and have practical content. API calls may fail for various reasons, such as invalid input parameters, authentication issues, or server errors. Your goal is to generate a response that accurately reflects the API's intended functionality, even if the input parameters are incorrect. Your response should be informative and relevant to the API's purpose, providing a clear and concise explanation of the expected output based on the input provided. Here is an example: API doc: `Example API Documentation` Request: `Example Request` Response: `Example Response`

User	API doc:
	`{api_doc}`
	Request:
	`{request}`

Table 10: Supervised fine-tuning system prompt.

CoT Mode System Prompt	
System	[CHAIN_OF_THOUGHT] You are an API Server operating within a specialized tool, tasked with understanding the purpose of each API based on provided documentation. Your job is to process specific API inputs and craft a well-formatted response reflecting the API's intended functionality. You should first infer the mechanism behind the API and then provide your response based on the input parameters. Your response must follow this JSON structure: `{ "mechanism_of_the_api": "", "error": "", "response": "" }` * MECHANISIM OF THE API: Try to infer how the API functions based on the input parameters. * ERROR: Leave empty unless there's an issue with the input. * RESPONSE: Provide content based on the API's function. If examples are ineffective, give an independent, meaningful response. Note: * Ensure responses are practical, clear, and relevant. * Handle incorrect input gracefully by explaining expected behavior. Here is an example: API doc: `Example API Documentation` Request: `Example Request` Response: `Example Response`

User	API doc:
	`{api_doc}`
	Request:
	`{request}`

Table 11: Chain-of-Thought system prompt.

Prompt Used In LLM-as-a-Judge	
System	You are a helpful assistant.

User	[Instruction]
	Act as an impartial judge to evaluate the quality of an AI API simulation output based on the provided API documentation and user request. Assess the simulation's accuracy in adhering to the documentation and fulfilling the user request. You will receive both a reference answer, representing a real API response, and the simulator's answer. The simulator's response does not need to match the reference answer exactly but must be faithful to the documentation and user request. The reference answer is just one possible output, but not the only one (You should not judge how similar the simulator's output is to the real one). The most important factor is whether the response is consistent with the API documentation and the user request. Pay attention to both the structure and the content of the response. Note that the response does not need to include all (even key) information in the documentation and the user request. As long as it is a reasonable response from the API, it should be rated as 10. Begin your evaluation by comparing the simulator's response with the documentation and user request. Identify and correct any mistakes. Be as objective as possible. After providing your explanation, you must rate the response on a scale of 0 to 10 by strictly following this format: "[[rating]]", for example: "Rating: [[5]]".
	[Question]
	`{question}`
	[The Start of Reference Answer]
	`{ref_answer_1}`
	[The End of Reference Answer]
	[The Start of Assistant's Answer]
	`{answer}`
	[The End of Assistant's Answer]

Table 12: LLM Judge Prompt in documentation and instruction following.

FAC Scoring Prompt	
User	Given a query and an answer provided by an AI agent, you now need to determine the answer_status of whether the well solved the query, i.e. whether the need of the query is satisfied. You need to output "Unsolved" or "Solved" and your reason. You must obey the following rules: You should response "Solved" when: 1. If the answer well provides the information needed by the query, then it is "Solved". The answer does not need to be perfect, and it only needs to make a genuine attempt to address the query. 2. Consider only Completeness: The answer attempts to address every part of the query, regardless of whether the information provided is factually correct or accurate, unless there is a severe factual error. 3. For Multi-part Queries: For queries with multiple parts, all parts must be addressed for the answer to be considered "Solved". 4. Genuine Attempt : The answer makes a genuine attempt to provide the requested information or perform the requested task for all parts of the query. This includes scenarios where the answer concludes that "nothing" is a reasonable response (e.g., when the requested information does not exist or is not available, or a possible answer of the query is nothing and the model answers nothing after reasonable attempts). You should response "Unsolved" when: 1. Refusal, Apology, or Non-engagement: The answer includes a refusal or apology (e.g., "I'm sorry, I can't help with that"). The answer does not directly engage with or address the query in any way. 2. Multi-part Queries: If the query has multiple parts and at least one part is not well addressed. 3. Severe Factual Error: If the answer contains a severe factual error that significantly impacts the usefulness of the information provided. Additional Guidelines: 1. VERY IMPORTANT: DO NOT BE TOO HARSH. The model does not need to be perfect, and the answer does not need to be flawless. It only needs to make a genuine attempt to address the query. 2. DO NOT evaluate factual accuracy or correctness of the information provided based on your knowledge. Assume that the information provided is accurate and focus solely on whether the answer attempts to address all parts of the query, unless there is a severe factual error that conflicts common knowledge. 3. Focus on Final Answer: Only the final answer is provided and should be considered, disregarding any processes that were used to generate the answer. You only need to judge whether the information need is satisfied. 4. Answer Completion: The agent does not need to detail how it arrived at the answer, only that the answer itself is complete and attempts to address the query. Here are some examples: xxxx Now give your reason and answer status in the following format: Answer Status xxx (can only be "Solved" or "Unsolved") Reason xxx

Table 13: Prompt used in the FAC score.

Query Generation Prompt	

User	You will be provided with several tools, tool descriptions, all of each tool's available API functions, the descriptions of these API functions, and the parameters required for each API function. Your task involves creating a varied, innovative, and detailed user query that employ API functions of multiple tools. For instance, given three tools 'nba_news' 'cat-facts' 'hotels': 'nba_news' has API functions "Get individual NBA source news" and "Get all NBA news", 'cat-facts' has API functions "Get all facts about cat" and "Get a random fact about cats", 'hotels' has API functions "properties/get-details (Deprecated)", "properties/list (Deprecated)" and "locations/v3/search". Your query should articulate something akin to: "I want to name my newborn cat after Kobe and host a party to celebrate its birth. Get me some cat facts and nba news to gather inspirations for the cat name. Also, find a proper hotel around my house in Houston Downtown for the party." This query exemplifies how to utilize API calls of all the given tools. A query that uses API calls of only one tool will not be accepted. Additionally, you must incorporate the input parameters required for each API call. To achieve this, generate random information for required parameters such as IP address, location, coordinates, etc. For instance, don't merely say 'an address', but provide the exact road and district names. Don't just mention 'a product', but specify wearables, milk, a blue blanket, a pan, etc. Don't refer to 'my company', but invent a company name instead. The first seven of the ten queries should be very specific.
	The query should combine API calls of different tools in various ways and include the necessary parameters. Note that you shouldn't ask 'which API to use', rather, simply state your needs that can be addressed by these APIs. You should also avoid asking for the input parameters required by the API call, but instead directly provide the parameters in your query.
	The final query should be complex and lengthy, describing a complicated scenario where all the provided API calls can be utilized to provide assistance within a single query.
	You should first think about possible related API combinations, then give your query. Related_apis are apis that can be used for a given query; those related apis have to strictly come from the provided api names. For each query, there should be multiple related_apis.
	Deliver your response in this JSON format:
	`{ "query":..., "related_apis":[[<tool name>, <api name>], [<tool name>, <api name>], [<tool name>, <api name>]] }`
	Examples:
	`{Example}`
	These are only examples to show you how to write the query. Do not use apis listed in the above examples, but rather, use the ones listed below in the INPUT.
	INPUT:
	`{tools_json}`
	OUTPUT:

Table 14: Prompt for generating queries.

Calls Writing Prompt	
User	I have an API at my hand and I want to use this API to solve a problem. Can you write a call to test if the API is reachable? Remember to include all the required parameters. You must follow the API documentation to write your call. Do not make up any parameters Please also note that a tool is a collection of APIs that are used to solve a specific problem. Your answer should be in the following json format: {{ "category":"", "tool_name":"", "api_name":"", "tool_input":'{{}}', "strip":"filter", }} Tool Documentation: {document}

Table 15: Prompt for writing tool calls.

R1 and R2 stand for two annotators.

Method	R1 Acc	R2 Acc
R1	–	92.2
R2	92.2	–
GPT 4o mini	79.4	82.3
GPT 4o	88.2	88.2
GPT 4 Turbo	**97.2**	**91.2**
Ours	88.3	88.3

Table 16: Evaluation model results

G Model Performance on StableToolBench with `MirrorAPI-Cache`

Performance of the models that are used in ToolBench is shown in Table 17 and Table 18. We use the replication data released by ToolBench so these models are run in real environments but evaluated with `gpt-4o` now in the Pass Rate metric. Note that these scores are much lower than those reported by ToolBench, probably because `gpt-4o` is much more strict than `gpt-3.5-turbo`. But the relative ranking of models keeps the same. Please also note that there are a lot of failing tools in the replication making the performance lower than our simulated environments. An example is as follows in ToolLLaMA v2 CoT:

- Step 4:
  - Name: `cifications_by_custom_id`
  - Arguments:

```
{
  "phonecustomid": 123456
}
```

  - Response:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>An Error Occurred:
        Internal Server Error</
        title>
        <style>xxx</style>
        </head>
        <body>
        <div class="container">
        <h1>Oops! An Error
            Occurred</h1>
        <h2>The server returned
            a "500␣Internal␣
            Server␣Error".</h2>
        <p>
        Something is broken.
            Please let us know
            what you were doing
            when this error
            occurred.
        We will fix it as soon
            as possible. Sorry
            for any
            inconvenience caused
            .
        </p>
        </div>
        </body>
</html>
```

H Typical Errors of MirrorAPI

There are several factors that the model performs less well than the realness:

Method	I1 Inst	I1 Cat	I1 Tool	I2 Cat	I2 Inst	I3 Inst	Average
GPT 4 0613 CoT (Rep)	18.8±0.3	24.6±0.3	16.2±1.6	13.7±1.1	16.0±1.5	2.2±0.8	14.3±1.0
GPT 4 0613 DFS (Rep)	18.4±1.3	29.8±1.3	25.7±1.2	20.2±1.3	16.4±0.4	2.2±0.8	18.5±0.9
ToolLLaMA CoT (Rep)	5.1±1.0	16.6±0.8	8.0±1.3	9.7±1.1	7.5±0.8	3.3±0.0	8.4±0.9
ToolLLaMA DFS (Rep)	7.4±0.5	16.1±0.6	12.4±0.3	9.9±0.8	9.1±0.4	0±0.0	8.2±0.6

Table 17: Solvable pass rate scores. "Inst" and "Cat" stand for the Instruction and Category subsets. "Rep" stands for the official replication data from ToolBench.

Method	G1 Inst	G1 Cat	G1 Tool	G2 Cat	G2 Inst	G3 Inst	Average
GPT 4 0613 CoT (Rep)	25.2	30.1	22.2	25.8	26.4	3.3	22.1
GPT 4 0613 DFS (Rep)	27.6	35.3	32.3	32.3	26.4	4.9	26.5
ToolLLaMA CoT (Rep)	0.0	19.0	11.4	15.3	8.5	1.6	9.3
ToolLLaMA DFS (Rep)	0.0	19.0	13.3	19.4	13.2	0.0	10.8

Table 18: FAC scores for replication models. "Inst" and "Cat" stand for the Instruction and Category subsets.

H.1 Limited Internal Knowledge

The model may only have partial information of an API's mechanisms without specific knowledge. An example is shown in Table 19.

H.2 Non-Adherence to API Requests

The model sometimes overlooks the exact parameters requested. For example, if asked for a list of 30 "creeping creatures," it might return only one, or if asked for airports in California, it might provide information on airports in different locations. An example is shown in Table 20.

H.3 Hallucinations

The model can generate information that is simply untrue. A typical example, shown in Table 21, is when prompted about popular songs from 2010 and it invents songs or attributes songs to the wrong artists.

H.4 Simple or Empty Responses

Occasionally, the model provides minimal or no meaningful information, such as returning a sparse JSON string that adds little value, rather than a detailed answer. An example is shown in Table 22

I AI Usage

We acknowledge that we use LLMs to help with polishing words and tables in the papers.

API Documentation	Sample Request	Reference Response	Actual Output

{
"api_name": "Airports by Keyword",
"description": "Returns airport names and IATA codes matching keyword",
"required_params": [
{"name": "keyword",
"type": "string"}
],
"tool_info": {
"name": "CarbonSutra",
"category": "Science",
"purpose": "Carbon emission estimations"
}
}
	{
	"tool_name": "CarbonSutra",
	"api_call": "Airports by Keyword",
	"params": {
	"keyword": "Yosemite, Banff, Galapagos"
	}
	}
		{
		"error": "No airports found",
		"status": 422,
		"details": "Invalid keyword combination"
		}
			{
			"data": [],
			"success": true,
			"status": 200
			}

Table 19: API Response Comparison Demonstrating Limited Context Awareness

API Documentation	Sample Request	Reference Response	Actual Output

{
"api_name": "Get All Creeping Creatures",
"description": "Get All Creeping (crawl close to ground) Creatures",
"required_params": [],
"optional_params": [
{"name": "limit",
"type": "string",
"default": "10"}
],
"tool_info": {
"name": "Dietary Laws",
"category": "Food",
"purpose": "Clean and Unclean animals from the bible"
}
}
	{
	"tool_name": "Dietary Laws",
	"api_call": "Get All Creeping Creatures",
	"params": {
	"limit": "30"
	}
	}
		{
		"error": null,
		"data": [
		{"name": "Guinea Pig",
		"type": "unclean",
		"image_src": "xxx.jpg"},
		...(29 more creatures)
		],
		"status": 200
		}
			{
			"error": null,
			"data": [
			{"name": "Locust",
			"type": "clean",
			"image_src": "xxx.jpg"}
			],
			"status": 200
			}

Table 20: Example of Non-Adherence to API Requests

API Documentation	Sample Request	Reference Response	Actual Output

{
"api_name": "Top Songs by Year",
"description": "Returns Spotify's top songs for a given year",
"required_params": [
{"name": "year",
"type": "string"}
],
"tool_info": {
"name": "MusicData API",
"category": "Music",
"purpose": "Music streaming data analysis"
}
}
	{
	"tool_name": "MusicData API",
	"api_call": "Top Songs by Year",
	"params": {
	"year": "2010"
	}
	}
		{
		"data": [
		{"ranking": 1,
		"artist_and_title": "Bruno Mars - Just the Way You Are",
		"streams": "2,391,682,258",
		"daily": "2,553,215"},
		...(additional songs)
		],
		"status": 200
		}
			{
			"data": [
			{"ranking": 1,
			"artistAndTitle": "Eminem - Love The Way You Lie",
			"streams": "1,286,362,947"},
			{"ranking": 7,
			"artistAndTitle": "Eminem - Frankenstein",
			"streams": "1,039,249,209"},
			{"ranking": 15,
			"artistAndTitle": "Eminem - I Just Punch In The Clock",
			"streams": "898,098,457"},
			...(additional songs)
			],
			"status": 200
			}

Table 21: Example of Hallucination in API Responses

API Documentation	Sample Request	Reference Response	Actual Output

{
"api_name": "Airports by Keyword",
"description": "Returns airport names and IATA codes matching keyword",
"required_params": [
{"name": "keyword",
"type": "string"}
],
"tool_info": {
"name": "CarbonSutra",
"category": "Science",
"purpose": "Carbon emission estimations"
}
}
	{
	"tool_name": "CarbonSutra",
	"api_call": "Airports by Keyword",
	"params": {
	"keyword": "green"
	}
	}
		{
		"data": [
		{"iata_code": "BWG",
		"airport_name": "Bowling Green Warren County Regional Airport"},
		...(additional airports)
		],
		"success": true,
		"status": 200
		}
			{
			"data": [],
			"success": true,
			"status": 200
			}

Table 22: Example of Empty Response Error