# SEAL: Suite for Evaluating API-use of LLMs

**Woojeong Kim**
Cornell University
wk247@cornell.edu

**Ashish Jagmohan**
Emergence AI
ashish@emergence.ai

**Aditya Vempaty**
Emergence AI
aditya@emergence.ai

## Abstract

Large language models (LLMs) have limitations in handling tasks that require real-time access to external APIs. While several benchmarks like ToolBench and APIGen have been developed to assess LLMs' API-use capabilities, they often suffer from issues such as lack of generalizability, limited multi-step reasoning coverage, and instability due to real-time API fluctuations. In this paper, we introduce SEAL, an end-to-end testbed designed to evaluate LLMs in real-world API usage. SEAL standardizes existing benchmarks, integrates an agent system for testing API retrieval and planning, and addresses the instability of real-time APIs by introducing a GPT-4-powered API simulator with caching for deterministic evaluations. Our testbed provides a comprehensive evaluation pipeline that covers API retrieval, API calls, and final responses, offering a reliable framework for structured performance comparison in diverse real-world scenarios. We plan to open-source SEAL with ongoing updates for new benchmarks.

## 1 Introduction

While large language models (LLMs) excel at many language tasks, they face limitations when handling tasks that require real-time access to specific information, such as current events, calculations, or web searches. Tools like calculators, code execution, and browsing extend LLMs' capabilities, enabling them to perform specialized tasks and access up-to-date knowledge, adapting dynamically to users' needs. Recently, several benchmarks have been introduced to assess LLMs' ability to interact with real-world APIs, moving beyond a small set of hand-coded tools to a broader pool of practical, real-world applications. These include ToolBench [Qin et al., 2023] and variants thereof, APIGen [Chen et al., 2024], AnyTool [Du et al., 2024] and MetaTool [Huang et al., 2023].

In this paper, we analyze major API-use[1] benchmarks and argue that there are critical gaps. Specifically, we identify a number of common issues, including the lack of clear holdout sets leading to overfitting, poor coverage of multi-step reasoning queries which are essential in real-world use cases of such systems, and deficiencies in benchmark quality and stability. Further, some benchmarks only focus on certain aspects; For example, AnyTool [Du et al., 2024] and MetaTool [Huang et al., 2023] focus on tool selection but overlook other important aspects such as the content of tool calls and the final response. Similarly, APIGen [Chen et al., 2024] tests function-calling capabilities but neglects multiple possible trajectories with only one hard-coded answer.

In response, we present SEAL, a comprehensive, end-to-end testbed for evaluating LLMs in tool usage, particularly with diverse real-world APIs. This testbed provides uniform test environment of API calling system by sanitizing and standardizing existing benchmarks, incorporating an agent system built on AutoGen [Wu et al., 2023] for testing both API retrieval and planning, and providing a robust evaluation pipeline. Due to the real-time nature of APIs, existing benchmarks often lack reliable evaluation as there are no static ground truth answers. To address this, we developed an API

[1]We use "tool" and "API" interchangeably throughout this paper. While we primarily focus on APIs, the same methodology can be extended to general tool usage with appropriate API-wrapper around them.

Preprint. Under review.

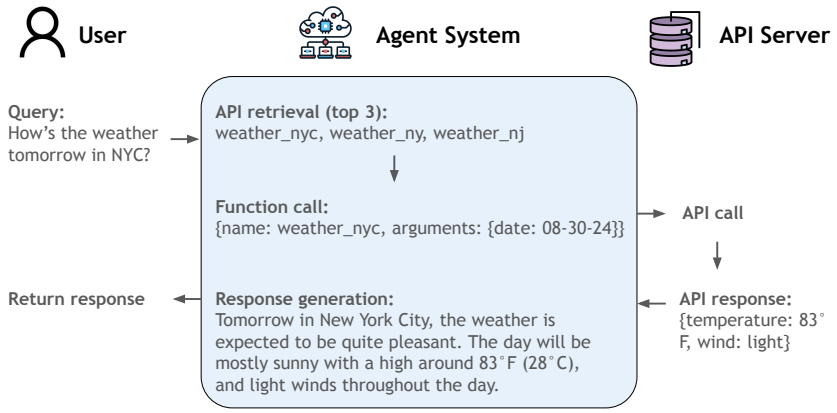



Figure 1: Workflow of a single-step, single-API-use system

Table 1: Comparison of tool-use benchmarks. For datasets with multiple subsets, we only report statistics for the largest subset: G1-train for Toolbench, and Huggingface for APIBench

| | Toolbench Qin et al. [2023] | APIGen [Chen et al., 2024] | AnyTool [Du et al., 2024] | MetaTool [Huang et al., 2023] | APIBench Patil et al. [2023] |
|---|---|---|---|---|---|
| API source | RapidAPI | RapidAPI | RapidAPI | OpenAI plugin | HuggingFace |
| Total # of queries | 40399 | 60000 | 407 | 21047 | 8191 |
| - multi-step only | 34052 | 20448 | 339 | 497 | 0 |
| Avg. # of APIs per query | 4.5 | 1.7 | 2.3 | 1.1 | 1.0 |
| # of APIs with queries | 8684 | 3179 | 307 | 199 | 914 |
| Total # of APIs | 16464 | 3605 | 16464 | 437 | 914 |

simulator powered by GPT-4 [Achiam et al., 2023] to generate plausible API responses, building on the approach in StableToolbench [Guo et al., 2024]. We further enhance this system with caching to enable more deterministic evaluations. Additionally, we offer a comprehensive evaluation framework that covers all aspects of API learning, including API retrieval, API calls, and the final response. Our testbed allows users to test their retrieval and planning methods while providing a more structured and reliable performance comparison.

Our contributions are as follows:

- We investigate the shortcomings of existing API-use benchmarks, identifying issues like lack of generalizability, limited multi-step reasoning coverage, and benchmark instability.
- We present SEAL, a comprehensive testbed that standardizes existing benchmarks and incorporates an agent-based system for evaluating LLMs in retrieval, planning, and execution.
- We offer an end-to-end evaluation framework covering API retrieval, API calls, and final responses, enabling structured performance comparison across diverse real-world scenarios.

## 2 Overall Landscape

Fig. 1 provides an overview of the typical workflow for tool-use systems. These systems consist of three main components: the user, the agent system, and the API server. The process begins when a user queries the agent system, for example, "How's the weather tomorrow in NYC?" The next step typically is a retrieval to identify relevant APIs for the language model that powers the agentic system. This is especially useful in scenarios with numerous real-world APIs, where providing all available APIs as input is impractical due to context limitations (such as length, and lost-in-the-middle phenomenon). The LLM then generates the arguments for tool calls based on the API documentation or the tool specifications and forwards them to the API server. Typically, the API server operates as a separate component outside the agent system. Once the API response is received, the agent system's final role is to generate a response summarizing the API's output. While this example involves a single-step process, in many real-world cases, these steps are repeated, with each step's

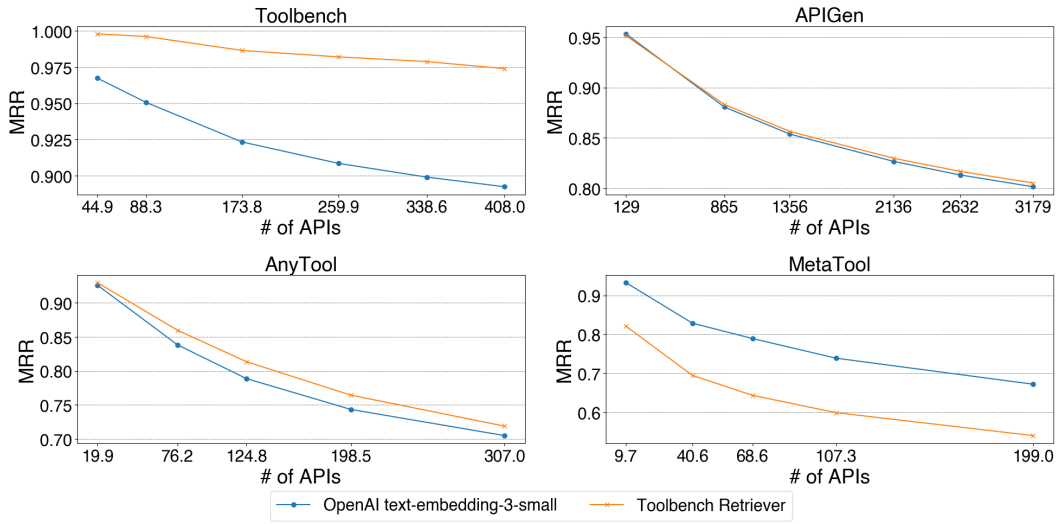



Figure 2: Comparison of embedding models' API retrieval performance across benchmarks. We fixed the total number of sampled queries and report the average performance over 10 sampling runs. For ToolBench, results are based on the test split, as the ToolBench retriever was trained on the train split.

input determined by the output of the previous step. There may also be an optional verification step to ensure intermediate and final responses are generated correctly.

This is a multi-step, complex process that relies on several LLM capabilities, including accurate retrieval, correct function calling, and factual summarization. However, existing benchmarks focus only on specific aspects of this pipeline: AnyTool and MetaTool emphasize tool selection, while APIGen focuses on function-calling capabilities. Section 3 talks about these challenges in more detail.

Another often-overlooked aspect is that this process involves collaboration between multiple agents, such as the tool retriever, the main orchestrator, and optionally, a planner or verifier. [Mekala et al., 2024, Shen et al., 2024] Most previous benchmarks [Chen et al., 2024, Qin et al., 2023, Patil et al., 2023] customize a single LLM to handle all these steps. We argue that this could be tackled more effectively using an agent system, where certain components are powered by an LLM. In fact, this separation is already happening internally, though it's not often recognized as distinct agent components. For example, tool retrieval is often handled independently of the LLM, using query/API embeddings generated by large embedding models or smaller sentence-transformer models, followed by vector lookups. Our testbed integrates these components under the concept of agents, and we implement it using the AutoGen [Wu et al., 2023] framework. Section 4 discusses this in more detail.

## 3 Challenges in Existing Benchmarks

We conduct an in-depth evaluation of existing benchmarks and identified several critical limitations that restrict their wider applicability. Table 1 presents key statistics for five benchmarks we focus on. While these are the primary benchmarks under discussion, there are additional ones within the scope that we plan to further support. Our analysis highlights the following four significant limitations.

### 3.1 Lack of Generalizability

Previous works have claimed that small, open-source models trained on specific benchmarks can perform as well as, or better than, general-purpose models. However, as noted by Kapoor et al. [2024], many agent benchmarks lack adequate holdout sets, or even fail to include them entirely. This is also true for API-calling benchmarks. For instance, APIGen does not provide a clear train-test split, making it difficult to evaluate models using the provided benchmark data. Similarly, ToolBench includes a very small test set that fails to represent the diversity of real-world scenarios. This limitation in benchmark design often results in models that perform well on narrow datasets but struggle with generalization. We illustrate this through an example of API retrievers, as retrieving

Table 2: Comparison of benchmarks on evaluation

| | **Toolbench** | **APIGen** | **AnyTool** | **MetaTool** | **APIBench** |
|---|---|---|---|---|---|
| API retrieval | ✓ | ✓ | ✓ | ✓ | ✗ |
| API call | ✗ | ✓ | ✗ | ✗ | ✗ |
| Final response | ✓ | ✗ | ✗ | ✗ | ✓ |

the appropriate APIs is critical to overall performance. Previous works [Qin et al., 2023, Kong et al., 2023] trained small sentence transformer models [Reimers, 2019] using contrastive learning objectives, and claim superior performance over proprietary, general-purpose embedding models like OpenAI's. However, we find that while these trained retrievers performed well on the same source of APIs, their performance significantly declines when applied to different sources. As shown in Figure 2, the retriever trained on ToolBench outperforms OpenAI's embedding model on three RapidAPI-based benchmarks but performs significantly worse on MetaTool, a dataset based on OpenAI plugins. This highlights the importance of generalization to various API sets, a critical feature in realistic scenarios where APIs are frequently updated, deprecated, or custom-built by users.

### 3.2 Bias Towards Simple Queries

To better simulate realistic scenarios, we emphasize the need for multi-tool and multi-step benchmarks. As state-of-the-art LLMs become increasingly adept at basic function calls [Qu et al., 2024], it is crucial to assess their ability to decompose complex user queries into smaller, executable substeps and plan actions accordingly. Unfortunately, most existing benchmarks consist predominantly of single-step queries, where a single API call is sufficient to complete the task, as shown in Fig 1. Notably, one of the most widely-used benchmarks, APIBench [Patil et al., 2023], consists solely of single-step and single-tool queries. Existing benchmarks also lack realistic queries that require sequentially dependent reasoning. We show a few examples in the Appendix A.2.

### 3.3 General Instability

A significant issue with current benchmarks is their instability, as static ground truth quickly becomes outdated, and API services exhibit variability. API responses are time-sensitive, and services change over time due to factors like deprecation, shifts in service definitions, and altered response behaviors. This instability makes it difficult to evaluate new systems on older benchmarks, hindering efforts to standardize evaluation. To mitigate these challenges, several benchmarks restrict themselves to small sets of hard-coded and deterministic tools [Schick et al., 2024, Lu et al., 2024], instead of dynamic real-world APIs. Other benchmarks limit themselves in both API pool size and the number of queries.

One widely used benchmark for API-use, that does not limit its scope as above, is the ToolBench benchmark. Despite the efforts of ToolBench to enhance stability by introducing a proxy server for RapidAPI, which simplifies access by handling authentication and overhead associated with real-time APIs, we found ToolBench to be highly unstable. Many APIs fail to return consistent responses and frequently produce a variety of errors. Previous work [Guo et al., 2024] highlighted ToolBench's instability and proposed solutions like caching API responses and using a GPT-4-based simulator for unresponsive APIs. However, these fixes were limited to a small subset of ToolBench, and running LLM simulators on demand still introduces stochastic variability.

### 3.4 Incomplete Evaluation

For a comprehensive evaluation of API-calling systems, it is essential to assess each stage of the pipeline: whether the correct tools are retrieved, whether the correct tools are called, whether the tool calls are accurate, and finally, whether the query response is correct. The real-time nature of APIs further complicates the evaluation of API-use systems, and existing works focus only on partial components of the overall pipeline described in Section 2. As shown in Table 2, AnyTool and MetaTool only evaluate API retrieval, while APIGen evaluates both API retrieval and calls but overlooks the fact that multiple trajectories can lead to a successful final response. Although ToolBench aims to provide ground truth across the entire pipeline, the provided ground truth labels for tool calls and final responses is somewhat outdated and unreliable. Approximately 40% of queries end up as "unsolvable," where the LLM used in ToolBench's own implementation fails to generate a

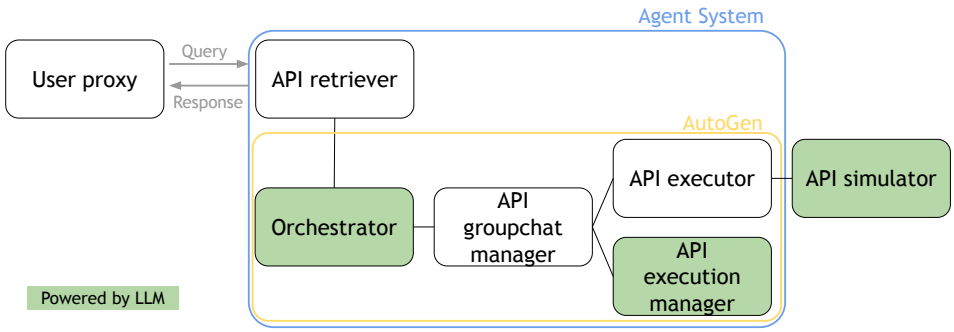



Figure 3: AutoGen system architecture

valid output. Each stage of the API-calling system needs to be evaluated to enable more reliable and comprehensive assessment.

## 4 SEAL Construction

SEALis a comprehensive, end-to-end testbed that builds on top of current benchmarks. It includes standardization and sanitization of queries from existing benchmarks, a flexible agent system capable of adapting to user demands, and a thorough evaluation pipeline.

### 4.1 Benchmark Standardization & Sanitization

We parse and standardize five existing benchmarks introduced in Table 1, into a unified format consisting of queries, APIs, query-to-API mapping, and query-to-API-call mapping. Note that API-call data is available for only 3 out of the 5 datasets, so this field remains empty for AnyTool and MetaTool. This standardized format enables the use of multiple benchmarks with diverse structures via a unified approach. For instance, users can search for APIs associated with a specific query ID or retrieve an API by its name or features. Data sanitization and filtering details are in the Appendix A.1.

### 4.2 Agent System Construction

Previous works have developed single LLMs to handle every step of the API-calling process. However, we argue that API-calling, and tool-calling in general, can be tackled more effectively using an agent-based system, where multiple agents collaborate, with some powered by LLMs. This design of separation allows for testing both tool retrieval and tool planning methods, enabling users to easily swap different agents in and out. It also opens up the possibility of replacing agents with smaller, specialized models, rather than relying on expensive general models to handle everything. We develop an adaptable agent system based on the AutoGen [Wu et al., 2023] framework. One of the key advantages of AutoGen is its customizable agents, which can operate in various modes by leveraging combinations of LLMs, human inputs, and tools/APIs. This flexibility allows the system to dynamically adjust agent behaviors based on user-specified configurations.

Fig. 3 illustrates the current architecture of SEAL. We follow the workflow of existing systems, where APIs are pre-selected and registered within the agent system. Within this system, API-calling is managed through a "group chat" interaction between a API Executor and a API Execution Manager. The API Execution Manager, powered by an LLM, generates the necessary API calls and arguments, while the API Executor communicates with a API Simulator in the background to simulate real API servers. Although this represents the simplest version of the architecture, the system is designed to allow easy addition or removal of agents. For instance, one could add a Planner Agent under the Orchestrator to break down multi-step queries into smaller sub-steps or introduce a Verifier Agent to ensure that each step has been executed correctly.

**API Retriever** Given the large size of the API pool in API-calling benchmarks, it is impractical to register all APIs with an LLM due to context length limits. We implement an API retriever as a separate class, allowing users to specify the embedding model of their choice. Available

Table 3: SEAL execution results on two benchmarks.

| **Benchmark: Toolbench** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # of queries | | API retrieval recall@10 | | API call recall | | API param. acc. | | Pass rate | |
| # of apis | Mean | Std. | Mean | Std. | Mean | Std. | Mean | Std. | Mean | Std. |
| 10 | 30.67 | 18.04 | 1.00 | 0.00 | 0.62 | 0.19 | 0.34 | 0.30 | 0.68 | 0.16 |
| 50 | 279.00 | 29.21 | 0.95 | 0.06 | 0.56 | 0.11 | 0.31 | 0.03 | 0.78 | 0.10 |
| 100 | 524.00 | 33.96 | 0.95 | 0.04 | 0.57 | 0.12 | 0.39 | 0.07 | 0.82 | 0.03 |
| 200 | 865.00 | NA | 0.91 | NA | 0.63 | NA | 0.43 | NA | 0.73 | NA |
| 500 | 2019.00 | NA | 0.85 | NA | 0.59 | NA | 0.44 | NA | 0.76 | NA |
| **Benchmark: APIGen** | | | | | | | | | | |
| | # of queries | | API retrieval recall@10 | | API call recall | | API param. acc. | | Pass rate | |
| # of apis | Mean | Std. | Mean | Std. | Mean | Std. | Mean | Std. | Mean | Std. |
| 10 | 10.00 | 4.36 | 1.00 | 0.00 | 0.98 | 0.03 | 0.93 | 0.01 | 0.87 | 0.12 |
| 50 | 116.00 | 32.36 | 0.96 | 0.01 | 0.94 | 0.02 | 0.80 | 0.10 | 0.80 | 0.05 |
| 100 | 375.33 | 118.59 | 0.97 | 0.02 | 0.95 | 0.03 | 0.84 | 0.06 | 0.84 | 0.03 |
| 200 | 726.00 | NA | 0.95 | NA | 0.93 | NA | 0.82 | NA | 0.86 | NA |
| 500 | 2456.00 | NA | 0.93 | NA | 0.87 | NA | 0.79 | NA | 0.85 | NA |

options include OpenAI embedding models, Gemini embedding models, and a customized sentence transformer model proposed by Qin et al. [2023]. Once the user query and API documents are embedded, the retriever efficiently performs vector lookups using the Faiss library.

**API Simulator** Since real APIs are inherently real-time, and we find existing benchmarks' API servers to be highly unstable, we develop a custom API simulator powered by GPT-4. Similar approach is used in Guo et al. [2024], where real-time API responses are cached and the simulator is called on demand. In our case, we fully replace and cache all API responses with simulations to minimize stochastic variability. Given instructions for simulation, API documentation, and API parameters, the simulator replicates API behavior. The simulator prompt can be found in Appendix A.4. Although we acknowledge that LLM-based systems may generate fabricated information, we believe it is essential to have reliable and deterministic APIs from the perspective of benchmarking.

### 4.3 Evaluation Pipeline

Previous works evaluate one of the following: (1) the final response of the system [Qin et al., 2023], considering that multiple reasoning trajectories can lead to success, or (2) a single hard-coded trajectory by checking if the API calls are correct [Chen et al., 2024]. We argue that a more comprehensive evaluation is needed, spanning the entire pipeline of API usage, including API retrieval, API calls, and the final response.

- **API Retrieval:** Did the system retrieve the correct tools? All five benchmarks listed in Table 2 provide ground truth for tool retrieval. We leverage this information and use standard retrieval metrics, such as Recall@K and Mean Reciprocal Rank (MRR), where $K = 10$.
- **API Call:** Were the correct tools called, and were the parameters passed accurately? We assess tool call performance by measuring recall, ignoring the order of calls and treating repeated calls to the same tool as distinct events. To evaluate API arguments, we use tool call accuracy, determining whether parameters and values were an exact match. Additionally, more fine-grained metrics, such as matching parameter names or values, can be employed as in Trivedi et al. [2024].
- **Final Response:** Did the system fully address the user query, and was the final response accurate? We adopt the AI critique methodology from Qin et al. [2023] to compute Pass Rate. First, an LLM assesses whether the query was successfully executed by analyzing the final response and categorizing it as solved, unsolved, or unsure. If unsure, it re-evaluates with the entire execution trajectory.

## 5 Results & Analysis

We evaluate SEAL using four key metrics: Recall@10 for API retrieval, Recall@N for API calls (varying N as the number of ground truth APIs), API parameter accuracy, and the pass rate of the final response. `gpt-4-turbo-2024-04-09` serves as a single model handling API calling, parameter generation, and response summarization.

Fig. 4 and Table 3 presents results from ToolBench and APIGen evaluations. We progressively increase the size of the API pool to assess whether task complexity scales with API pool size. As the API pool grows, the associated queries also vary. Our results indicate that as the API pool expands, performance declines in both API retrieval and call accuracy, demonstrating that the task becomes more challenging with more APIs. However, this decline is less evident in the pass rate. We also conduct three sampling runs with API pool sizes of 10, 50, and 100. The standard deviation across these runs is substantial, particularly in ToolBench. This is due to the limited number of queries associated with each API, causing each sampling to yield significantly different query distributions. In contrast, APIGen exhibit smaller but still notable variance. We plan to increase the number of bootstrapping runs in future work for more reliable and scalable evaluation.

We identify three major error cases:

**Incorrect API Retrieval** The most common error is partially solved queries due to incorrect API retrieval, especially for multi-tool queries. Current vector lookup systems struggle to retrieve multiple relevant APIs. This underscores the need for multi-step, hierarchical planning to break down complex queries into sub-steps for more accurate API retrieval and resolution. Below is an example where the system partially solves a query because it fails to retrieve an API for fetching technical specifications of a film.

- Query: What are the technical specifications of the film 'tt0133093' and search for news in Italian about 'tecnologia'.
- Final Answer:
  # Technical Specifications of the Film 'The Matrix' (tt0133093)
  There was an error retrieving the technical specifications for the film 'The Matrix' (tt0133093).
  # News in Italian about 'Tecnologia'
  1. **Tech Trends in Cinema: How Technology is Shaping the Future of Film** An in-depth article exploring the latest technological advancements in the film industry and their impact on movie production and viewing experiences...
- Result: Unsolved
- Reason: The answer does not provide the technical specifications of the film 'tt0133093' as requested in the query. It does, however, provide news articles in Italian about 'tecnologia'.

**Incorrect API Parameters** Another frequent errors are incorrect API parameters. This highlights the challenge LLMs face in correctly extracting relevant information from queries. For example, below is a case where the system incorrectly identifies latitude and longitude values.

- Query: Can you find the time zone for a Canadian postal code G2J and a latitude-longitude pair of 48.8567, 2.3508?
- Final Answer: I successfully retrieved the geographical boundary information for the Canadian postal code FSA "G2J," which corresponds to a region in Quebec City, Quebec, Canada. However, there was an error in fetching the time zone for the latitude-longitude pair (48.8567, 2.3508).
- Correct API call: `{ 'arguments': { 'lat': 48.8567, 'lng': 23.508 }, 'name': 'tz_lookup_by_location' }`
- Actual API call: `{ 'arguments': { 'lat': 488567, 'lng': 23508 }, 'name': 'tz_lookup_by_location' }`
- Result: Unsolved
- Reason: The tool call for the latitude-longitude pair (48.8567, 2.3508) returned an error due

to invalid values provided. The correct latitude-longitude values should be within the range of -90 to 90 for latitude and -180 to 180 for longitude.

**Inconsistent Evaluation** There are cases where evaluation is inconsistent, largely because the LLM serves as both the solver and the evaluator. LLM critique is stochastic, and at times, inconsistent, leading to different outcomes when re-evaluating the same query. Below is an example where, despite fully addressing the query, the LLM incorrectly marked it as unsolved. This variability underscores the importance of multi-faceted evaluation beyond relying solely on final output assessments.

• Query: Fetch auto-complete results for 'winter coat' from Kohls and 'kitchen appliance' from Wayfair.
• Final Answer: Here are the auto-complete suggestions for your queries:
**For 'winter coat' from Kohls:**
- Winter Coat for Men, Winter Coat for Women, ...
**For 'kitchen appliance' from Wayfair:**
- Kitchen appliance set, Kitchen appliance package deals...
These suggestions are based on popular searches and available products related to your queries.
• Result: Unsolved
• Reason: The answer provides auto-complete suggestions for 'winter coat' from Kohls and 'kitchen appliance' from Wayfair, which are based on popular searches and available products related to the queries.

## 6 Related Work

**LLM Tool-use** Recent works have proposed methods and models for enabling LLMs to interact with Tools and real-world APIs. [Xu et al., 2023b, Zeng et al., 2023, Shen et al., 2024, Wang et al., 2023, Xu et al., 2023a, Chen et al., 2023, Ruan et al., 2023, Li et al., 2024, Song et al., 2023, Yang et al., 2024, Schick et al., 2024] Chen et al. [2024], Qin et al. [2023], Patil et al. [2023] also propose custom models trained on their respective benchmarks, based on open-source models, demonstrating solid in-domain and out-of-domain performance. Our testbed facilitates the development of more advanced LLMs for tool-use by providing a ready-made framework, allowing researchers to easily test new planning methods without building an entire system from scratch.

**Tool-use Benchmarks** In addition to the five benchmarks previously discussed, several others have been proposed for evaluating LLMs' tool-use capabilities [Li et al., 2023, Patil et al., 2023, Xu et al., 2023a]. $\tau$-bench [Yao et al., 2024], ToolSandbox [Lu et al., 2024], and AppWorld [Trivedi et al., 2024] focus on the interactive and conversational aspects between users and agents in tool-learning. For multimodal benchmarks, m&m's [Ma et al., 2024] and MLLM-Tool [Wang et al., 2024] test LLMs' performance across natural language and vision. ToolQA [Zhuang et al., 2023] evaluates LLMs' ability to use external tools for question answering, while Ultratool [Huang et al., 2024] also incorporates tool creation alongside other aspects. Our benchmark suite is comprehensive and it recycles existing benchmarks' queries and ground truth while addressing missing components in evaluation.

## 7 Conclusion

In this work, we introduce SEAL, a comprehensive testbed designed to address key gaps in existing LLM tool-use benchmarks, particularly for real-world API interactions. We identify critical issues like overfitting, limited support for multi-step reasoning, and instability due to dynamic API behaviors. To overcome these challenges, SEAL include an API simulator powered by GPT-4 and caching responses and a robust evaluation framework covering the full API usage pipeline—from retrieval and calls to final response. Our agent-based system provide a structured platform for reliable performance comparisons. We believe SEAL will facilitate better development and evaluation of API-driven LLMs, supporting more rigorous and reproducible testing in future research.

## References


Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Yujia Chen, Cuiyun Gao, Muyijie Zhu, Qing Liao, Yong Wang, and Guoai Xu. Apigen: Generative api method recommendation. *arXiv preprint arXiv:2401.15843*, 2024.

Zhipeng Chen, Kun Zhou, Beichen Zhang, Zheng Gong, Wayne Xin Zhao, and Ji-Rong Wen. Chatcot: Tool-augmented chain-of-thought reasoning on chat-based large language models. *arXiv preprint arXiv:2305.14323*, 2023.

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*, 2024.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*, 2024.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, et al. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *arXiv preprint arXiv:2401.17167*, 2024.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Yilun Kong, Jingqing Ruan, Yihong Chen, Bin Zhang, Tianpeng Bao, Shiwei Shi, Guoqing Du, Xiaoru Hu, Hangyu Mao, Ziyue Li, et al. Tptu-v2: Boosting task planning and tool usage of large language model-based agents in real-world systems. *arXiv preprint arXiv:2311.11315*, 2023.

Chuanhao Li, Runhan Yang, Tiankai Li, Milad Bafarassat, Kourosh Sharifi, Dirk Bergemann, and Zhuoran Yang. Stride: A tool-assisted llm agent framework for strategic and interactive decision-making. *arXiv preprint arXiv:2405.16376*, 2024.

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

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. *arXiv preprint arXiv:2408.04682*, 2024.

Zixian Ma, Weikai Huang, Jieyu Zhang, Tanmay Gupta, and Ranjay Krishna. m&m's: A benchmark to evaluate tool-use for multi-step multi-modal tasks. In *Synthetic Data for Computer Vision Workshop@ CVPR 2024*, 2024.

Dheeraj Mekala, Jason Weston, Jack Lanchantin, Roberta Raileanu, Maria Lomeli, Jingbo Shang, and Jane Dwivedi-Yu. Toolverifier: Generalization to new tools via self-verification. *arXiv preprint arXiv:2402.14158*, 2024.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *arXiv preprint arXiv:2405.17935*, 2024.

N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Ziyue Li, Xingyu Zeng, et al. Tptu: large language model-based ai agents for task planning and tool usage. *arXiv preprint arXiv:2308.03427*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324*, 2024.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*, 2023.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

Chenyu Wang, Weixin Luo, Qianyu Chen, Haonan Mai, Jindi Guo, Sixun Dong, XM Xuan, Zhengxin Li, Lin Ma, and Shenghua Gao. Mllm-tool: A multimodal large language model for tool agent learning. *arXiv preprint arXiv:2401.10727*, 4, 2024.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

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

Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, et al. Lemur: Harmonizing natural language and code for language agents. *arXiv preprint arXiv:2310.06830*, 2023b.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. Gpt4tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems*, 36, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. $\tau$-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36: 50117–50143, 2023.

# A Appendix

## A.1 Benchmark Standardization Details

**ToolBench Sanitization** A significant issue with ToolBench is the presence of numerous unsolvable queries, which degrade the overall quality of the benchmark. To address this, we apply a series of filters to exclude queries that cannot be mapped to the available API pool. Specifically, we use the G1 subset of ToolBench training data (the largest among the G1-G3 subsets), which contains 72,783 queries. We then filter out queries that meet any of the following criteria:

- The finish type is "give_up," indicating that the query could not be solved using the available APIs.
- No matching APIs are found in the API pool, including cases of hallucinated function names.
- Incorrect functions, not present in the function pool, are called.
- Errors occur while parsing API arguments.

After preprocessing, the total number of queries is reduced to 40,399, with 8,684 APIs remaining. Following additional multi-step and multi-tool filtering, we arrive at a final count of 34,055 queries and 7,559 APIs.

**Multi-step Filtering** We filter queries to focus on those requiring multi-step reasoning and the use of multiple tools. Since ToolBench and APIGen provide tool-call ground truth, we leverage this information to identify multi-step queries. For datasets that do not include ground truth information such as tool selection benchmarks, we extract only the multi-tool queries.

## A.2 Necessity for More Realistic Benchmarks

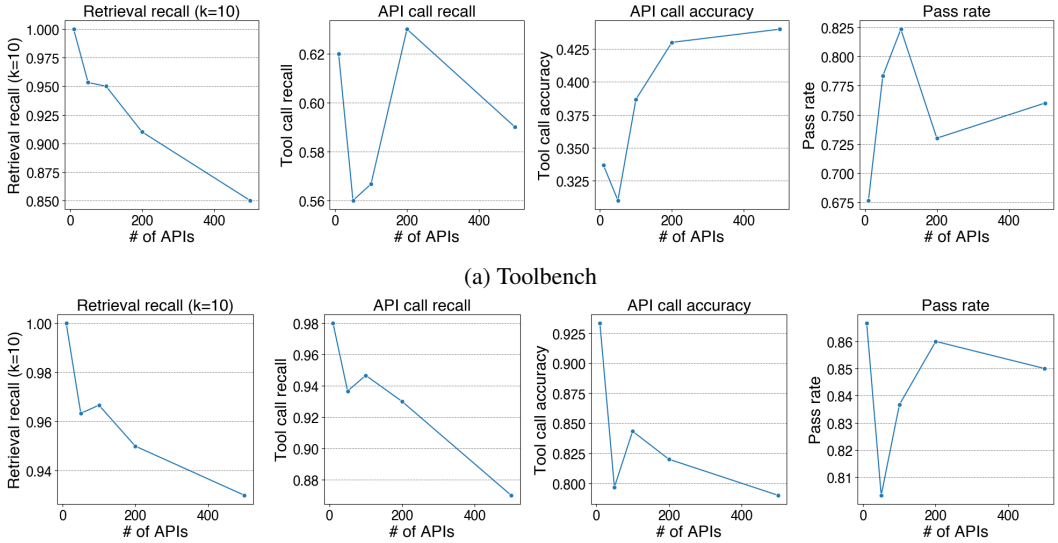
Current benchmarks often fail to reflect real-world scenarios, particularly when it comes to queries requiring sequential, dependent reasoning. For example, ToolBench frequently features simplistic user queries like "Perform a task with API A," which typically result from having LLMs generate queries for a given set of APIs. Among the benchmarks we analyzed, APIGen stands out for having the highest-quality queries. However, even APIGen rarely includes multi-step queries that requires sequentially dependent reasoning. Instead, most queries in APIGen are parallel, such as "Perform tasks A and B," where tasks A and B are independent of each other. We provide examples of this dynamic below.

- Query: Generate the first 10 Fibonacci numbers and calculate the standard deviation of the numbers [3.5, 4.2, 5.1, 6.7, 7.3]
- Query: Can you split the list [1, 2, 3, 4, 5, 6] into chunks of size 2, and then generate a random string of length 10 with only uppercase letters?
- Query: Find verses with 'wisdom', 'knowledge', and 'understanding'. Also, generate a 15-character random password.

As a result, models like GPT-4 continue to perform well, achieving over an 80% pass rate on these benchmarks. This is despite relying on a relatively simple setup, where a single LLM handles all tasks. We find that a significant portion of existing benchmark queries follow this pattern, highlighting the need for more complex benchmarks. Such benchmarks would provide a more accurate measure of LLM capabilities in real-world scenarios.

## A.3 Bootstrapping Results

Figure 4 presents a line plot based on the data from Table 3. As the size of the API pool progressively increases, we observe an overall degradation in performance. However, this trend is less pronounced in the pass rate.




(a) Toolbench

(b) APIGen

Figure 4: SEAL execution results on two benchmarks.

### A.4 API Simulator Prompt

```
Imagine you are an API Server. Your role is to simulate API calls
    based on the API documentation provided in a JSON format. API
    documentation includes the API's name, description, and input
    parameters. There are two types of parameters: required and
    optional. Optional parameters are specified as "optional" in the
    "type" field.

Following is the documentation for the API you need to simulate:

    {API_INFO}

Your task is to generate a JSON response that aligns with the
    expected output of the API. As you receive specific inputs for
    this API call, analyze these inputs to determine their intended
    purpose.
Your responses must adhere to a specific JSON structure as the
    following:

{
    "error": "",
    "response": "<Your_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 and directly
    address the API's intended functionality. If the provided
    examples are mostly error messages or lack substantial content,
    use your judgment to create relevant and accurate responses. The
```

```
    key is to maintain the JSON format's integrity while ensuring
    that your response accurately reflects the API's intended output.

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 be around 100 to 200 words, containing rich
    information given the api input parameters. Keep Your answer
    short and simple.
- your response must be effective and have practical content.
- try to simulate the API call and return as helpful information as
    possible. Instead of saying "The API successfully executed and
    returned something", provide a more detailed response.
- do not mention that this is a simulation in your response, assume
    that this is a real scenario and provide imaginary responses if
    the information required is not available
```

### A.5 SEAL Execution Example

Figure 5 illustrates an actual execution example from SEAL. Built on top of the AutoGen framework, SEAL allows users to easily integrate and experiment with different agents in a plug-and-play fashion. Additionally, the system enables monitoring of interactions between agents, providing a flexible and user-friendly environment for testing various tool-use scenarios.

```
********************************************************************************
Starting a new chat....

********************************************************************************
user (to orchestrator):

What are the coordinates of Paris and Moscow?

--------------------------------------------------------------------------------

********************************************************************************
Starting a new chat....

********************************************************************************
orchestrator (to chat_manager):

Please select the most relevant tools and execute them: What are the coordinates of Paris and Moscow?

--------------------------------------------------------------------------------

Next speaker: tool_execution_manager

tool_execution_manager (to chat_manager):

***** Suggested tool call (call_Sz6bHs5lRSugb6uUlG3PQf00): geocode_city *****
Arguments:
{"city": "Paris"}
**********************************************************************
***** Suggested tool call (call_GfOsHWrHeEooGTxbaGI1xaqv): geocode_city *****
Arguments:
{"city": "Moscow"}
**********************************************************************

--------------------------------------------------------------------------------

Next speaker: tool_executor


>>>>>>>> EXECUTING FUNCTION geocode_city...

>>>>>>>> EXECUTING FUNCTION geocode_city...
tool_executor (to chat_manager):

tool_executor (to chat_manager):

***** Response from calling tool (call_Sz6bHs5lRSugb6uUlG3PQf00) *****
{"error": "", "response": {"city": "Paris", "coordinates": {"latitude": 48.8566, "longitude": 2.3522}, "description": "Paris, Fra
nce's capital, is a major European city and a global center for art, fashion, gastronomy, and culture. Its 19th-century cityscape
 is crisscrossed by wide boulevards and the River Seine. Beyond such landmarks as the Eiffel Tower and the 12th-century, Gothic N
otre-Dame cathedral, the city is known for its cafe culture and designer boutiques along the Rue du Faubourg Saint-Honor\u00e9."}
}
**********************************************************************

--------------------------------------------------------------------------------
tool_executor (to chat_manager):

***** Response from calling tool (call_GfOsHWrHeEooGTxbaGI1xaqv) *****
{"error": "", "response": {"city": "Moscow", "coordinates": {"latitude": 55.755826, "longitude": 37.6173}, "description": "Moscow
, the capital city of Russia, is located at the coordinates 55.755826 latitude and 37.6173 longitude. It is known for its histori
cal architecture, including the Kremlin, Red Square, and the iconic St. Basil's Cathedral. Moscow is not only a major political,
economic, cultural, and scientific center in Russia and Eastern Europe, but it also serves as the largest city entirely on the Eu
ropean continent."}}
**********************************************************************
--------------------------------------------------------------------------------

Next speaker: tool_execution_manager

tool_execution_manager (to chat_manager):

The coordinates for Paris and Moscow are as follows:

- **Paris, France**
  - **Latitude:** 48.8566
  - **Longitude:** 2.3522
  - **Description:** Paris, France's capital, is a major European city and a global center for art, fashion, gastronomy, and cult
ure. Its 19th-century cityscape is crisscrossed by wide boulevards and the River Seine. Beyond such landmarks as the Eiffel Tower
 and the 12th-century, Gothic Notre-Dame cathedral, the city is known for its cafe culture and designer boutiques along the Rue d
u Faubourg Saint-Honoré.

- **Moscow, Russia**
  - **Latitude:** 55.755826
  - **Longitude:** 37.6173
  - **Description:** Moscow, the capital city of Russia, is known for its historical architecture, including the Kremlin, Red Squ
are, and the iconic St. Basil's Cathedral. Moscow is not only a major political, economic, cultural, and scientific center in Rus
sia and Eastern Europe, but it also serves as the largest city entirely on the European continent.

--------------------------------------------------------------------------------
```

Figure 5: Execution example of SEAL