# CRITICTOOL: Evaluating Self-Critique Capabilities of Large Language Models in Tool-Calling Error Scenarios

**Shiting Huang**[1*] **Zhen Fang**[1*] **Zehui Chen**[1] **Siyu Yuan**[2] **Junjie Ye**[2]
**Yu Zeng**[1] **Lin Chen**[1] **Qi Mao**[3] **Feng Zhao**[1†]
[1]MoE Key Lab of BIPC, USTC [2]Fudan University
[3]Communication University of China

## Abstract

The ability of large language models (LLMs) to utilize external tools has enabled them to tackle an increasingly diverse range of tasks. However, as the tasks become more complex and long-horizon, the intricate tool utilization process may trigger various unexpected errors. Therefore, how to effectively handle such errors, including identifying, diagnosing, and recovering from them, has emerged as a key research direction for advancing tool learning. In this work, we first extensively analyze the types of errors encountered during the function-calling process on several competitive tool evaluation benchmarks. Based on it, we introduce CRITICTOOL, a comprehensive critique evaluation benchmark specialized for tool learning. Building upon a novel evolutionary strategy for dataset construction, CRITICTOOL holds diverse tool-use errors with varying complexities, which better reflects real-world scenarios. We conduct extensive experiments on CRITICTOOL, and validate the generalization and effectiveness of our constructed benchmark strategy. We also provide an in-depth analysis of the tool reflection ability on various LLMs, offering a new perspective on the field of tool learning in LLMs. The code is available at https://github.com/Shellorley0513/CriticTool.

## 1 Introduction

Large Language Models (LLMs) represent a groundbreaking advancement in artificial intelligence, demonstrating remarkable capabilities in various tasks (Zhao et al., 2023; Jiang et al., 2024; Chen et al., 2023; McAleese et al., 2024). The interaction between LLMs and external tools empowers them to address more complex tasks, as these tool-calling systems increasingly adapt to dynamic real-world environments (Chen et al., 2024c).

Driven by practical applications and attractive ability, the evaluation of tool-use capabilities for LLMs remains a topic of ongoing research. Existing works are typically confined to single-tool usage scenarios (Xu et al., 2023; Patil et al., 2024) or comparing the executions with predefined golden answers (Shen et al., 2023; Ye et al., 2024a,b; Chen et al., 2024b). However, real-world applications often involve complex and multi-step tool-calling tasks, where intricate intermediate trajectories introduce opportunities for errors arising either from LLMs themselves (Yan et al., 2024; Sun et al., 2024) or from external factors (Guo et al., 2024a). Due to the complexity of the external environment, combined with the inherently challenging nature of tool-use tasks, neglecting the process status of tool invocation may result in biased evaluation. Current work primarily addresses these challenges by either filtering out erroneous data (Liu et al., 2024) or treating errors as suboptimal nodes to expand the tool answer search space (Qin et al., 2023; Chen et al., 2024a; Abdelaziz et al., 2024; Song et al., 2024), while existing benchmarks focus mainly on categorizing error types without examining how LLMs detect and recover from them (Kokane et al., 2024). As a result, these approaches fail to provide insights into error-handling behaviors during tool calls, leading to an insufficient evaluation of their tool-use capabilities. Given the diverse sources of errors and the various strategies required to address them, we argue that the benchmarks which overlook LLMs' error recovery cannot accurately evaluate a model's actual tool-use capability.

To address these challenges, we introduce CRITICTOOL, the first self-critique evaluation benchmark for tool utilization of LLMs. Distinct from prior result-oriented evaluation methods, we categorize error patterns more finely and evaluate models from multiple perspectives, enabling a deeper exploration of LLMs' tool-use capabilities in error-prone scenarios. Specifically, we categorize errors from two main sources: internal model-driven errors and external environment errors. We then di-

[*] Equal Contribution
[†] Corresponding author



*Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 26672–26704
November 4-9, 2025 

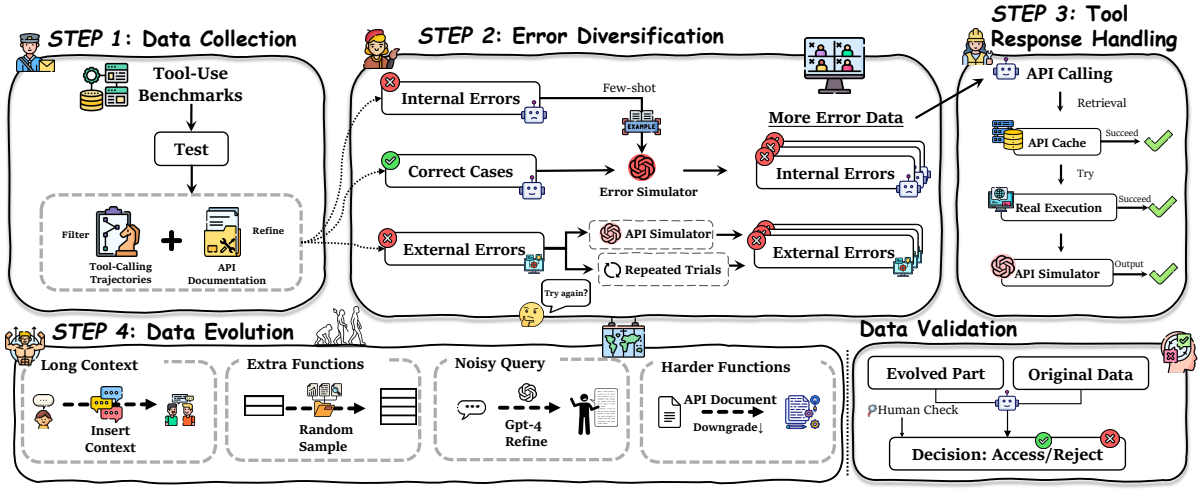


Figure 1: **Overview of CRITICTOOL construction pipeline.** The pipeline begins with collecting and testing tool-use benchmarks to obtain a variety of correct and incorrect tool-calling trajectories. GPT-based simulators and repeated API calls are employed to diversify internal and external error patterns. And responses to internal errors are generated via cache retrieval, API execution, and API simulator. Finally, the error data is evolved using four distinct strategies, followed by verification and manual review.

versify our error dataset by ensuring the errors span a wide range of tools and design fine-grained evaluation protocols for two sources of errors. This paradigm enables a granular evaluation of LLMs' self-critique capabilities across different dimensions: reflect and correct for internal model-driven errors, and retry with skip or finish for external environment errors.

By conducting extensive experiments on CRITICTOOL, we perform a thorough analysis of the results, providing valuable insights into LLMs' behavior when encountering different types of errors during tool calls. We observe that different models exhibit varying self-critique behaviors when encountering errors from different sources.

The main contributions of our work are summarized as follows:

- We observe LLMs' performance in several popular and high-quality tool-use benchmarks and provide a comprehensive analysis of error distributions.
- To the best of our knowledge, we are the first to introduce CRITICTOOL, a tool self-critique evaluation benchmark for LLMs, categorizing errors from different sources and patterns.
- We propose a novel data evolution strategy to enrich the error dataset by incorporating more complex data scenarios, thus broadening the scope and depth of evaluation for LLMs in real-world applications.
- With extensive experiments, we provide a detailed analysis of the self-critique ability of various LLMs, offering a new perspective in the field of tool learning.

Table 1: The success rates (%) of advanced LLMs in recovering from errors across the four datasets.

| | NESTFUL | API-Bank | T-Eval | BFCL |
|---|---|---|---|---|
| **Qwen-turbo** | 12.64 | 6.25 | 35.14 | 29.47 |
| **Qwen2.5-72B** | 13.87 | 8.69 | 38.71 | 22.73 |
| **GPT-3.5** | 18.10 | 7.69 | 51.11 | 7.14 |
| **GPT-4o** | 22.16 | 17.39 | 54.44 | 28.57 |

## 2 CRITICTOOL

In this section, we begin with presenting an in-depth analysis of the key issues in current tool learning, highlighting the pressing need for tool-specific critique evaluation benchmarks. Building on these observations, we introduce CRITICTOOL, a benchmark designed to systematically explore LLMs' self-critique[1] capabilities.

### 2.1 Motivation: LLMs' Performance on Popular Tool-Use Benchmarks

Tool utilization is a critical yet challenging task in large language model (LLM) applications, requiring sophisticated reasoning and practical adaptation. To identify the current limitations in tool learning, we conduct an in-depth analysis of LLM's behavioral patterns across various tool-calling benchmarks (Refer to Appendix A for more details). As shown in Tab. 1, our investigation reveals a noteworthy phenomenon: *most LLMs strug-*

[1]The model identifying and correctly handling errors.

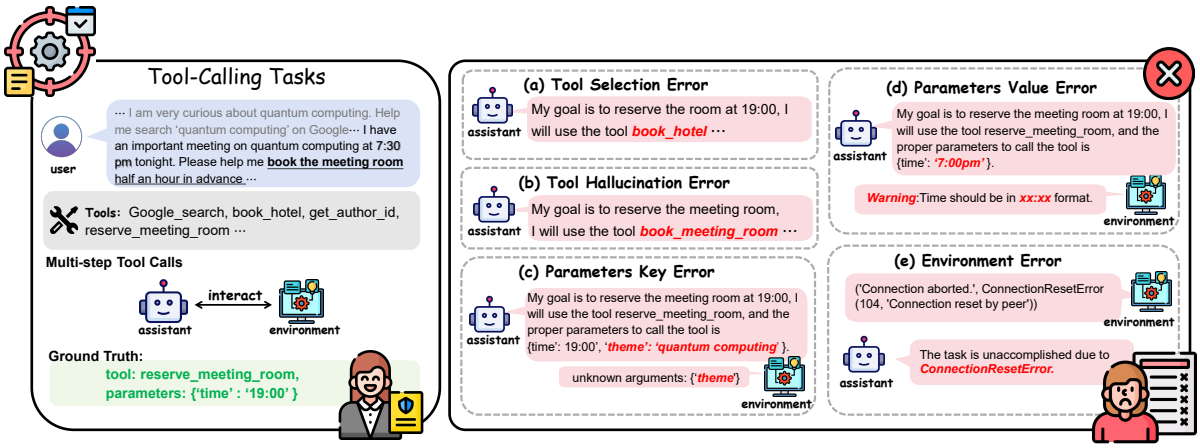


Figure 2: **Examples of Errors in multi-step tool call tasks.** Multi-step tool call errors are categorized into five patterns based on the source and characteristics of the errors: **Tool Selection Errors, Tool Hallucination Errors, Parameters Key Errors, Parameters Value Errors and Environment Errors.**

*gle to recover from errors*[2] *during the tool-calling process, resulting in eventual task failure*. This issue becomes particularly pronounced as tasks grow more complex and long-horizon. Despite the significance of this limitation, existing tool utilization benchmarks rarely directly consider the ability for self-critique, leading to insufficient attention toward improving this capability in tool learning. As highlighted by o1 (OpenAI, 2024), the ability to self-critique is essential for executing long-horizon tasks effectively and serves as a pathway to scalable oversight in LLM reasoning. In this work, we seek to fill this gap by introducing CRITICTOOL, a benchmark designed to systematically evaluate the self-critique capability in tool learning.

## 2.2 Dataset Construction

The construction of the dataset in CRITICTOOL consists of four main phases: tool-use data collection, error diversification, tool response handling, and data evolution. The overview of the construction is shown in Fig. 1. More implementation details can be found in Appendix C.1 and C.2.

### 2.2.1 Error Patterns

From our observations of LLMs' tool-use performance in § 2.1, we identify several frequently occurring error patterns when LLMs function as tool-calling assistants, as illustrated in Fig. 2. These errors stem from two primary sources: model capability limitations often give rise to internal model-driven errors related to both tool and parameter handling, while external environment errors will disrupt task completion.

- **Tool Selection Errors**: The assistant selects an existing but unsuitable tool for the given task, often resulting from generating an incorrect goal, or misunderstanding usage of the tool.
- **Tool Hallucination Errors**: The assistant attempts to use a non-existent tool, typically caused by task misinterpretation or failure to recognize available tools.
- **Parameter Key Errors**: The assistant passes incorrect parameter keys, either omitting required ones or including irrelevant keys, usually due to task miscomprehension or forgetting tool requirement details.
- **Parameter Value Errors**: The assistant provides incorrect parameter values, usually stemming from failure to comply with the expected input format or overlooking task details.
- **Environment Errors**: Real-world APIs may not always be stable (Guo et al., 2024a). Issues such as connection timeouts or lack of user permissions can disrupt tool interactions, and may cause the assistant to endlessly retry failed calls.

### 2.2.2 Tool-Use Data Collection

To construct CRITICTOOL, our goal is developing a tool-use dataset that spans diverse domains of tools and captures a wide range of errors that LLMs encounter in tool call scenarios. Existing benchmarks have already collected realistic APIs and generated well-designed tool-use tasks with excellent diversity and appropriate complexity, making them ideal sources of tool-use data. We use the datasets from high-quality tool-use benchmarks, including BFCL v3 (Yan et al., 2024) and T-Eval (Chen et al.,

[2]Recover from error refers to the ability of an LLM to successfully handle an error in a given step.

2024b), which provide access to 203 real-world APIs across 23 tools and a variety of multi-step tool-use tasks that require complex agent-tool interactions, perfectly aligning with our goals.

We have curated error-containing data while observing LLMs' behavioral patterns across these benchmarks in § 2.1, but it is far from sufficient. To facilitate more controlled error data generation, we first collect the ground truth tool-calling trajectories including tool call actions and the corresponding tool responses across various tasks in these datasets. Any data containing errors, such as incorrect annotations or failed tool calls, is carefully manually filtered to ensure the quality and reliability of our dataset. Next, we extract API documentation and refine any ambiguous or inadequate descriptions to ensure clarity and precision, minimizing potential misunderstandings. To further enhance consistency, we standardize all tool-calling trajectories and API descriptions, which aligns formats across different benchmarks, creating a coherent framework that facilitates consistent prompts and reliable tool-use interactions throughout our evaluation.

#### 2.2.3 Error Diversification

We have identified five patterns of errors from two sources in § 2.2.1. To ensure the comprehensive coverage of potential scenarios, we systematically diversify these errors, significantly expanding our error repository.

• **Internal Model-Driven Errors**: The internal model-driven error data collected from previous observation has two limitations that (1) it comes from a small subset of tools and tasks, and (2) the tests primarily involve advanced LLMs, which restricts the coverage of errors that less capable models might produce. Moreover, our observation reveals that LLMs tend to exhibit similar behaviors within a specific error pattern, despite interacting with different tools. This similarity allows us to expand the diversity of errors in the calling of all tools. We prompt GPT-4o as an error simulator, simulating error-prone behaviors of tool-calling assistants. Using examples of error patterns collected from observation as few-shot demonstrations (Brown et al., 2020), error simulator is tasked with generating diverse instances of errors across a wider range of tools and tasks.

• **External Environment Errors**: During data collection, we capture numerous instances of tool responses containing external environment errors and match them with their corresponding tools. However, not all tools in the benchmark datasets include such error examples. To fill this gap, we perform repeated calls to the accessible APIs to collect the error responses arising from environmental instability, and employ GPT-4o as an API simulator to collect such errors for inaccessible APIs.

#### 2.2.4 Tool Response Handling

The responses LLMs receive from the environments during tool calls are crucial for them to self-criticize, making it essential to obtain tool responses corresponding to internal model-driven errors. However, due to permission restrictions, not all collected APIs are executable. Inspired by StableToolBench (Guo et al., 2024a), we adopt a systematic approach for tool response collection based on the availability status of each API.

• **Cache Retrieval**: We first search the cache to check whether the tool and parameters used in the current call have previously been cached. If a match is found, the cached response is used as the environment's response for the current tool call.

• **API Execution**: If there is no match in the cache, we then verify the accessibility of API. The tool call is executed and the actual API response is used if the API is available.

• **Simulator Response**: When neither cache nor API is available, we employ GPT-4o as an API simulator to ensure that the tool-calling assistant still receives feedback for its current action.

#### 2.2.5 Data Evolution

Real-world tool calls typically encompass complex contexts, sophisticated tools, and ambiguous user queries (Wang et al., 2024b). To achieve a more realistic evaluation of LLM performance in tool call tasks, we propose a strategy termed Scalable and Robust Mixed Self-Evolution (SRM) to facilitate the self-evolution of data within the origin benchmark. Specifically, we focus on two critical factors of tool-use tasks: scale and robustness. Based on these factors, we develop four distinct evolutionary sub-strategies on these perspectives that closely align LLM tool-use tasks with real-world scenarios while preserving the ground truth annotations.

• **Long Context:** We introduce extended conversations from LongBench (Bai et al., 2023), mix it with tool-calling data randomly as the context, and insert them prior to the user's tool-use query.

• **Extra Tools:** Most existing benchmarks merely supply the tools required for specific test tasks, which contrasts sharply with the vast number of

APIs involved in real applications. Thus, we propose the Extra Tools evolution strategy, which randomly incorporates additional tools into API lists.
• **Noisy Query:** Real user queries are often verbose, vague, include unnecessary information, and are prone to typographical errors, which challenge LLMs' ability to interpret intent. We employ GPT-4o to simulate human language habits, particular focusing on addressing irrelevant information, cumbersome expressions, and typographical issues.
• **Harder Tools:** DRAFT (Qu et al., 2024) and BFCL v2 (Yan et al., 2024) illustrate the substantial impact that API documentation has on LLM tool calls. Therefore, we deliberately degrade the API document by prompting GPT-4o, thereby making the idealized APIs documentation more realistic.

We combine the four evolutionary sub-strategies to increase the difficulty of LLM tool-use tasks, involving three key components: context, queries, and the API list, enabling the exploration of scalability and robustness in self-critique.

After the SRM process, we verify the data to ensure that the ground truth remains unchanged. To prevent inappropriate self-critique behavior arises from biases by the evolutionary strategies, we introduce equivalence verification, a novel data verification approach. We use GPT-4o to check whether the modifications or additions made during the evolution process significantly impact the tool-use tasks (refer to Appendix C.2).

#### 2.2.6 Dataset Summary

We apply a dual filtering mechanism to all generated error data, combining automated and manual processes to mitigate potential biases (as discussed in Appendix C.3), resulting in a pass rate of 18.63%. The final CRITICTOOL dataset consists of 1,490 base examples and 1,250 evolved examples. More detailed statistics are provided in the Appendix B.2.

### 2.3 Fine-Grained Evaluation

CRITICTOOL comprehensively evaluates the self-critique capabilities of LLMs by breaking them down into multiple dimensions, across different error patterns encountered during tool interaction.

#### 2.3.1 Self-Critique Task Decomposition

In CRITICTOOL, each tool-use task is defined as a tuple $(Q, T)$, where $Q$ is the task query, and $T$ represents the list of APIs available for the tool-calling assistant. We define the trajectory $\mathcal{T}$ as a sequence of tool-response pairs $\{(a_i, r_i)\}$, capturing the interaction between the assistant's action $a$ and the corresponding tool response $r$ in the $i$-th step. The action $a$ is regarded as either $(goal, tool, args)$ or $(tool, args)$ depending on whether the chain-of-thought strategy is applied.

The complex interactions between the assistant and the environment can lead to potential errors at any step, underscoring the importance of evaluating LLMs' self-critique capabilities at the step level (Ye et al., 2024b). Consequently, the test data consists of the first $k$ steps of the tool-calling trajectory for each task, where $k$ is randomly chosen, and any errors may be introduced at step $k$.

In internal model-driven errors critique tasks, CRITICTOOL employs both error-free and error-injected data to ensure fairness and robustness. We evaluate the $(k+1)$-th step and deconstruct the self-critique process into two dimensions. The tool-calling assistant should recognize whether an error occurred during the preceding tool call first and identify its specific category. This process of identifying and analyzing errors is defined as **reflect**, a fundamental step in the model's self-critique. Based on the result of the reflection, the model needs to take corrective action to recover from the error. We define this process as **correct**, highlighting the model's ability to improve and adapt its behavior effectively. Thus, the solution path is $S = (c, \hat{a})$ or $S = (\hat{a})$, where $c$ represents the reflect of the error when the model identify it.

For tasks involving external environment errors, the assistant is expected to properly handle the response from the environment that contains the error signal in the subsequent steps. We encourage the assistant to **retry** the failed tool calls a limited number of times to avoid the incidental error caused by environmental instability. If the issue persists despite multiple retries, the assistant should **skip** the problematic step and address any remaining feasible subtasks or **finish** the tool-calling process and inform the user that further guidance is required. The solution path is defined as a sequence of actions $S = \{\hat{a}_1, \hat{a}_2, \dots\}$.

#### 2.3.2 Evaluation Metrics

CRITICTOOL employs fine-grained evaluation metrics to assess each dimension of self-critique behavior of LLMs across different error scenarios. The details are provided in Appendix C.4.
• **REFLECT:** The reflect evaluator asks the assistant to determine whether to produce a critique $c^{pred}$, based on the correctness of tool call action

Table 2: **Main Results of CRITICTOOL. Bold** indicates the best performance across all models, while underline denotes the best performance within the same group and scale of models.

| Models | Internal Model-Driven Errors | | | | External Environment Errors | | | | Overall |
|---|---|---|---|---|---|---|---|---|---|
| | Reflect | | Correct | | Retry | Skip/Finish | | | |
| | Detect | Category | Tool | Args | | Break | Tool | Args | |
| ***Closed-Source Large Language Models*** | | | | | | | | | |
| Claude3.5 | 81.59 | 55.70 | 84.89 | 77.63 | 38.22 | 56.27 | 22.06 | 26.48 | 55.83 |
| GPT-3.5 | 71.18 | 62.90 | 71.36 | 58.09 | 10.37 | 89.45 | 52.23 | 41.27 | 60.93 |
| GPT-4o | 78.71 | **69.70** | **86.05** | **80.25** | 20.99 | **92.08** | **53.66** | **42.67** | **69.01** |
| ***Open-Source Large Language Models*** | | | | | | | | | |
| LLaMA3-8B | 56.39 | 29.24 | 73.81 | 65.17 | 31.81 | 74.67 | 27.11 | 29.95 | 50.84 |
| LLaMA3.1-8B | 83.77 | 68.09 | 78.26 | 69.11 | 50.94 | 73.58 | 25.00 | 22.10 | 58.04 |
| Qwen2.5-7B | 82.86 | 44.21 | 77.32 | 69.26 | 28.41 | 83.06 | 42.28 | 24.08 | 58.61 |
| GLM4 - 9B - chat | 56.12 | 24.01 | 59.03 | 48.56 | 17.89 | 89.23 | 35.11 | 22.05 | 47.57 |
| Ministral - 8B | 46.15 | 23.45 | 67.23 | 57.12 | 50.11 | 59.03 | 17.02 | 20.11 | 43.77 |
| LLaMA3-70B | 56.11 | 29.37 | 69.13 | 62.61 | 32.29 | 73.18 | 27.66 | 27.52 | 49.25 |
| LLaMA3.1 - 70B | 79.52 | 59.78 | 82.34 | 65.47 | **63.12** | 91.23 | 51.58 | 25.89 | 65.21 |
| Qwen2.5-72B | **86.14** | 52.81 | 82.59 | 77.60 | 36.91 | 91.75 | 52.71 | 30.03 | 65.70 |
| ***Tool-Use-Finetuned Large Language Models*** | | | | | | | | | |
| ToolLLaMA2 - 7B | 0.58 | 0.00 | 3.34 | 0.61 | 0.92 | 1.77 | 0.91 | 0.00 | 0.13 |
| ToolACE - 8B | 12.98 | 0.95 | 14.23 | 13.22 | 1.25 | 8.23 | 7.67 | 12.21 | 9.43 |
| AgentLM-7B | 22.97 | 0.00 | 47.86 | 37.20 | 11.95 | 84.70 | 18.13 | 17.55 | 33.78 |

$a_k$. Then, $c^{pred}$ is compared with the golden answer $c^{gt}$ if an error exists in $a_k$.

• **CORRECT:** The correct evaluator asks the assistant to generate a corrected action $\hat{a}^{pred}$ for a detected error in tool call action $a_k$, and compares $\hat{a}^{pred}$ with the golden answer $\hat{a}^{gt}$.

• **RETRY:** The assistant is asked to generate a repeated tool call $\hat{a}_1^{pred}$ if any error signal is found in $r_k$. The evaluator compares $\hat{a}_1^{pred}$ with the golden answer $\hat{a}_1^{gt}$, which corresponds to the action $a_k$.

• **SKIP:** If the error from the environment cannot be resolved within the retry limit, the assistant should skip and proceed with the next feasible subtask. The skip action $\hat{a}_n^{pred}$ is compared to the golden answer $\hat{a}_2^{gt}$, which indicates the ground truth action for the next subtask.

• **FINISH:** The evaluator checks whether the assistant terminates the tool call and waits for further instructions from the user after several unsuccessful attempts to resolve the environmental error.

• **OVERALL:** We calculate the overall score by weighing the self-critique dimensions based on their importance in completing a tool-calling task. The weight assigned to reflect is 0.2, to correct is 0.3, to retry is 0.05, and to skip/finish is 0.45.

# 3 Experiments

## 3.1 Experiment Setup

We conduct evaluations on CRITICTOOL using a diverse set of 14 LLMs, to establish a comprehensive self-critique benchmark for assessing the capabilities of current large language models. For closed-source LLMs, we select three prominent models: Claude3.5 (Anthropic, 2024) developed by Anthropic, alongside GPT-3.5 (OpenAI, 2022) and GPT-4o (Hurst et al., 2024) provided by OpenAI.[3] For open-source LLMs, we evaluate numerous models including LLaMA3, LLaMA3.1 (AI@Meta, 2024), Qwen2.5 (Team, 2024a,b), GLM4 (GLM et al., 2024), Ministral(AI, 2024). For tool-use-fineturned LLMs, we evaluate ToolLLaMA2 (Qin et al., 2023), ToolACE (Liu et al., 2024) and AgentLM (Zeng et al., 2023).

## 3.2 Benchmarking Results on CRITICTOOL

The detailed experimental results are shown in Tab. 2. Experiments using the chain-of-thought strategy (Wei et al., 2022) are also conducted, leading to improvements in LLMs' self-critique performance, with the results provided in the Appendix D.2. We analyze the benchmarking results by exploring the following four questions.

**Q1: Which Model is Better at Tool Self-Critique?**

GPT-4o leads in self-critique performance for tool-use error scenarios, achieving an impressive overall score of 69.01. Close behind, large-scale open-source models LLaMA3.1-70B and Qwen2.5-72B,

[3]The version for GPT-4o is `gpt-4o-2024-08-06`, for GPT-3.5 is `gpt-3.5-turbo-16k`, and for Claude3.5 is `claude-3-5-sonnet-20241022`.

deliver comparable scores, showcasing strong self-critique capabilities.

For internal model-driven errors, the closed-source models GPT-4o and Claude3.5 deliver comparable top performance, though Claude3.5 slightly underperforms in error categorization. In contrast, open-source models exhibit substantial variability in self-critique performance. While most open-source models significantly lag behind the closed-source models, highlighting a clear gap in their capabilities, LLaMA3.1 and Qwen2.5 stand out as notable exceptions. Their performance not only approaches but occasionally surpasses that of closed-source models. However, tool-use-fineturned models show disappointing results in handling internal errors. Except for AgentLM-8B, the other models exhibit almost no instruction-following or self-critique capabilities, which can be attributed to the damage to their generalization ability caused by fine-tuning on specific data.

For external environment errors, most models can recognize errors and avoid endless repetition, though Claude3.5 and Ministral-8B shows weaker performance in this regard, and some tool-use-finetuned models entirely lack this ability. When it comes to handling errors by either proceeding with subsequent tasks or finish tool call action, GPT-4o outperforms other models, with some large-scale open-source models achieving comparably strong performance.

**Q2: What is the Self-critique Performance of LLMs across Various Scenarios?**

In the internal critique task, models should proceed with subsequent tool-calling tasks within error-injected data. However, poor performance models tend to exhibit over-reflection, mistakenly classifying a correct step as an errors. For error-injected cases, models are expected to accurately reflect and correct the mistake it made in the previous step, but many models with limited critique capabilities fail in such task. In the tool selection error scenario, LLMs may select the wrong tool while still providing valid parameters, leading to silent errors without explicit signals from the environment (Sun et al., 2024), hindering models' error reflection. In such cases, the most frequently observed poor self-critique behaviors are correction without reflection or error Ignorance. In contrast, the other three internal error scenarios often trigger explicit error signals due to invalid tool inputs or parameters, aiding models in reflecting and achieving higher self-critique success rates. Nonetheless, weaker

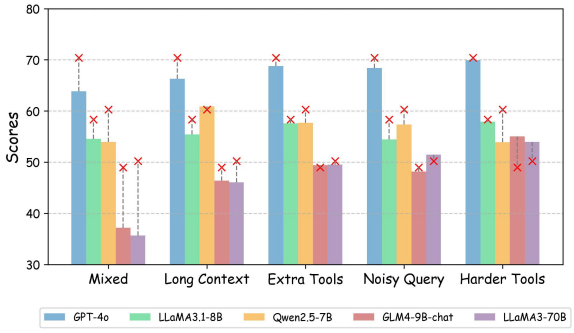


Figure 3: Comparison of the performance of five models across various evolution strategies. The red cross indicates the score corresponding to the base dataset.

models may still display failure to detect, failure to correct, or even experience unexpected tool call interruptions.

In the external critique task, the model should retry the failed operation retry within limits, exit the loop appropriately, and either complete the remaining subtasks or ask user for guidance. However, when models fail to recognize errors, they tend to repeat the same call more than three times, resulting in a significant resource drain. Some models go further by hallucinating, offering false answers to user questions rather than asking for guidance.

**Q3: How does Data Evolution Affect Model Performance?**

As illustrated in Fig. 3, the data evolution leads to a decline in the scores of all LLMs. GPT-4o retains its SOTA results, while Qwen2.5-7B also demonstrates impressive capabilities. In contrast, LLaMA3-70B experiences significant performance degradation, falling below the performance of most small scale models. This is consistent with CriticBench (Lin et al., 2024) experimental observation. We attribute this to the unstable generalizability of the offline data, a limitation that becomes increasingly pronounced as the number of model parameters grows. We independently test the four sub-strategies to investigate their impact on models' self-critic performance. The negative impact on the model decreases in the following order: Long Context, Noisy Query, Extra Tools and Harder Tools. Long Context and Extra Tools increase the difficulty of retrieval and challenge the model's ability to follow instructions and Extra Tools introduce relatively little extra data. Noisy Query presents a significant challenge to the model's capacity for comprehension and parameter transfer, reminiscent of the disruptive influence encapsulated by the adage 'A loose cannon'. However, as the API documents become more verbose and longer, some

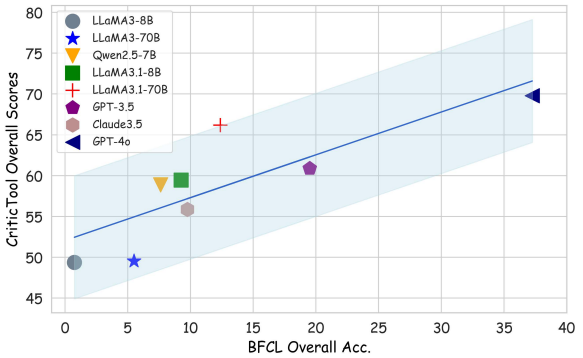


Figure 4: Comparison between BFCL Overall Accuracy and CRITICTOOL Overall Scores across several models. LLMs show similar trends in tool-use and self-critique capabilities.

models demonstrate improved comprehension of the APIs, leading to slight performance enhancements, such as GLM4-9B-chat.

Overall, for the model, the three key components—the context, query, and tool list—are not merely superimposed. The interplay between scalable and robust levels results in a compounding effect, causing the model's performance to degrade more rapidly under the hybrid strategy compared to individual strategies. The detailed results can be found in Appendix C.2.3.

**Q4: What is the Relationship Between Tool-Use and Self-Critique Capabilities?**

We compare the fine-grained evaluations on CRITICTOOL with the results of the benchmark designed to explore tool-use capabilities, investigating the relationship between models' self-critique capabilities in tool-calling tasks and their tool-use capabilities. We analyze the overall accuracy metric from tool-use benchmarks to examine the relationship between the tool-use performances of selected models and their Overall performance on CRITICTOOL. As results shown in Fig. 4, we observe a general alignment between the trends in models' tool-use and self-critique capabilities. This observation not only indicates a strong connection between models' ability to accurately use tools and their self-critique capabilities, suggesting that strengthening self-critique mechanisms could provide a promising avenue for enhancing overall tool-use performance, but also validates the rationale behind our benchmark.

## 4 Related Work

**Tool Learning with LLMs** There are currently two primary technical approaches for enhancing the tool invocation capability of LLMs (Shen et al., 2023; Yuan et al., 2024). The first approach focuses on constructing high-quality tool call data and improving the model's tool invocation capabilities through fine-tuning(Kong et al., 2024; Chen et al., 2024a; Patil et al., 2024). The second approach involves leveraging contextual tool call demonstrations to augment the model's ability to invoke tools through in-context learning (Wang et al., 2024a).

The evaluation of tool invocation capabilities across different models is also an urgent issue. Common evaluation frameworks involve comparing model predictions to ground truth (Yan et al., 2024; Guo et al., 2024b), while ToolBench (Qin et al., 2023) contrasts model predictions with those generated by advanced LLMs, such as GPT-4. Although some studies (Yan et al., 2024; Yao et al., 2024; Sun et al., 2024) have identified common errors in tool invocations, they unfortunately lack in-depth analysis and the design of targeted evaluation frameworks. In contrast to the aforementioned benchmarks, CRITICTOOL is the first to analyze various errors and evaluate the self-critic ability in tool invocation as far as we know.

**Self-Critique of LLMs** Learning from incorrect attempts can help prevent similar errors, thereby enabling deeper insights into the data and facilitating self-learning (Ke et al., 2024; Shinn et al., 2023; An et al., 2023; Ying et al., 2024; Zhang et al., 2024; Tian et al., 2024). CriticEval (Lan et al., 2024) evaluate the self-critique ability of LLMs on nine key tasks, including math and code, across four critical dimensions. For tool calls, the self-critic strategy is particularly well-suited for this complex task, which integrates various important capabilities on massive and constantly updated tools (Gou et al., 2023). However, to the best of our knowledge, no prior work has specifically explored the evaluation of self-critique in tool invocations. Recognizing the unique characteristics of tool calls compared to other tasks, CRITICTOOL adopts a targeted and fine-grained evaluation framework.

## 5 Conclusion

In this paper, we propose CRITICTOOL, the first benchmark for tool self-critique in LLM tool evaluation as far as we know. CRITICTOOL explicitly distinguishes between internal model errors and external environment errors, classifies evaluation methods, and employs data evolution strategies to uncover the true capabilities of the models under

evaluation. This evaluation offers a comprehensive analysis and identifies the primary bottlenecks in current LLMs' tool learning, providing valuable insights for the future development of tool agents.

## Limitations

While CRITICTOOL offers the first fine-grained and comprehensive evaluation of tool invocation self-criticism, as far as we know, it still has the following two limitations. (1) Our dataset builds upon and extends BFCL and T-eval. Despite refinement and filtering, the quality of the underlying dataset still impacts the overall quality and discriminative power of CRITICTOOL to some extent. (2) The construction of our benchmark relies on GPT-4o for error generation, evolution, and verification. The synthetic data may inevitably introduce biases inherent to GPT-4o. However, CRITICTOOL has employed multiple strategies in its data construction pipeline to mitigate these biases, ensuring high data quality and a reliable benchmark. Moreover, the dependence on high-performance LLM results in significant economic costs, posing challenges to the sustainability of large-scale benchmark development.

Future work should tackle these challenges by developing more rational and cost-effective data construction methods.

## Acknowledgements

This work was supported by the Anhui Provincial Natural Science Foundation under Grant 2108085UD12; the National Natural Science Foundation of China (NSFC) under Grants No. 62471445 and No. 62201526. We acknowledge the support of GPU cluster built by MCC Lab of Information Science and Technology Institution, USTC. The AI-driven experiments, simulations and model training were performed on the robotic AI-Scientist platform of Chinese Academy of Sciences.

## References


Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. 2024. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1131–1139.

Mistral AI. 2024. Un ministral, des ministraux.

AI@Meta. 2024. Llama 3 model card.

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2023. Learning from mistakes makes llm better reasoner. *arXiv preprint arXiv:2310.20689*.

Anthropic. 2024. Claude 3.5 sonnet.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Kinjal Basu, Ibrahim Abdelaziz, Kelsey Bradford, Maxwell Crouse, Kiran Kate, Sadhana Kumaravel, Saurabh Goyal, Asim Munawar, Yara Rizk, Xin Wang, et al. 2024. Nestful: A benchmark for evaluating llms on nested sequences of api calls. *arXiv preprint arXiv:2409.03797*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Lin Chen, Jisong Li, Xiaoyi Dong, Pan Zhang, Conghui He, Jiaqi Wang, Feng Zhao, and Dahua Lin. 2023. Sharegpt4v: Improving large multimodal models with better captions. *arXiv preprint arXiv:2311.12793*.

Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. 2024a. Advancing tool-augmented large language models: Integrating insights from errors in inference trees. *arXiv preprint arXiv:2406.07115*.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. 2024b. 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*, pages 9510–9529.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024c. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. In *Findings of the Association for Computational Linguistics*, pages 9354–9366.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

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

Zishan Guo, Yufei Huang, and Deyi Xiong. 2024b. CToolEval: A Chinese benchmark for LLM-powered agent evaluation in real-world API interactions. In *Findings of the Association for Computational Linguistics*, pages 15711–15724.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Pei Ke, Bosi Wen, Andrew Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Shengyuan Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, Jie Tang, and Minlie Huang. 2024. CritiqueLLM: Towards an informative critique generation model for evaluation of large language model generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 13034–13054.

Shirley Kokane, Ming Zhu, Tulika Awalgaonkar, Jianguo Zhang, Thai Hoang, Akshara Prabhakar, Zuxin Liu, Tian Lan, Liangwei Yang, Juntao Tan, et al. 2024. Spectool: A benchmark for characterizing errors in tool-use llms. *arXiv preprint arXiv:2411.13547*.

Yilun Kong, Jingqing Ruan, YiHong Chen, Bin Zhang, Tianpeng Bao, Shi Shiwei, du Guo Qing, Xiaoru Hu, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, and Xueqian Wang. 2024. TPTU-v2: Boosting task planning and tool usage of large language model-based agents in real-world industry systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 371–385.

Tian Lan, Wenwei Zhang, Chen Xu, Heyan Huang, Dahua Lin, Kai Chen, and Xian-ling Mao. 2024. Criticeval: Evaluating large language models as critic. *arXiv preprint arXiv:2402.13764*.

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

Zicheng Lin, Zhibin Gou, Tian Liang, Ruilin Luo, Haowei Liu, and Yujiu Yang. 2024. CriticBench: Benchmarking LLMs for critique-correct reasoning. In *Findings of the Association for Computational Linguistics*, pages 1552–1587.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. 2024. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. Llm critics help catch llm bugs. *arXiv preprint arXiv:2407.00215*.

OpenAI. 2022. Introducing chatgpt.

OpenAI. 2024. Introducing openai o1.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565.

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

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. From exploration to mastery: Enabling llms to master tools via self-driven interactions. *arXiv preprint arXiv:2410.08197*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2023. Taskbench: Benchmarking large language models for task automation. *arXiv preprint arXiv:2311.18760*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. 2024. Trial and error: Exploration-based trajectory optimization of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 7584–7600.

Jimin Sun, So Yeon Min, Yingshan Chang, and Yonatan Bisk. 2024. Tools fail: Detecting silent errors in faulty tools. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14272–14289.

Qwen Team. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Qwen Team. 2024b. Qwen2.5: A party of foundation models.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. 2024. Toward self-improvement of llms via imagination, searching, and criticizing. *arXiv preprint arXiv:2404.12253*.

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*, pages 10583–10604.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Zhongyu Wei, and Xuanjing Huang. 2024b. Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation. *arXiv preprint arXiv:2402.11443*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

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

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard.

Jihan Yao, Wenxuan Ding, Shangbin Feng, Lucy Lu Wang, and Yulia Tsvetkov. 2024. Varying shades of wrong: Aligning llms with wrong answers only. *arXiv preprint arXiv:2410.11055*.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Qi Zhang, Tao Gui, et al. 2024a. Tooleyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. *arXiv preprint arXiv:2401.00741*.

Junjie Ye, Yilong Wu, Songyang Gao, Caishuang Huang, Sixian Li, Guanyu Li, Xiaoran Fan, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024b. RoTBench: A multi-level benchmark for evaluating the robustness of large language models in tool learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 313–333.

Jiahao Ying, Mingbao Lin, Yixin Cao, Wei Tang, Bo Wang, Qianru Sun, Xuanjing Huang, and Shuicheng Yan. 2024. LLMs-as-instructors: Learning from errors toward automating model improvement. In *Findings of the Association for Computational Linguistics*, pages 11185–11208.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. 2024. Easytool: Enhancing llm-based agents with concise tool instruction. *arXiv preprint arXiv:2401.06201*.

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

Wenqi Zhang, Yongliang Shen, Linjuan Wu, Qiuying Peng, Jun Wang, Yueting Zhuang, and Weiming Lu. 2024. Self-contrast: Better reflection through inconsistent solving perspectives. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 3602–3622.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

# Appendix

## A Observation: Insight into LLMs' Tool-Use Performance

In § 2.1, we test BFCL v3 (Yan et al., 2024), T-Eval (Chen et al., 2024b), API-Bank (Li et al., 2023), and NESTFUL (Basu et al., 2024) to conduct an in-depth analysis of LLMs' behavioral patterns. The details of these benchmarks are provided below.

**BFCL V3** is a comprehensive benchmark for evaluating LLMs' performance in multi-step and multi-turn tool calling. The benchmark includes 200 basic tool-use trajectories, along with an additional 800 trajectories that introduce various complexities built upon these basic data.

**T-Eval** provides 553 tool-use trajectories, breaking down tasks into sub-processes including instruction following, planning, reasoning, retrieval, understanding, and review.

**API-bank** has 314 tool-use trajectories to evaluate LLMs' capabilities in planning, retrieving, and calling APIs.

**NESTFUL** is designed to better evaluate LLMs on nested sequences of tool calls. It compiles 85 executable tool-use traces and 215 non-executable traces from the different datasets, as well as synthetic data generated by LLMs.

We first observe that the prompts and tool-call formats used in these benchmarks varied, which could lead to discrepancies in how LLMs follow instructions. To address this, we standardize the test data into a consistent format, as Fig. 10, ensuring LLMs execute tasks sequentially and consistently across benchmarks. Then, we randomly select a subset of the test data from these benchmarks and summarize the frequently occurring error patterns in the test results. The distribution of error patterns is shown in Tab. 3.

In the experiment, we observe LLMs' performance in the presence of errors, and gain insight into their different behavior across different errors, as shown in Fig. 11 and 12. When LLMs continue executing tool-use tasks after making mistakes, we find that some of them could recognize and correct their mistakes, while most perform poorly. In cases where tool responses contain errors due to instability, many LLMs become trapped in repetitive retry loops, with few capable of recognizing the issue and breaking free by either skipping the current step or terminating the task.

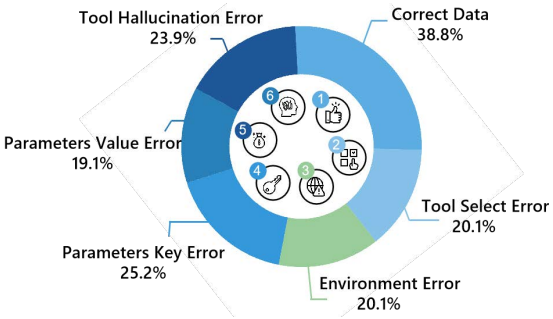


Figure 5: Error distribution for Base data in CRITICTOOL.

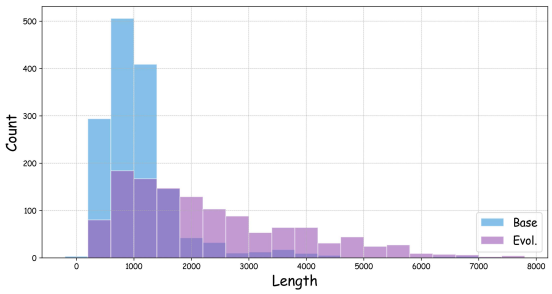


Figure 6: Length distribution for Base and Evolution data in CRITICTOOL, measured by the number of tokens.

## B CRITICTOOL Benchmark Details

### B.1 Comparison

Tab. 4 shows how CRITICTOOL compares against existing tool-use and critic benchmarks.

### B.2 Dataset Summary

The base dataset of CRITICTOOL originates from 733 high-quality tool-call trajectories, consisting of 1490 test cases in total, which contains 1316 internal model-driven error test cases and 174 external environment error test cases. On this basis, we retain the error distribution on the base data and randomly select to construct CRITICTOOL evolution dataset (be simplified to Evol.), generating 1000 internal and 250 external new test cases. We visualize the error distribution and length distribution for the base and evolved datasets.

Fig. 5 illustrates the error distribution of CRITICTOOL, which comprehensively covers the behavior patterns of LLMs observed across mainstream benchmarks.

Fig. 6 shows that each set of the base benchmark has 1291 tokens on average, while each evolved examples contains 2387 tokens on average, validating the generalization and discrimination for tool utilization self-critic evaluation.

Table 3: Error distribution among LLMs in tool-use benchmarks.

| Benchmark | Model | Total | Tool Sel. | Tool Halluc. | Param. Key | Param. Value |
|---|---|---|---|---|---|---|
| BFCL V3 | Qwen-turbo | 184 | 82 | 1 | 0 | 13 |
| | Qwen2.5-72B | 216 | 74 | 0 | 0 | 12 |
| | GPT-3.5 | 202 | 85 | 0 | 0 | 13 |
| | GPT-4o | 213 | 70 | 0 | 0 | 6 |
| T-Eval | Qwen-turbo | 452 | 36 | 3 | 4 | 36 |
| | Qwen2.5-72B | 469 | 29 | 1 | 1 | 28 |
| | GPT-3.5 | 466 | 38 | 13 | 10 | 29 |
| | GPT-4o | 470 | 29 | 0 | 0 | 23 |
| API-bank | Qwen-turbo | 259 | 2 | 1 | 0 | 13 |
| | Qwen2.5-72B | 184 | 82 | 2 | 0 | 19 |
| | GPT-3.5 | 275 | 6 | 1 | 1 | 18 |
| | GPT-4o | 280 | 6 | 0 | 1 | 10 |
| NESTFUL | Qwen-turbo | 215 | 9 | 1 | 27 | 29 |
| | Qwen2.5-72B | 212 | 22 | 3 | 23 | 26 |
| | GPT-3.5 | 215 | 13 | 22 | 20 | 22 |
| | GPT-4o | 215 | 4 | 10 | 7 | 14 |

Table 4: Comparison of CRITICTOOL with other existing tool-use and critique benchmarks.

| Model | Critic for Error | Function Call | API Response | Multi-Step | Fine-Grained Eval | Data by Difficulty Levels |
|---|---|---|---|---|---|---|
| **CriticBench** (Lin et al., 2024) | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| **CriticEval** (Lan et al., 2024) | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| **API-Bank** (Li et al., 2023) | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |
| **BFCL** (Yan et al., 2024) | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| **NestFul** (Basu et al., 2024) | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ |
| **T-Eval** (Chen et al., 2024b) | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **CRITICTOOL** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

# C Implementation Details

## C.1 Data Collection

We collect 733 ground truth tool-calling trajectories from high-quality tool-use benchmarks, BFCL (Yan et al., 2024) and T-Eval (Chen et al., 2024b). To facilitate following controlled error data generation, we manually filter out 485 trajectories that contain no errors and refine the API documentation to ensure that all API descriptions are clear and accurate. To bridge the gap between different instruction formats, we standardize both the trajectories and API documentation, as illustrated in Fig. 13 and 14. This standardization ensures compatibility and reduces variability in the data, enabling a more consistent evaluation of LLMs' performance in self-critique capabilities.

## C.2 Prompts Demonstration

Refer to the corresponding prompt block for a detailed demonstration.

### C.2.1 Error Data Diversification

We prompt GPT-4o as error simulator, and the corresponding prompt is presented in Fig. 15.

### C.2.2 Tool Responses Generation

We prompt GPT-4o as API simulator, and the corresponding prompt is presented in Fig. 16.

### C.2.3 Data Evolution

The framework of the data evolution has been shown in Fig. 7. And Tab. 5, presents a simplified example of our Scalable and Robust Mixed Self-Evolution (SRM) evolution strategy.

**Noisy Query**: We prompt GPT-4o to refine the user query, and the corresponding prompt is presented in Fig. 20.

**Harder Tools**: We prompt GPT-4o to downgrade the API documentation, and the corresponding prompt is presented in Fig. 21.

**Mixed Evolution**: In mixed evolution, we randomly we randomly select 2-4 evolution strategies for each case.

**Data Verification**: We prompt GPT-4o to verify the evolution data, and the corresponding prompt is presented in Fig. 22, 23, 24, 25.

Table 5: A simplified example of our data evolution strategy.

**Original Tool Call Trajectory**

**Context:** None.
**Tool List**: 'name': 'Email.send', 'description': 'Sends an email to a specified recipient with the given subject and content.'
**User Query**: Compose an email to all team members at `team_members@example.com` detailing the features of the forthcoming film, 'Avengers: Endgame'. Subsequently, ascertain the availability of the first available meeting room from 2:00 PM to 4:00 PM and book it for our weekly marketing assembly.

| Perspective | Sub-strategy | Changed Items | Examples |
|---|---|---|---|
| Scalable | Long Context | Context | **Insert Context 1:** [A summary task of about 800 tokens.]<br>**Insert Context 2:** [A former Tool-Calling Task of about 400 tokens] |
| | Extra Tools | Tool List | **Add Tools:** Email.show, Email.check, Email.read, ArxivSearch.get_arxiv_information, BINGMap.search_nearby... |
| Robust | Noisy Query | User Query | **Refine Query:** My favourite film is *Avengers: Endgame*, I want to share it to my team members. Compose an emaail (typo, email) to all tam nembers (typo, team members) at `team_members@example.com` detailing the features of the forthcoming film *Avengers: Endgame*, including its plot, main characters, and key action sequences. You can also mention how the movie fits into the Marvel Cinematic Universe and its expected impact on upcoming releases. Following that, ascertain the availability of the first available meeting room from 2:00 PM to 4:00 PM and book it for our weekly marketing assembly. Additionally, weekly marketing assembly is very important. So please confirm the booking once it's done. |
| | Harder Tools | Tool List | **Refine API Document:** send a email |

### C.3 Mitigating Bias in Synthetic Error Data

To scale the data and conduct comprehensive evaluations, we utilize GPT-4o for data synthesis during both the generation and evolution phases. Inevitably, such synthetic data may inherit biases from the generating model, including: (1) implausible scenarios that are meaningless (e.g., asking a weather API for stock prices); (2) oversimplified error logic that is unrepresentative of realistic failures (e.g., simple parameter typos); (3) a lack of error diversity, with an over-representation of specific error (e.g., most of parameters value error being an integer passed as a string); (4) intent drift in evolved queries. Therefore, we implemented a rigorous filtering process to minimize these biases.

To further address these issues, we design a dedicated pipeline that mitigates such biases through a rigorous two-stage filtering process. First, automated filtering removes problematic samples through regex-based syntax and pattern validation, keyword screening for meaningless queries, and an LLM-based detector for semantic intent shifts. Subsequently, manual review targets these issues to discard unreliable cases and ensure that the final dataset aligns with realistic task scenarios, yielding a pass rate of 18.63

### C.4 Detailed Evaluation Metrics

In the CRITICTOOL, self-critique capabilities are divided into multiple dimensions based on errors from different sources: Reflect, Correct, Retry, and Skip/Finish. All responses must strictly adhere to the JSON format.

We have defined the formalization of tool calls in § 2.3: each tool-calling task is represented as a tuple $(Q, T)$, where $Q$ is the query associated with the task, and $T$ denotes the list of tools that the assistant can utilize. The tool-calling trajectory $\mathcal{T}$ is a sequence of tool-response pairs $\{(a_i, r_i)\}$, which capture the interaction between the assistant's actions $a$ and the corresponding tool responses $r$ in the $i$-th step. The action $a$ is regarded as either $(goal, tool, args)$ or $(tool, args)$ depending on whether the chain-of-thought (CoT) strategy is used. The test data consists of the first $k$ steps of the tool-calling trajectory for each task, where $k$ is randomly selected, and errors may be introduced at step $k$.

In an internal model-driven error task, given a tool list $T$, query $Q$, a tool-calling trajectory $\mathcal{T} = \{(a_1, r_1) \dots (a_k, r_k)\}$, and an error may be contained in $a_k$. The assistant is asked to generate solution $S^{pred} = (c^{pred}, \hat{a}^{pred})$ if it identifies an error in $a_k$, and $S^{pred} = (\hat{a}^{pred})$ otherwise. The golden solution is $S^{gt} = \{\hat{a}_1^{gt}, \hat{a}_2^{gt}\}$, where $\hat{a}_1^{gt} = a_k$ and $\hat{a}_2^{gt}$ is the ground truth action for next subtask.

In the case of external environment error, given a tool list $T$, query $Q$, and a tool-calling trajectory $\mathcal{T} = \{(a_1, r_1) \dots (a_k, r_k)\}$, where an exter-

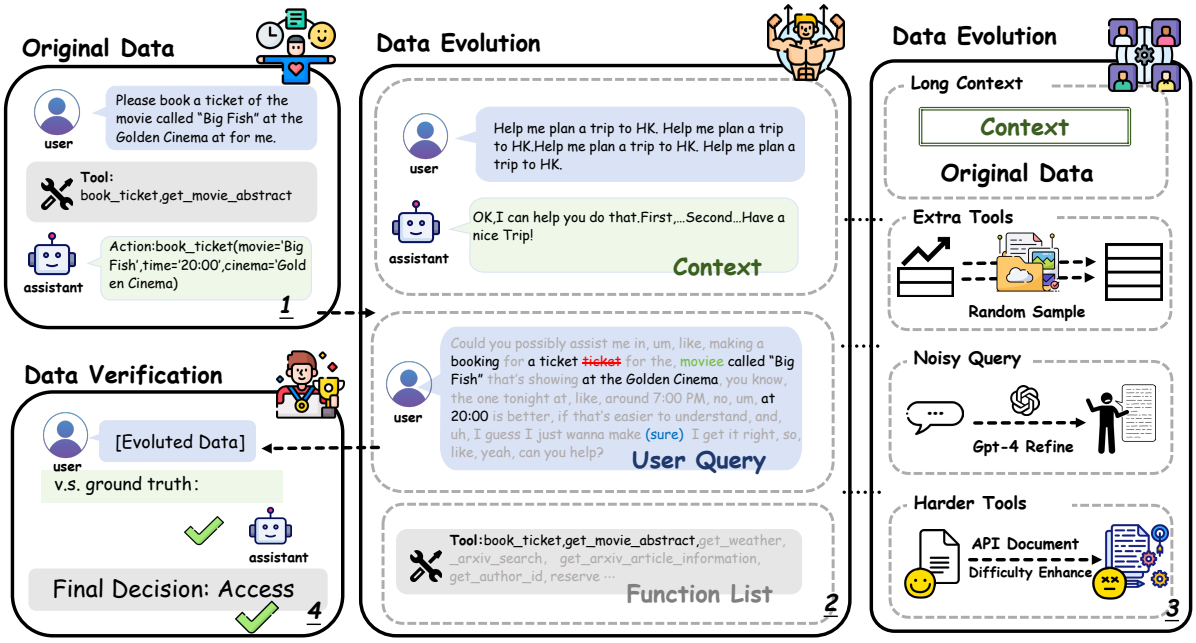


Figure 7: The framework of **Scalable and Robust Mixed Self-Evolution (SRM).**

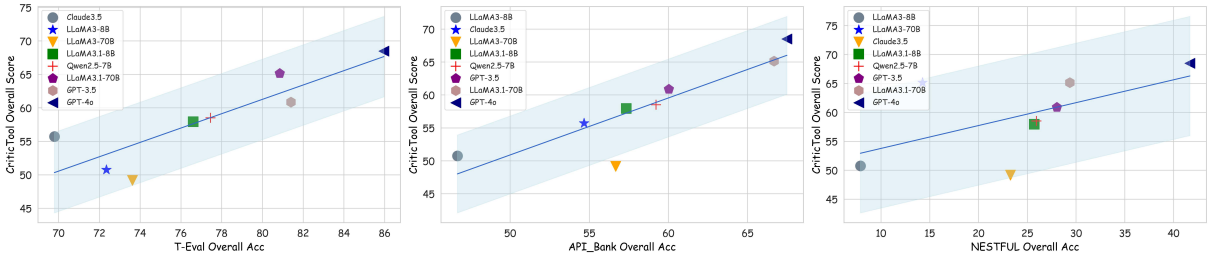


Figure 8: Comparison of CRITICTOOL Overall Scores with tool-use benchmarks' Overall Accuracy across several models.

nal error occurs in $r_k$. The assistant is tasked with retrying the action $a_k$ no more than three times, then break free from the loop and either proceed with executing the next subtasks or finish the tool call. If the predicted action $\hat{a} = a_k$, we return the erroneous response $r_k$ to allow the assistant to proceed. Once $\hat{a} \neq a_k$ is detected, or if more than three steps are executed, we stop the assistant's reasoning and obtain a sequence of predicted solution $S^{pred} = \{\hat{a}_1^{pred}, \hat{a}_2^{pred}, \ldots\}$. The golden solution is $S^{gt} = \{\hat{a}_1^{gt}, \hat{a}_2^{gt}\}$, where $\hat{a}_1^{gt} = a_k$ and $\hat{a}_2^{gt}$ is the ground truth action for next subtask. The evaluation process is shown in the Fig. 9.

### C.4.1 REFLECT

The reflect evaluator measures the model's ability to recognize the errors in tool call trajectories. For error-free trajectory where solution path is $S^{gt} = (a^{gt})$, the evaluation focuses solely on detection accuracy. If LLM predicts $S^{pred} = (a^{pred})$, the detect score is 1; otherwise, it is 0. For error-injected trajectory where solution path is $S^{gt} = (c^{gt}, a^{gt})$, the detection score is 1 if $c^{pred}$ in prediction $S^{pred}$, and 0 otherwise. The evaluator then determines whether the predicted error category $c^{pred}$ matches the ground truth $c^{gt}$, achieving category score 1 if the same and 0 otherwise.

### C.4.2 CORRECT

The correct evaluator assesses the model's ability to correct its actions after making a mistake. For trajectories containing errors, the evaluator first verifies whether the predicted $tool^{pred}$ matches the golden answer $tool^{gt}$. If correct, the tool score is 1, and the evaluator proceeds to evaluate the correctness of the input parameters. Otherwise, both the tool and args scores are set to 0. Then, the evaluator checks whether the passed parameter keys are missing or redundant, and the args score is set to 0 if any discrepancy exists. For parameters with types such as 'string' or 'any', the evaluator uses Sentence-BERT (Reimers and Gurevych, 2019), which involves embedding the two sentences, to compute the cosine similarity

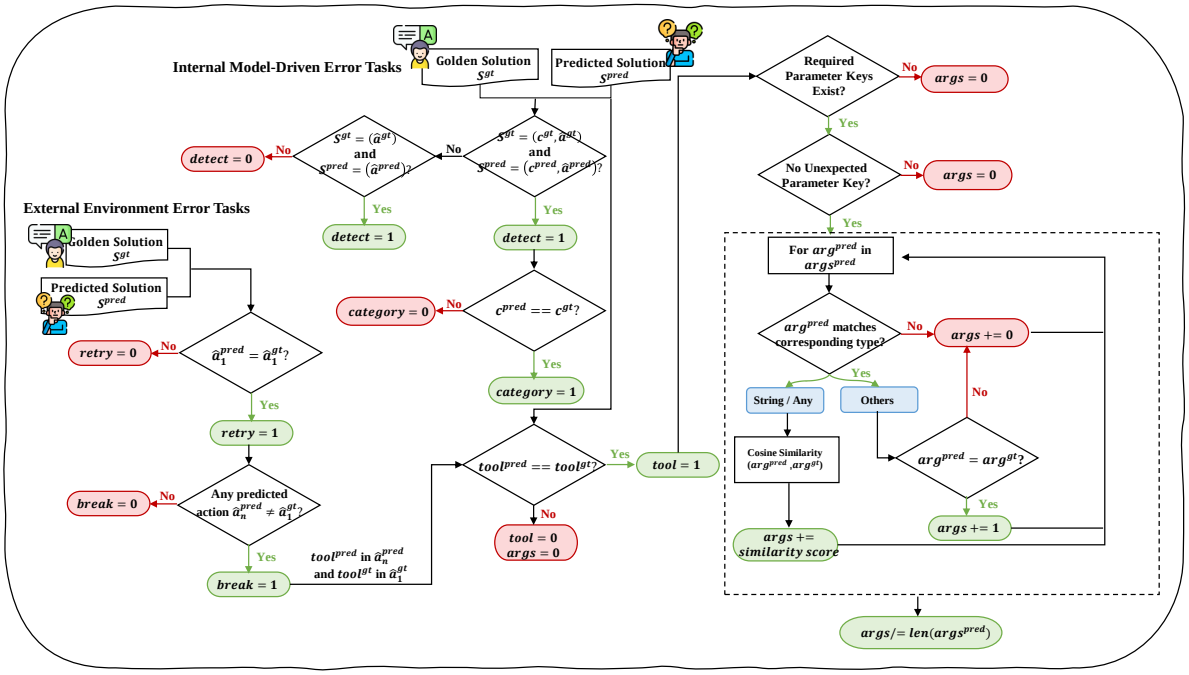


Figure 9: The framework of **Evaluation Process.**

between the embeddings of each predicted parameter value of $args^{pred}$ and the ground truth value $args^{gt}$ as their scores. The underlying BERT model used is `all-mpnet-base-v2`.[4] For all other parameter types, the predicted values must match the ground truth values exactly. Finally, the average score across all parameters is calculated as the args score. If the CoT strategy is applied, the evaluator uses Sentence-BERT to embed the predicted thought $thought^{pred}$ and the ground truth thought $thought^{gt}$, then calculates their cosine similarity as the thought score.

#### C.4.3 RETRY

The retry evaluator checks whether the predicted action $\hat{a}_1^{pred}$ is identical to the ground truth action $\hat{a}_1^{gt}$, the retry score is 1 if the same and 0 otherwise.

#### C.4.4 SKIP

The skip evaluator first examines all predicted actions to check if there exists any $\hat{a}^{pred} \neq \hat{a}_1^{gt}$, which indicates that the model has skipped the current retry step. If such a case $\hat{a}_n^{pred}$ is found, the break score then set to 1. The evaluator then compares the predicted action for next subtask $\hat{a}_n^{pred}$ with the golden answer $\hat{a}_1^{gt}$. The tool, args and thought score are determined using the same comparison method as in the correct evaluation.

[4] https://www.sbert.net/docs/pretrained_models.html

#### C.4.5 FINISH

The finish evaluator first evaluates the break score in the same manner as the skip evaluator. It then checks whether the break-free action $\hat{a}_n^{pred}$ is $FinishAction$. If so, the tool score is set to 1.

### C.5 Experimental Details

To evaluate the pure ability of the single model, we do not use any optimization methods in the main text, such as ReAct. To assess whether the model with optimization methods exhibits a distribution comparable to the original benchmark—including indicator scores and the model's relative strengths and weaknesses—we also generated CRITICTOOL with chain-of-thought (CoT). CRITICTOOL-CoT contains 810 internal model-driven error test cases and 126 external environment error test cases. Similarly, we use CRITICTOOL-CoT as the base and evolutionary dataset and obtain a total of 1,250 evolved test cases. Experimental results with CoT will be presented in the Appendix D.2.

## D Additional Results

### D.1 Full Results on CRITICTOOL

We show the full results on CRITICTOOLin Tab. 6.

### D.2 Full Results on CRITICTOOL-CoT

We show the full results on CRITICTOOL-CoT in Tab. 8.

Table 6: Results of CRITICTOOL on **Base** and **Evolutionary Datasets. Bold** indicates the best performance across all models, while underline denotes the best performance within the same group and scale of models.

| Models | Internal Model-Driven Errors | | | | | | | | External Environment Errors | | | | | | | | Overall | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Reflect | | | | Correct | | | | Retry | | Skip/Finish | | | | | | | |
| | Detect | | Category | | Tool | | Args | | | | Break | | Tool | | Args | | | |
| | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol |
| ***Closed-Source Large Language Models*** | | | | | | | | | | | | | | | | | | |
| Claude3.5 | 85.0 | 77.3 | 60.7 | 50.1 | 87.1 | 81.5 | 80.2 | 74.9 | 45.7 | 33.8 | 57.2 | 55.8 | 22.7 | 21.0 | 26.7 | 25.4 | 57.9 | 53.4 |
| GPT-3.5 | 73.3 | 70.1 | 61.3 | 62.9 | 72.0 | 70.4 | 58.6 | 55.7 | 12.6 | 8.4 | 92.5 | 86.0 | 54.6 | 51.0 | **46.4** | 35.3 | 62.7 | 58.9 |
| GPT-4o | 80.6 | 76.2 | **73.0** | 65.3 | **87.6** | **84.0** | **82.3** | **77.6** | 19.8 | 21.8 | 94.8 | 88.6 | 53.7 | **53.2** | 46.1 | **38.3** | **70.9** | **65.2** |
| ***Open-Source Large Language Models*** | | | | | | | | | | | | | | | | | | |
| LLaMA3-8B | 51.0 | 63.5 | 26.5 | 32.9 | 75.6 | 71.5 | 67.6 | 62.0 | 35.6 | 29.2 | 73.3 | 75.6 | 28.4 | 26.2 | 31.3 | 29.0 | 51.0 | 50.7 |
| LLaMA3.1-8B | 84.5 | **82.8** | 68.6 | **67.4** | 80.4 | 75.5 | 72.3 | 64.9 | 52.9 | 49.6 | 71.0 | 75.4 | 24.4 | 25.4 | 21.2 | 22.7 | 58.3 | 57.1 |
| Qwen2.5-7B | 85.1 | 79.9 | 43.1 | 45.6 | 79.6 | 74.4 | 72.1 | 65.5 | 34.2 | 24.4 | 87.6 | 79.9 | 46.0 | 39.7 | 19.7 | 27.2 | 60.3 | 56.8 |
| GLM4 - 9B - chat | 60.8 | 52.6 | 26.7 | 24.3 | 63.2 | 57.8 | 53.1 | 47.1 | 22.4 | 16.3 | 84.8 | 93.7 | 39.1 | 35.3 | 20.5 | 23.9 | 49.0 | 45.1 |
| Ministral - 8B | 47.0 | 50.2 | 23.8 | 29.7 | 70.6 | 67.2 | 61.4 | 55.8 | 56.0 | **48.4** | 58.0 | 64.1 | 20.4 | 18.3 | 28.1 | 17.2 | 45.7 | 42.0 |
| LLaMA3-70B | 61.4 | 49.1 | 33.7 | 23.6 | 72.6 | 64.6 | 66.5 | 57.4 | 37.0 | 29.0 | 58.8 | 83.2 | 30.9 | 25.4 | 30.2 | 25.7 | 50.2 | 47.0 |
| LLaMA3.1-70B | 83.6 | 78.2 | 64.3 | 57.6 | 84.4 | 81.6 | 69.3 | 64.6 | **71.8** | 59.9 | 85.6 | **98.1** | 53.7 | 50.1 | 31.0 | 25.1 | 67.0 | 64.7 |
| Qwen2.5-72B | **89.4** | 82.2 | 58.9 | 51.9 | 84.5 | 82.6 | 77.9 | 76.3 | 38.8 | 41.2 | **95.1** | 87.6 | **56.9** | 48.9 | 32.4 | 28.1 | 68.8 | 63.4 |
| ***Tool-Use-Finetuned Large Language Models*** | | | | | | | | | | | | | | | | | | |
| ToolLLaMA2-7B | 0.8 | 0.4 | 0.0 | 0.0 | 4.1 | 2.3 | 0.6 | 0.7 | 1.0 | 0.8 | 1.2 | 0.0 | 0.7 | 1.1 | 0.0 | 0.0 | 1.1 | 0.6 |
| ToolACE-8B | 12.8 | 13.8 | 0.9 | 1.0 | 14.5 | 14.9 | 13.2 | 13.2 | 1.4 | 1.1 | 13.2 | 3.8 | 6.9 | 7.4 | 10.9 | 13.5 | 10.3 | 9.2 |
| AgentLM-7B | 24.9 | 20.4 | 0.0 | 0.0 | 56.0 | 37.1 | 44.1 | 28.1 | 12.1 | 11.8 | 85.1 | 84.4 | 20.4 | 16.5 | 21.0 | 15.2 | 37.1 | 29.8 |

Table 7: **Self-Critique Evaluation** on different error patterns.

| Models | Tool Sel. Errors | | Tool Halluc. Errors | | Param. Key Errors | | Param. Value Errors | |
|---|---|---|---|---|---|---|---|---|
| | Reflect | Correct | Reflect | Correct | Reflect | Correct | Reflect | Correct |
| ***Closed-Source Large Language Models*** | | | | | | | | |
| Claude3.5 | 10.15 | 56.29 | 93.29 | 65.74 | 93.21 | 90.59 | 94.11 | 90.80 |
| GPT-3.5 | 7.32 | 32.81 | 80.10 | 27.89 | 82.65 | 79.07 | 86.96 | 66.28 |
| GPT-4o | 23.42 | 59.18 | 97.72 | 70.43 | 79.65 | 92.81 | 86.17 | 90.22 |
| ***Open-Source Large Language Models*** | | | | | | | | |
| LLaMA3-8B | 7.68 | 41.58 | 70.30 | 52.29 | 61.39 | 83.07 | 67.79 | 78.12 |
| LLaMA3.1-8B | 19.48 | 41.29 | 97.49 | 54.69 | 98.47 | 88.90 | 92.60 | 82.60 |
| Qwen2.5-7B | 28.14 | 37.61 | 96.51 | 57.68 | 97.40 | 85.96 | 93.38 | 85.25 |
| GLM4-9B-chat | 9.58 | 18.35 | 61.42 | 42.34 | 55.98 | 69.83 | 62.93 | 55.86 |
| Ministral-8B | 4.27 | 34.42 | 70.07 | 42.38 | 23.68 | 77.86 | 29.43 | 70.35 |
| LLaMA3-70B | 8.15 | 43.09 | 70.21 | 55.33 | 57.48 | 76.95 | 54.99 | 66.00 |
| LLaMA3.1-70B | 14.11 | 49.66 | 94.51 | 51.17 | 90.79 | 78.61 | 91.53 | 83.18 |
| Qwen2.5-72B | 36.92 | 55.91 | 94.03 | 59.34 | 95.37 | 91.08 | 97.03 | 93.73 |
| ***Tool-Use-Finetuned Large Language Models*** | | | | | | | | |
| ToolLLaMA2-7B | 0.29 | 0.00 | 0.76 | 0.00 | 0.30 | 0.93 | 1.00 | 1.65 |
| ToolACE-8B | 0.28 | 11.11 | 3.25 | 5.01 | 2.74 | 19.16 | 4.31 | 13.48 |
| AgentLM-7B | 0.56 | 20.70 | 1.26 | 22.83 | 0.30 | 50.62 | 0.68 | 40.53 |

Table 8: Results of CRITICTOOL-CoT on **Base** and **Evolutionary Datasets.**

| Models | Internal Model-Driven Errors | | | | | | | | External Environment Errors | | | | | | | | Overall | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Reflect | | | | Correct | | | | Retry | | Skip/Finish | | | | | | | |
| | Detect | | Category | | Tool | | Args | | | | Break | | Tool | | Args | | | |
| | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol | Base | Evol |
| ***Closed-Source Large Language Models*** | | | | | | | | | | | | | | | | | | |
| Claude3.5 | **91.7** | 83.2 | 71.2 | 57.5 | **90.7** | **86.3** | 83.8 | 79.1 | 37.3 | 26.4 | 94.4 | 67.5 | 36.9 | 24.7 | 51.4 | 36.5 | 71.8 | 59.3 |
| GPT-3.5 | 67.0 | 70.4 | 52.1 | 49.7 | 84.4 | 77.3 | 70.3 | 64.0 | 15.1 | 6.0 | 81.0 | 83.8 | 63.5 | 59.0 | 48.5 | 40.1 | 64.8 | 63.4 |
| GPT-4o | 91.4 | **88.3** | **86.5** | **82.5** | 90.4 | 84.2 | **85.1** | **80.9** | 45.6 | 40.5 | **100.0** | 99.2 | 47.6 | 46.8 | **62.9** | **61.5** | **78.0** | **73.2** |
| ***Open-Source Large Language Models*** | | | | | | | | | | | | | | | | | | |
| LLaMA3-8B | 70.9 | 71.9 | 48.9 | 40.7 | 79.8 | 78.6 | 74.0 | 71.9 | 43.7 | 44.2 | 82.9 | 78.1 | 55.6 | 41.1 | 29.9 | 32.0 | 62.5 | 58.7 |
| LLaMA3.1-8B | 90.2 | 83.5 | 77.7 | 71.6 | 85.3 | 80.4 | 79.1 | 71.7 | 52.0 | 54.0 | 89.3 | 89.6 | 56.3 | 53.6 | 28.3 | 30.0 | 70.1 | 67.0 |
| Qwen2.5-7B | 88.5 | 79.8 | 49.1 | 43.6 | 83.5 | 82.2 | 77.2 | 75.3 | 79.4 | 69.3 | 92.1 | 93.7 | 56.0 | 53.7 | 34.9 | 30.6 | 69.3 | 66.1 |
| GLM4 - 9B - chat | 78.4 | 59.3 | 33.0 | 28.8 | 76.5 | 67.2 | 65.2 | 57.8 | 28.2 | 21.9 | 86.1 | 90.3 | 49.6 | 43.4 | 42.0 | 37.6 | 60.4 | 52.7 |
| Ministral - 8B | 45.6 | 45.9 | 20.5 | 20.2 | 76.1 | 72.1 | 68.7 | 62.5 | 69.0 | 59.9 | 40.5 | 51.3 | 15.5 | 14.5 | 23.6 | 13.1 | 43.7 | 43.6 |
| LLaMA3 - 70B | 69.1 | 57.5 | 42.8 | 33.2 | 83.3 | 72.8 | 75.8 | 64.2 | 56.4 | 39.2 | 83.2 | 86.2 | 50.0 | 45.3 | 25.4 | 28.4 | 61.7 | 53.0 |
| LLaMA3.1 - 70B | 90.0 | 77.2 | 75.8 | 62.2 | 85.8 | 82.7 | 73.4 | 69.2 | **70.2** | 63.0 | 96.4 | 97.1 | 65.9 | 59.0 | 36.8 | 27.9 | 73.8 | 65.2 |
| Qwen2.5 - 72B | **91.7** | 83.4 | 57.9 | 48.3 | 85.3 | 80.3 | 79.6 | 73.1 | 69.8 | **67.3** | 96.8 | **99.3** | **68.3** | **62.6** | 57.4 | 47.7 | 76.6 | 72.7 |
| ***Tool-Use-Finetuned Large Language Models*** | | | | | | | | | | | | | | | | | | |
| ToolLLaMA2-7B | 0.4 | 0.6 | 0.0 | 0.0 | 0.9 | 1.5 | 0.2 | 0.2 | 0.0 | 1.5 | 0.4 | 1.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.3 | 0.6 |
| ToolACE-8B | 14.6 | 9.1 | 1.8 | 1.0 | 20.4 | 16.5 | 18.2 | 14.3 | 4.0 | 2.2 | 10.7 | 2.4 | 7.1 | 6.2 | 10.5 | 14.8 | 11.9 | 10.3 |
| AgentLM-7B | 25.2 | 16.5 | 0.0 | 0.0 | 48.6 | 31.8 | 35.4 | 22.9 | 47.5 | 40.9 | 48.3 | 59.8 | 19.4 | 17.6 | 16.4 | 21.6 | 30.1 | 26.7 |

Standardization of Test Data

**System Prompt:**

You are a assistant who can utilize external tools. You can call the following tools:

```
[API List]
```

To use a tool, please directly generate the response in JSON format. Do NOT add comments (//) in the response.

```
{
    "name": string,   // tool name to call
    "args": Record<string, any>   // input params required by current tool in JSON format
}
```

If you already know the answer, or you do not need to use tools, please using the following format to reply:

```
{
    "name": "FinishAction",
    "args": "string"    // your final response to the query
}
```

Remember: only generate ONE step each time.

---

**User:** Help me maintain a social media presence by crafting a tweet that states, 'Managed to archive important data files!' using the hashtags #DataManagement and #Efficiency.

**Assistant:**

```
{
    "name": "TwitterAPI.authenticate_twitter",
    "args": {"username": "dr_smith", "password": "securePass123"}
}
```

**Function:**

```
{
    "authentication_status": true
}
```

**Prediction:** The assistant's tool-calling action for the next step.

**Ground True:**

```
{
    "name": "TwitterAPI.post_tweet",
    "args": {"content": "Managed to archive important data files!", "tags": ["#DataManagement",
    "#Efficiency"]}
}
```

Figure 10: An example of standardized test data.

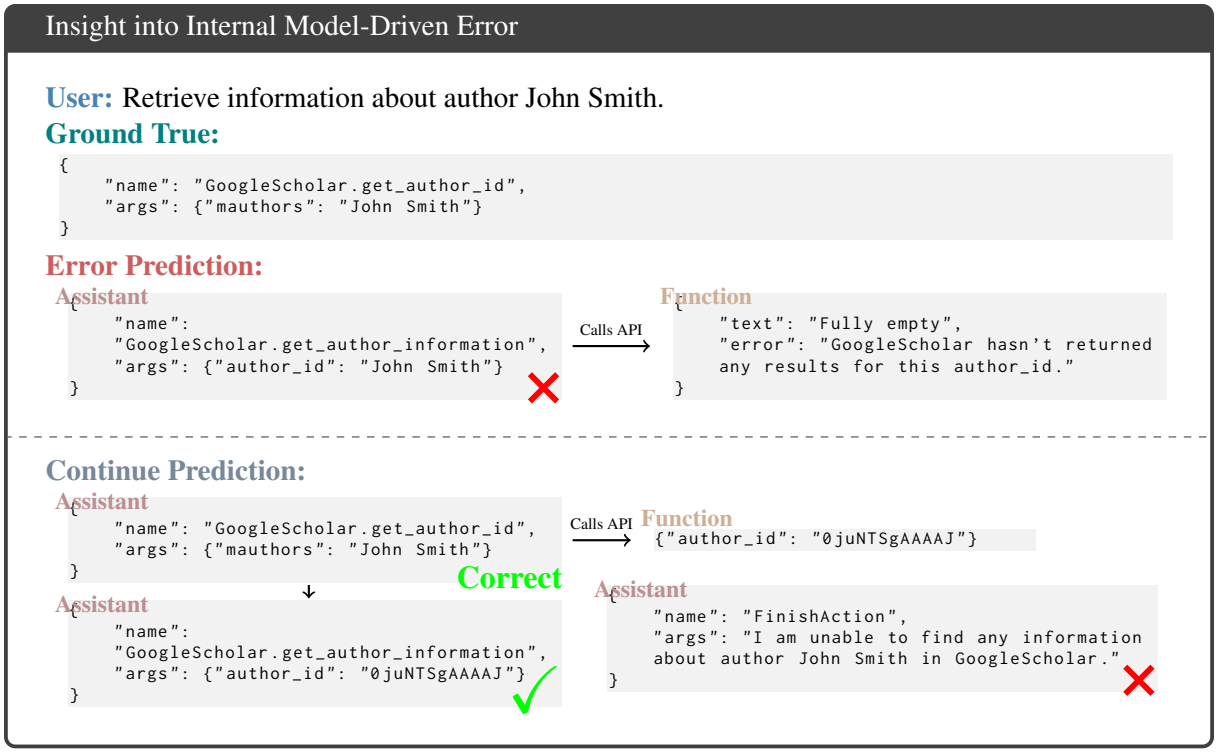


Figure 11: Examples of LLMs' performance in the presence of internal errors.

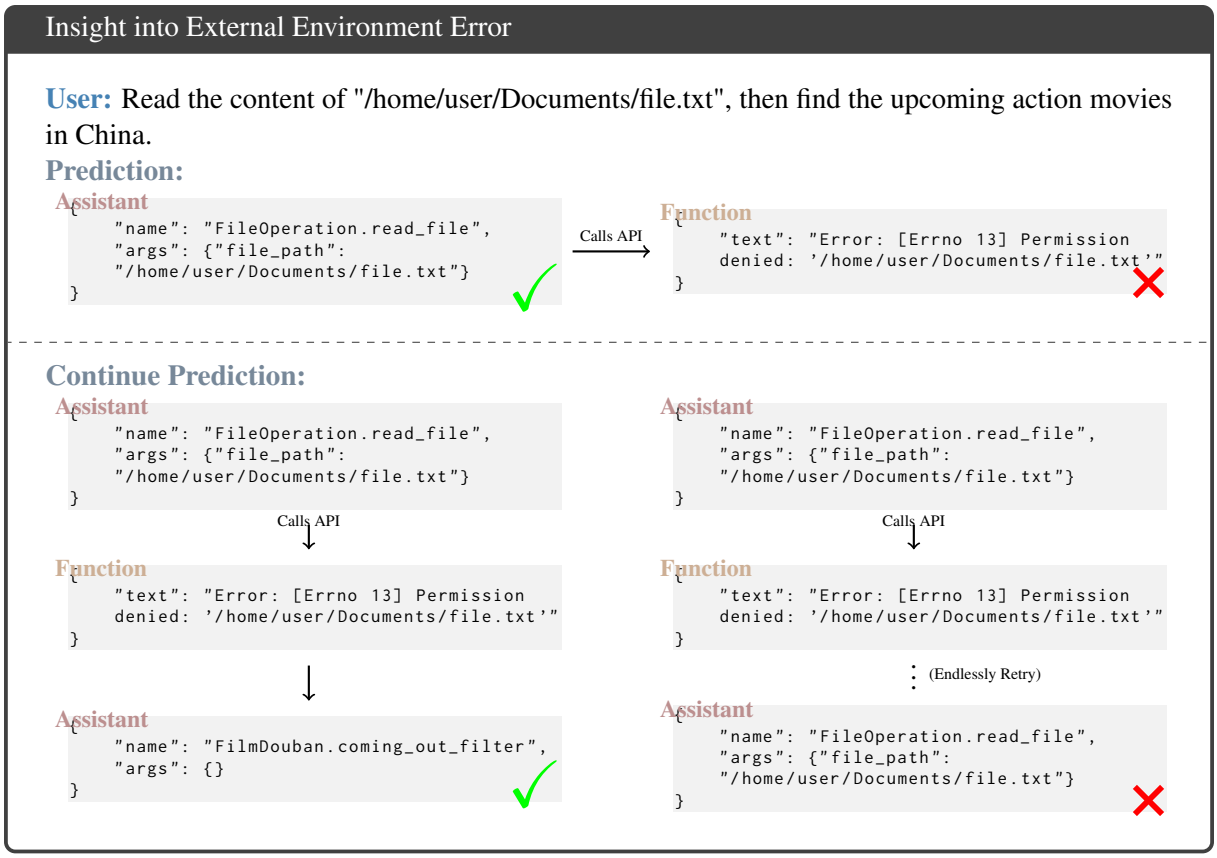


Figure 12: Examples of LLMs' performance in the presence of external errors.

Collected Tool-Calling Trajectory

**System Prompt:**
Same as standardized test data.

---

**User:** Tool-use task.

**Assistant:**

```
{
    "name": Golden answer API1,
    "args": Golden answer parameters1
}
```

**Function:**

```
{Response from API1 to the input parameters1}
```

**Assistant:**

```
{
    "name": Golden answer API2,
    "args": Golden answer parameters2
}
```

**Function:**

```
{Response from API2 to the input parameters2}
```

⋮

**Assistant:**

```
{
    "name": "FinishAction",
    "args": "The answer of the task is ..."
}
```

Figure 13: An example of collected tool-calling trajectories.

Refined API Documentation

```
{
    "name": "TravelAPI.cancel_booking",
    "description": "Cancel a booking",
    "required_parameters": [
        {
            "name": "access_token",
            "type": "string",
            "description": "[Required] The access token obtained from the authenticate"
        },
        {
            "name": "booking_id",
            "type": "string",
            "description": "[Required] The ID of the booking"
        }
    ],
    "optional_parameters": [],
    "return_data": [
        {
            "name": "cancel_status",
            "description": "The status of the cancellation, True if successful, False if failed"
        },
        {
            "name": "error",
            "description": "The error message if the cancellation failed"
        }
    ]
}
```

Figure 14: An example refined API documentation: TravelAPI.

Error Simulator

**System Prompt:**

**Character Introduction**

You are a large language modeling engineer, and your current task is to modify some conversation datas of large language model interacting with some external tool APIs. Your goal is to modify the content of the last reply of assistant in the correct dialog so that an error occurs and matches the error category I have given.

**Description of the Dialogues Structure**

- User presents the task and describes the problems to be solved.
- Assistant replies to solve the problems, may call the tool API or give the answer directly.
- Function is a tool API return that provides actual data or the results of performing a specific action.
- The interaction consists of several steps, and the assistant solves the problems step-by-step by calling functions.

**Your Task**

- Find the dialog to be modified: identify the last assistant response in each dialog that is the target of the message you need to modify.
- Understanding error categories: I will provide you with a specific error category, and you need to analyze the original dialog according to the error category and find out what needs to be modified, making sure that each step of your analysis is clear and reasonable.
- Conduct modifications: make the appropriate modifications based on the error category so that the dialog contains errors that match that error category.

**Response Format**

Follow the JSON format to output only the modified dialog without redundancy, and do not add comments (//) in the response.

```
{
"role": "assistant",
"content":"{('thought': string,  // goal at current step)
            'name': string,     // tool name to call
            'args': Record<string, any>}    // input params required by current tool in JSON
            format"
}
```

**Notes**

- Accuracy of JSON format: Please strictly follow the reply format, and output only the modified wrong tool call action of assistant.
- Reasonability of tool call: even if the error is generated, the called tool and its argument settings should be within a reasonable range, and the error should have some relevance to the correct dialog.
- Keep the chain of thought clear: although it is a simulation of the dialog and errors, assistant's thought process still needs to be clear and reasonable. Even if an error occurs, the logic of the assistant's reasoning when calling the tool should be complete.

**Modification Example**

```
[Randomly select 3 instances of a specific pattern of error from benchmark tests as few-shot.]
```

---

**User:**

Now I'll provide you with the error type and the correct dialog trajectory, please modify the last assistant's response to correspond to the error type.

```
Error Type: Tool Select Error/Tool Hallucination Error/Parameters Key Error/Parameters Value Error
Correct Dialog Trajectory: [randomly select the first k steps of tool call trajectory]
```

Figure 15: An example prompt of Error Diversification.

API Simulator

**System Prompt:**

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 response that aligns with the expected output of the API, guided by the provided examples.

Please note that your answer should not contain anything other than a json format object, which should be parsable directly to json, which is as follows:

```
{
    "error": "",
    "response": "<Your_Response>"
}
```

The error field should returns an explicit error message describing the cause of the error if there are any errors in the API Input. The response field must adhere strictly JSON format. <Your_Response> should contain the return_data 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.

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. If the user explicitly requests messages about failed API calls, and most of the examples provided get an error response despite passing in correct and valid parameters, please generate a failed tool call response containing some external environment errors. The external environment errors include rate limit exceeded, permission denied, maximum quota exceeded, timeout, connection error and so on. Please randomly select one kind of error above, the error message should match the corresponding API as much as possible, and don't show the words "external environment error".

Note that:

- You should strictly validate the parameters of the API Input to ensure all required_parameters are provided, the value of each parameter strictly conforms to the type specified in the API documentation, and there are no redundant parameter keys passed in. Be careful to identify the types of incoming parameters, even if they are the same as those specified by required_parameters when converted to strings, a different type can cause an error.
- If there is no error in the API Input and no explicit require by user, you should fill in the response field according to the rules, and the error field should remain empty. Otherwise, you should fill in the error field according to the rules, and the response field should remain empty.
- The response and error fields are not allowed to be filled in at the same time, you are only allowed to fill in one depending on the situation.
- Your response should be around 100 to 200 words, containing rich information given the API input parameters. Keep Your answer short and simple.

---

**User:**

API Documentation:

```
{API_doc}
```

API Examples:

```
{API_cache}
```

API Input:

```
{input args}
```

Figure 16: Prompt of API simulator.

CRITICTOOL Data Example1: Internal Error Critique Task

You are a assistant who can utilize external tools. You can call the following tools:

```
[API List]
```

To use a tool, please directly generate the response in JSON format. Do NOT add comments (//) in the response.

```
{
    "name": string,   // tool name to call
    "args": Record<string, any>   // input params required by current tool in JSON format
}
```

If you already know the answer, or you do not need to use tools, please using the following format to reply:

```
{
    "name": "FinishAction",
    "args": "string"    // your final response to the query
}
```

If you realize that an error occurred in the last step, then please analyze the error in the last step and regenerate the step to ensure that you get the correct response. Indicate the error type and select one of the following categories: tool select error (an exists but incorrect tool was chosen), tool hallucination error (a nonexistent tool was selected), parameters value error (the input parameter format or content is incorrect), parameters key error (incoming parameter keys are unrecognizable or required parameters are missing). And please using the following format to reply:

```
{
    "error": string,      //description about the type of error in the last step (tool select
                           error, tool hallucination error, parameters value error, parameters key
                           error)
    "name": string,       //tool name to call
    "args": Record<string, any>       // input params required by current tool in JSON format
}
```

Remember: only generate ONE step each time.

**User:**

Send an email to the team members (team_members@example.com) with the title 'Project Update', ... After sending the email, check for unread messages in the inbox.

---

**Error-Injected Data** (parameters key error, 'content' should be 'body'):

**Assistant:**

```
{
  "name": "Email.send",
  "args": "{'address': 'team_members@example.com', 'title': 'Project Update', 'content': 'Dear
  team members...'}"
}
```

**Function:**

Invalid parameter key: 'content'. Please provide all required parameters as defined in the API documentation.

---

**Error-Free Data** (interaction data without any errors):

**Assistant:**

```
{
  "name": "Email.send",
  "args": "{'address': 'team_members@example.com', 'title': 'Project Update', 'body': 'Dear team
  members...'}"
}
```

**Function:**

```
{
  "state": "Send mail successfully."
}
```

Figure 17: An example of internal error critique task in CRITICTOOL.

CRITICTOOL Data Example2: External Error Critique Task

You are a assistant who can utilize external tools. You can call the following tools:

```
[API List]
```

To use a tool, please directly generate the response in JSON format. Do NOT add comments (//) in the response.

```
{
    "name": string,   // tool name to call
    "args": Record<string, any>   // input params required by current tool in JSON format
}
```

If you already know the answer, or you do not need to use tools, please using the following format to reply:

```
{
    "name": "FinishAction",
    "args": "string"    // your final response to the query
}
```

(Continue) If you meet environment error from tool call, please retry to call it NO MORE THAN THREE TIMES, and if it still fails, you should skip the current step and perform the next task. Please using the following format to reply:

```
{
    "name": string,   // tool name to call
    "args": Record<string, any>   // input params required by current tool in JSON format
}
or
{
    "name": "FinishAction",
    "args": "string"    // your final response to the query
}
```

(Finish) If you meet environment error from tool call, please retry to call it NO MORE THAN THREE TIMES, and if it still fails, you should report the error to the user and ask the user for next action. Please using the following format to reply:

```
{
    "name": "FinishAction",
    "args": "string"    // ask user for guidance
}
```

---

**User:**

Could you fetch the meta information from the first three articles about 'data analysis'? After that, I would like to know what movies are currently showing in theaters.

**Assistant:** (correct tool call)

```
{
  "name": "ArxivSearch.get_arxiv_article_information",
  "args": "{'query': 'data analysis'}"
}
```

**Function:**

ERROR: Permission denied, unable to reach Arxiv.org at the moment. Please try again later.

Figure 18: An example of external error critique task in CRITICTOOL.

CRITICTOOL Data Example 3: Noisy Query Evolution Task

You are an assistant who can utilize external tools. You can call the following tools:

```
[API List]
```

To use a tool, please directly generate the response in JSON format. Do NOT add comments (//) in the response.

```
{
    "name": string,   // tool name to call
    "args": Record<string, any>   // input params required by current tool in JSON format
}
```

If you already know the answer, or you do not need to use tools, please using the following format to reply:

```
{
    "name": "FinishAction",
    "args": "string"    // your final response to the query
}//Some prompts are simplified
```

Remember: only generate ONE step each time.

**User:**

Compose an email to all team members at `team_members@example.com` detailing the features of the forthcoming film, 'Avengers: Endgame'. Subsequently, ascertain the availability of the first available meeting room from 2:00 PM to 4:00 PM and book it for our weekly marketing assembly.

**Assistant:**

```
[Wrong Response]
```

**Function:**

```
[Wrong Function Feedback]
```

---

**Complex Information**:

**User:**

My favourite film is *Avengers: Endgame*, I want to share it to my team members. Compose an email to all team members at `team_members@example.com` detailing the features of the forthcoming film *Avengers: Endgame*, including its plot, main characters, and key action sequences. You can also mention how the movie fits into the Marvel Cinematic Universe and its expected impact on upcoming releases. Following that, ascertain the availability of the first available meeting room from 2:00 PM to 4:00 PM and book it for our weekly marketing assembly. Additionally, weekly marketing assembly is very important. So please confirm the booking once it's done.

---

**Spelling Errors**:

**User:**

Compose an email to all tem nembers (typo, should be team members) at `team_members@example.com` detailing the features of the forthcomeing (typo, forthcoming) film, *Avengers: Endgame*. Subsequently, ascertain the availability of the first available meeting room form (typo, from) 2:00 PM to 4:00 PM and book it for our weekly marketig (typo, marketing) assembly.

---

**Expression Habits**:

**User:**

Please draft an email to all team members at `team_members@example.com`, highlighting the key features of the upcoming film *Avengers: Endgame*. Afterward, could you check if the first available meeting room is free from 2:00 PM to 4:00 PM and reserve it for our weekly marketing meeting?

Figure 19: An example of Noisy Query Evolution task in CRITICTOOL.

Noisy Query Evolution

**System Prompt:**

**Your Task**

- You are a helpful assistant and will receive a request from a user. This request is sent to a task related to the LLM model.
- Your task is to make this request as human-like as possible, such as adding irrelevant information, adjusting the expression habits that are irrelevant to the final task, adding spelling errors that do not affect the task, etc.

**Example**

Here is an example:

```
{
"Original Query": string,   // the original query
    "Query": string,   // the example refined query
}
```

**Response Format**

Please follow the JSON format and output according to the following structure

```
{
    "Query": string,   // the refined query
    "Explanation": string,   // the reason why you refine the query
}
```

Remember: be careful NOT to affect the completion of the task.

---

**User:** Here is the user query to be refined: Copy the txt contents of the 'Quarter1_Reports' directory and place it in a new directory naming it 'Archived_Quarter1.

Figure 20: An example prompt of Noisy Query Evolution.

Harder Tools Evolution

**System Prompt:**

**Your Task**

- You are a helpful expert. You will receive an API document. You need to change the description of this API but do not change other parts, especially parameters, etc.
- You can change the expression to make it more verbose. Do not change the original meaning of the description.

**Example**

Here is an example:

```
{
{
    "Original Document": dict,   // the original document
    "API Document": dict,   // the refined API document
}
}
```

**Response Format**

Please follow the JSON format and output according to the following structure

```
{
    "API Document": dict,   // the refined API document
    "Explanation": string,   // the reason why you refine the API document
}
```

Remember: be careful NOT to affect the completion of the API.

---

**User:** Here is the API document to be refined:

```
    {
        "name": "TimeTool.get_curr_time",
        "description": "Retrieve the current date and time",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            {
                "name": "time",
                "description": "The current date and time in the format YYYY-MM-DD HH:MM"
            }
        ]
    },
```

Figure 21: An example prompt of Harder Tools Evolution.

The verification of Long Context

**System Prompt:**
**Your Task**
- You are a helpful expert. You will receive a context from LLM and a user query task. Please judge whether the context will affect the task.
- Please be strict on this question. If it will affect, please reply Yes. If it will not affect, please reply No.
**Response Format**
Please follow the JSON format and output according to the following structure

```
{
    "Result": string,   // Yes or No
    "Reason": string,   // the reason why you think the context will or will not affect the task
}
```

---

**User:** Here is the context:

```
{
"role": "user",
"content":"..."
},
{
"role": "assistant",
"content":"..." the context extracted from LongBench
}
```

and the user task is:

```
I am planning a trip from Times Square to Central Park in New York City. I'd like to know the best
path to take, such as walking, biking, or taking public transportation.
```

Figure 22: An example prompt of the verification of Long Context.

The verification of Noisy Query

**System Prompt:**
**Your Task**
- You are a helpful expert. You will receive two user queries: A and B. You need to determine whether B completely contains the tasks in A and whether there is no ambiguity and typo in the important expression parts.
- If there is no ambiguity, output Yes, and if there is ambiguity, output No.
**Response Format**
Please follow the JSON format and output according to the following structure

```
{
    "Result": string,   // Yes or No
    "Reason": string,   // the reason why there is or is not ambiguity
}
```

---

**User:** Here is the user query A:

```
I am planning a trip from Times Square to Central Park in New York City. I'd like to know the best
path to take, such as walking, biking, or taking public transportation.  // the origin user query
```

Here is the user query B:

```
I am in the process of meticulously planning an excursion from the bustling Times Square to the
serene Central Park in the heart of New York City. I am quite curious to discover the most optimal
route to embark upon for this journey, whether it be the leisurely stroll of walking, the
environmentally friendly and energetic biking, or the efficient and convenient public
transportation system. Each option presents its own unique set of advantages and challenges, and I
am eager to weigh them all carefully.  // the new evolved user query
```

Figure 23: An example prompt of the verification of Noisy Query.

The verification of Extral Tools

**System Prompt:**

**Your Task**

- You are a helpful expert. You will receive two tool lists: tool list A and B. Your task is to determine whether there are particularly similar functions in these two function lists.
- If they are particularly similar, reply yes, otherwise reply no. Please be strict on this question.

**Response Format**

Please follow the JSON format and output according to the following structure

```
{
    "Result": string,   // Yes or No
    "Reason": string,   // the reason why the two tool lists are similar or different
}
```

---

**User:** Here is the tool list A:

```
    {
        "name": "Tool 1",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },
        {
        "name": "Tool 2",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },// the original tool list
```

Here is the tool list B:

```
    {
        "name": "Tool 3",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },
        {
        "name": "Tool 4",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },// the new added tool list
```

Figure 24: An example prompt of the verification of Extra Tools.

The verification of Harder Tools

**System Prompt:**
**Your Task**
- You will receive two API documents: API A and B. Your task is to determine whether the two APIs are equivalent, that is, whether the corresponding functions have the same parameters and whether the descriptions have the same meaning.
- The expressions may be slightly different, ignore typos).
- If they are equivalent, answer Yes, otherwise answer No.

**Response Format**
Please follow the JSON format and output according to the following structure

```
{
    "Result": string,   // Yes or No
    "Reason": string,   // the reason why the two APIs are equivalent or different
}
```

---

**User:** Here is the API A:

```
    {
        "name": "Tool 1",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },  //the origin API document
```

Here is the API B:

```
    {
        "name": "Tool 2",
        "description": "...",
        "required_parameters": [],
        "optional_parameters": [],
        "return_data": [
            "..."
        ]
    },  //the new evoluted API document
```

Figure 25: An example prompt of the verification of Harder Tools.

Self-Critique Performance in Internal Error-Injected Task

**Task:**
Error-injected data from internal critique task in Fig. 17.

**Prediction (Ground True):**

```
{
    "error": "parameters_key_error",
    "name": "Email.send",
    "args": "{'address': 'team_members@example.com', 'title': 'Project Update', 'body': 'Dear team
    members...'}"
}
```

**Prediction (Error Ignorance):**

```
{
    "name": "Email.show",
    "args": "{'only_unread': True}"
}
```

**Prediction (Correction Without Reflection):**

```
{
    "name": "Email.send",
    "args": "{'address': 'team_members@example.com', 'title': 'Project Update', 'body': 'Dear team
    members...'}"
}
```

**Prediction (Failure to Detect):**

```
{
    "error": "parameters_value_error",
    ...    //(Error correction is uncertain)
}
```

**Prediction (Failure to Correct):**

```
{
    "error": "parameters_key_error",
    ...    //(Does not match groundtruth answer)
}
```

**Prediction (Tool Call Interruption):**

```
{
    "name": "FinishAction",
    "args": "Cannot send email successfully."
}
```

Figure 26: Qualitative examples of self-critique performance in internal error-injected task.

Self-Critique Performance in Internal Error-Free Task

**Task:**
Error-free data from internal critique task in Fig. 17.

**Prediction (Ground True):**

```
{
    "name": "Email.show",
    "args": "{'only_unread': True}"
}
```

**Prediction (Over-Reflection):**

```
{
    "error": ...,
      ...
}
```

Figure 27: Qualitative examples of self-critique performance in internal error-free task.

Self-Critique Performance in External Error Task

**Task:**
External critique task in Fig. 18.

**Prediction (Continue Ground True):**

```
{
    "name": "ArxivSearch.get_arxiv_article_information",
    "args": "{'query': 'data analysis'}"
}
...(Repeat no more than three times)
{
    "name": "FilmTMDB.get_now_playing",
    "args": "{}"
}
```

**Prediction (Finish Ground True):**

```
{
    "name": "ArxivSearch.get_arxiv_article_information",
    "args": "{'query': 'data analysis'}"
}
...(Repeat no more than three times)
{
    "name": "FinishAction",
    "args": "I am unable to access 'ArxivSearch.get_arxiv_article_information' to get articles on
    data analysis, could you provide further guidance on what to do?"
}
```

**Prediction (Endless Retry):**

```
{
    "name": "ArxivSearch.get_arxiv_article_information",
    "args": "{'query': 'data analysis'}"
}
...(Repeat more than three times)
```

**Prediction (Hallucination):**

```
{
  "name": "FinishAction",
  "args": "I have fetched the meta information from the first three articles about 'data
  analysis'."
}
```

Figure 28: Qualitative examples of self-critique performance in external error task.