Do LLMs Catch Their Own Mistakes? A Comprehensive Benchmark for Reflective Tool Use LLMs

Anonymous ACL submission

Abstract

Large language models (LLMs) increasingly rely on external tools to complete complex tasks, yet their ability to recognize and correct their own tool-use mistakes remains underexplored. Existing benchmarks primarily evaluate planning and execution success, overlooking the self-reflective dimension of tool use. To address this gap, we present ReflecTool-Bench, the first benchmark designed to assess LLMs' self-reflective reasoning in tool-augmented multi-turn dialogues. ReflecTool-Bench covers 10 domains with 88 distinct APIs and 968 annotated dialogues, systematically injecting diverse error types arising from both user and assistant behavior. The benchmark defines two complementary evaluation setups: the Critique task, where models diagnose errors in third-party dialogues, and the Self-Reflection Task, where models must detect and repair their own prior tool-use mistakes. We introduce fine-grained metrics for error detection, error classification, correction accuracy, and explanation quality, enabling a holistic assessment of reflective reasoning. Evaluations across 12 state-of-the-art models, including both API-based closed source models and open source models, reveal that while models can reliably identify user-originated errors, they struggle with assistant-originated ones, and performance drops sharply when moving from critique to self-reflection.

1 Introduction

The rapid development of Large Language Models (LLMs) has enabled them to demonstrate remarkable reasoning and generation capabilities across a wide range of tasks (Tan et al., 2024; Liu et al., 2024; Fu et al., 2024; Qin et al., 2023a; Ouyang et al., 2022). This exceptional ability has further motivated the design of intelligent agents capable of interacting with external environments. Later, the integration of external tool use has further empowered LLMs, allowing them to retrieve knowledge, perform actions, and complete complex multi-step goals (Parisi et al., 2022; Schick et al., 2023; Mialon et al., 2023). Despite these attractive capabilities, a critical question remains underexplored: *are LLMs aware of their own mistakes during tool usage and able to propose viable corrections?* Existing works mainly focus on evaluating tool utilization ability based on final outputs or single-step tool calls (Qin et al., 2023b; Li et al., 2023; Chen et al., 2023b; Guo et al., 2024). Nevertheless, such evaluations overlook the reflective dimension of reasoning—whether a model can recognize errors, explain them, and fix them through correction. Moreover, many current benchmarks emphasize tool execution accuracy or planning success, while neglecting self-reflective reasoning and error-awareness in multi-turn dialogues.

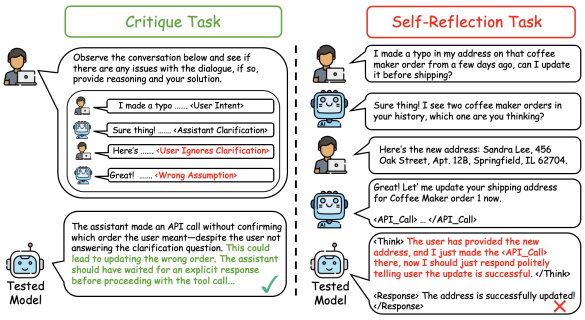


Figure 1: Two tasks in ReflecTool-Bench. The left side illustrates the Critique task, where the model evaluates completed dialogues as an external observer. The right side depicts the Self-Reflection Task, where the model continues its own dialogue without explicit prompts.

To address these issues, we propose ReflecTool-Bench, a comprehensive benchmark designed to evaluate self-reflective reasoning in tool-augmented multi-turn dialogues. Unlike prior works that measure only execution correctness, ReflecTool-Bench systematically injects diverse error types and examines whether models can (1) detect the error, (2) explain their causes, and (3) repair tool calls. In particular, our benchmark in-

troduces a decomposition of evaluation into two complementary tasks: the *Critique task*, where the model acts as an external observer of dialogues, and the *Self-Reflection Task*, where the model acts as the assistant in the conversation and needs to identify and correct its own prior mistakes based on the context. Through extensive experiments on ReflecTool-Bench, we find that while state-of-the-art models can reliably identify errors, they struggle with those originating from the assistant itself. In these cases, when the model selects the wrong tool, misinterprets user intent, or produces malformed API calls, its correction accuracy remains far behind its detection ability. Later, the error analysis shows that most models fail to bridge the gap between recognition and actionable repair, underscoring the challenge of reflective reasoning. Our major contributions are summarized as follows:

1. We introduce ReflecTool-Bench, the first benchmark that systematically evaluates the self-reflective reasoning of LLMs in tool-augmented multi-turn dialogues.

2. ReflecTool-Bench employs a multi-agent framework to construct realistic dialogues, inject diverse user- and assistant-originated errors, and provide structured annotations for error detection, explanation, and correction.

3. We conduct extensive experiments with both API-based and open-source LLMs, offering a comprehensive analysis of their reflective capabilities, error patterns, and limitations.

2 Related Works

Self-Reflective LLMs. Recent works show that LLMs can improve through self-reflection, where models iteratively refine their outputs based on internal feedback (Renze and Guven, 2024; Madaan et al., 2023; Pan et al., 2023; Chen et al., 2023a; Asai et al., 2024; Wang et al., 2022; Huang et al., 2023). Some methods rely on single-turn self-critique (Madaan et al., 2023; Chen et al., 2023a), while others employ multiple model instances, such as in multi-agent debate (Du et al., 2023; Sun et al., 2025). However, these approaches often fail to build a coherent understanding when relying solely on past outputs or self-evaluations (Zhang et al., 2024). Reflexion (Shinn et al., 2023) incorporates prior trajectories and rewards to guide future behavior, while Retroformer (Yao et al., 2023) further improves reflection by fine-tuning a dedicated reflector using RLHF. Despite these advanced investigations, none of the prior works has examined LLMs' reflective and corrective abilities in tool-use scenarios, which introduce distinct challenges in reasoning and feedback integration.

Augmenting LLMs with Tool-Usage. A number of works explore empowering LLMs with external tools, either by integrating tool usage directly into model weights via special tokens (Schick et al., 2023; Parisi et al., 2022; Lewis et al., 2020; Hao et al., 2023), or by leveraging in-context learning to guide tool use through descriptive prompts (Hsieh et al., 2023; Mialon et al., 2023; Ruan et al., 2023; Patil et al., 2024; Tang et al., 2023). Building on these developments, several benchmarks have been proposed to evaluate tool-use capabilities. For example, ToolQA (Zhuang et al., 2023) assesses response comparison across different tool interactions; ToolBench (Qin et al., 2023b) uses LLM judges to evaluate full solution paths; API-Bank (Li et al., 2023) evaluates tool-use capabilities across API selection, parameter prediction, and execution accuracy; and T-Eval (Chen et al., 2023b) offers fine-grained evaluation across planning, reasoning, and retrieval dimensions. In contrast, ReflecTool-Bench is the first benchmark designed to assess LLMs' self-reflective and self-corrective capabilities in tool-use scenarios, highlighting a new dimension of autonomous reasoning and error awareness.

3 The ReflecTool-Bench Benchmark

In this section, we first elaborate on the design motivations and data construction pipeline behind ReflecTool-Bench, including how we leverage multi-agent framework to collaboratively generate both correct and erroneous dialogues, grounded in diverse API schemas. Building on this dataset, we define a suite of evaluation metrics targeting multiple dimensions of self-reflective reasoning. These metrics are assessed under two experimental setups: the **Critique task**, where the model evaluates tool-use errors in a third-party dialogue, and the **Self-Reflection Task**, where the model reflects on and corrects its own tool calls. Together, these setups enable a holistic assessment of LLMs' ability to reason about tool-use failures from both external and internal perspectives. Finally, we provide a comprehensive statistical overview of ReflecTool-Bench, highlighting its scale, domain coverage, error distribution, and tool diversity.

Example API Tool Schema

```
{
  "name": "order_status",
 "description": "Gets info about a customer's past orders,
        including status, delivery date, and product info.",
  "parameters": {
    "type": "object",
    "orderName": {
      "type": "string",
    "description": "Product name or type from the customer
                    (e.g., 'AirPods', 'book order')."
    },
    "orderDate": {
      "type": "string",
      "description": "When the order was placed
                     (e.g., 'last week', 'June 12')."
    },
    "deliveryDate": {
      "type": "string",
     "description": "Expected or actual delivery window
                     (e.g., 'June 20-22')."
    }
  }
  "required": ["orderName"],
}
```

Figure 2: Example tool schema used to guide agentic behavior in dialogue generation and error injection.

3.1 Dataset Construction

The construction of ReflecTool-Bench involves three stages: *tool collection*, *dialogue creation*, and *error injection*. While the tool collection stage focuses on curating APIs from various domains, the latter two stages employ a multi-agent framework to simulate realistic tool-augmented interactions and systematically inject errors. An overview of the entire benchmark construction process is illustrated in Figure 3, and detailed agent prompting strategies are provided in Appendix D.

3.2 Tool Collection

To ground the benchmark in realistic tool-use scenarios, we collect API schemas from two major sources: the RapidAPI marketplace (RapidAPI) and the OpenAPI directory (TM Forum). These platforms offer a broad and diverse set of public APIs across domains such as e-commerce, finance, travel, and more. Our selection emphasizes real-world relevance and parameter richness, ensuring that the resulting tasks reflect practical decision-making challenges rather than synthetic or overly simplified examples. Each selected API is represented in a structured JSON schema that specifies its name, functionality, and input parameters. A processed API is shown in Figure 2. To ensure quality and task diversity, human annotators manually curated and verified each tool. This involved filtering out low-quality or trivial APIs, verifying parameter definitions, and selecting tools that allow for plausible multi-turn reasoning and error scenarios in downstream dialogue generation.

3.3 Dialogue Creation

Inspired by Chen et al. (2023b) that employed a multi-agent pipeline for dataset construction, we adopt a similar framework to generate our dataset, as detailed in the following subsections. A sample generated conversation is shown in Figure 7.

Description Agent. This agent takes the collected API tool from the data collection stage and generates a descriptive demonstration of this tool including its functionality, required parameters, their representations, and the output format. etc. These descriptions are later used to instruct downstream agents and improve the overall coherence and correctness of the dialogue.

User Agent. To simulate realistic user behavior across diverse domains, we design a user agent that acts as a layperson sending requests about relevant questions within the field. For example, if the domain is e-commerce, the user may have relevant request, such as returning or replacing items. Most importantly, the user agent has no access to the tool schema or parameter specifications. Instead, it is guided by high-level task prompts instructing it to make natural, goal-oriented requests, and supported by curated reference examples annotated by domain experts. This design ensures that user requests remain natural and often reflect real-world challenges—such as being under-specified, ambiguous, or noisy.

Assistant Agent. The assistant agent simulates a domain-specific LLM assistant capable of executing tool calls. It is provided with both the structured API schema and the detailed schema descriptions generated by the *Description Agent*. The assistant engages in a multi-turn dialogue with the *User Agent*, responding to user queries and invoking tools where appropriate.

To encourage interpretability and structured reasoning, the assistant follows a standardized prompting mechanism that separates internal reasoning and tool interaction. Each response consists of a `<think>` block (articulating its rationale), a `<tool call>` block (if a tool invocation is needed), and a `<response>` block. Human annotators reviewed dialogues to ensure that each assistant-generated dialogue concludes with a correct tool call. These generated dialogues serve as the ground truth upon which subsequent error injection is performed.

3.4 Error Injection

Next, to assess a model's capacity for self-reflection in the presence of tool-use failures, we introduce a structured error injection stage that systematically perturbs the dialogues generated by the interaction between the user and assistant agents. This process simulates realistic tool-related mistakes and allows for fine-grained evaluation of the model's ability to detect, explain, and correct different types of errors including **parameter mismatch**, **missing required parameter**, **incorrect tool selection**, **ambiguous reference**, and **over/under specification**. Detailed definitions of each error are provided in Appendix A.1. We categorize errors based on their origin, introducing two specialized agents: the User Error Agent and the Assistant Error Agent.

User Error Agent. The User Error Agent modifies the original user request to simulate plausible failures encountered in real-world human-computer interactions. These modifications are designed to preserve the user's intent while introducing challenges such as ambiguous phrasing, under-specified requests, irrelevant over-specifications, or incorrect assumptions about the system's capabilities. For example, the user may ignore or misunderstand a follow-up question from the assistant, respond with an off-topic answer, or repeat their original request without adding new information. In other cases, the user might provide vague or overly general descriptions—such as saying "book a ticket" without specifying a destination—or mix conflicting constraints, like asking for "the cheapest and fastest option." These behaviors reflect realistic user uncertainty and introduce ambiguity into the assistant's reasoning process, making it more challenging to identify the correct tool usage. Despite these changes, the underlying task goal and the correct tool call remain fixed, enabling evaluation under consistent ground truth.

Assistant Error Agent. The Assistant Error Agent, by contrast, introduces failures in the assistant's reasoning or execution of tool calls. These errors are grounded in common failure modes observed in LLM behavior, such as selecting an incorrect tool for the task, providing incorrect or incomplete parameter values, invoking a tool prematurely without sufficient user clarification, or failing to call a tool when one is clearly required. In each case, the assistant's response deviates from the correct behavior while maintaining the original user input, allowing us to isolate and evaluate the assistant's internal decision-making process.

Each error-injected dialogue is paired with a structured annotation that provides supervision signals. This includes the type of error introduced, the original ground truth tool call from the clean dialogue, and an explanation detailing how the error arose, why it led to a faulty outcome, and what the assistant should have done instead.

3.5 Evaluation Decomposition

To evaluate the self-reflective ability of LLMs in tool-augmented dialogues from multiple perspectives, we decompose the evaluation into two distinct setups: the *Critique task* (non-reflective evaluation) and the *Self-Reflection Task* (reflective evaluation). These setups differ in how the model interacts with the dialogue and whether it is judging its own behavior or that of another agent.

3.5.1 Critique task.

In the Critique task, the model is presented with a complete dialogue between a user and an assistant, including any tool calls and their corresponding outputs. The model acts purely as an external observer, with no involvement in generating the conversation. Its objective is to analyze the dialogue holistically and determine whether any tool-use error occurred. If an error is identified, the model must further explain its reasoning, classify the type of error, and propose a corrected tool call. This task enables retrospective evaluation of an LLM's ability to detect, interpret, and resolve tool-use failures caused by either the user or the assistant. We define the following evaluation metrics to quantify performance in this setting:

Error Detection Accuracy. This metric measures the model's ability to correctly determine whether a dialogue contains a tool-use error. Let N be the total number of evaluation samples. For each sample i, let $y_i \in \{\texttt{Correct}, \texttt{Incorrect}\}$ be the ground truth label, and $\hat{y}_i$ the model's predicted label. The error detection accuracy is defined as:

$$\text{Acc}_{\text{detect}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[\hat{y}_i = y_i]. \quad (1)$$

We report this metric separately for user-originated and assistant-originated errors, as well as their average.

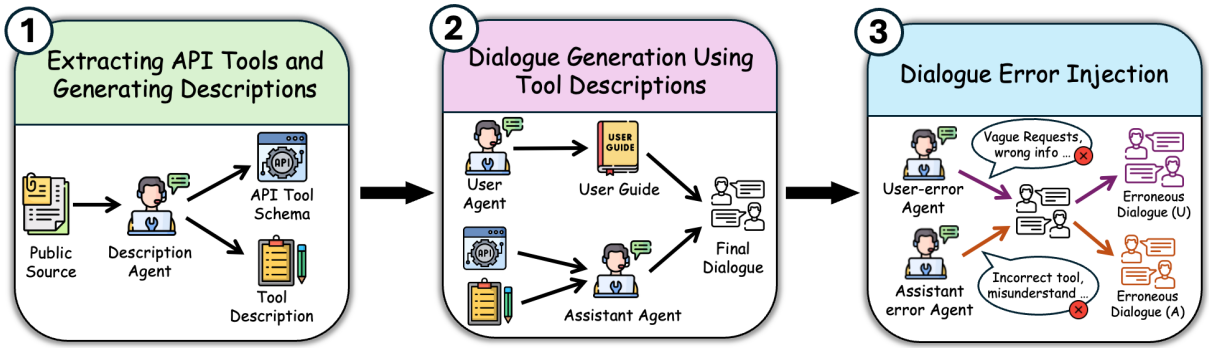


Figure 3: Demonstration of the creation of ReflecTool-Bench, a three-stage pipeline involving tool extraction, dialogue generation with user–assistant interactions, and error injection to simulate realistic tool-use failures.

Error Classification Accuracy. If the model predicts that an error occurred (i.e., $\hat{y}_i = \texttt{Incorrect}$), it must also classify the error into one of several predefined types. Let S_{err} be the subset of samples for which the model predicts an error, with $|S_{\text{err}}|$ denoting its size. Let c_i be the true error class for sample i, and $\hat{c}_i$ the predicted class. The classification accuracy is computed as:

$$\text{Acc}_{\text{class}} = \frac{1}{|S_{\text{err}}|} \sum_{i \in S_{\text{err}}} \mathbb{1}[\hat{c}_i = c_i]. \quad (2)$$

This metric reflects the model's ability to correctly identify the nature of the error once it has been detected.

Correction Accuracy. For the same subset S_{err}, the model must also propose a corrected tool call. Let t_i denote the ground truth corrected tool call for sample i, and $\hat{t}_i$ the model's proposed correction. The correction accuracy is defined as:

$$\text{Acc}_{\text{corr}} = \frac{1}{|S_{\text{err}}|} \sum_{i \in S_{\text{err}}} \mathbb{1}[\hat{t}_i = t_i]. \quad (3)$$

This metric evaluates whether the model can successfully recover the correct tool usage after identifying and analyzing an error.

Reasoning Quality. We evaluate the model's explanation quality on a 1–5 scale based on two dimensions: correctness and helpfulness. A score of 5 corresponds to a fully accurate, clear, and comprehensive explanation that closely aligns with the ground truth, while a score of 1 reflects a misleading, incorrect, or uninformative response. Scoring is performed automatically using GPT-based evaluation with a fixed rubric and manually crafted few-shot examples. We report the average score separately for user- and assistant-originated errors. Full scoring details and the evaluation prompt are provided in Appendix C.

3.5.2 Self-Reflection Task.

In contrast to the Critique task, which evaluates models as external observers, the Self-Reflection Task places the tested model directly into an ongoing dialogue with a user. Here, the dialogue is pre-generated by agents and potentially contains a tool-use error—either from the user or the assistant. The tested model is then inserted into the role of the assistant and prompted to continue the conversation from that point onward, without being explicitly told that an error has occurred. Importantly, the model receives no special instructions to perform reflection or correction. Instead, it is evaluated based on whether it spontaneously realizes the error in the prior turns and takes corrective action through its natural continuation. This setup emulates realistic deployment scenarios where models need to recognize and respond to mistakes without external prompting.

We evaluate the model's response using the same three dimensions as in the Critique task: error detection accuracy, correction accuracy, and reasoning quality. Since the model's behavior is embedded in a free-form response rather than structured output, we leverage GPT 4o as a judge to automatically evaluate whether the model has successfully (1) identified the previous error, (2) corrected it, and (3) provided a coherent and helpful rationale. The explanation is scored on a 1–5 scale using the same rubric as in the critique setting. Metrics are reported separately for user-induced and assistant-induced errors, along with their averages. Full scoring guidelines and evaluation prompts are provided in Appendix C.

3.6 Dataset Summary

To this end, ReflecTool-Bench covers 10 diverse domains and 42 subdomains, spanning 88 distinct APIs and 968 annotated tool-augmented dia-

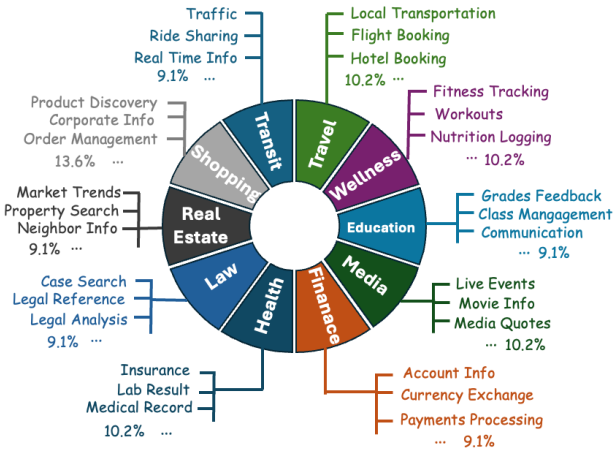


Figure 4: Specific domain/subdomain distribution of ReflecTool-Bench

logues. Each dialogue involves user requests, assistant reasoning, assistant responses, tool calls, and simulated results. For both the critique and self-reflection tasks, we construct 11 dialogue variants per scenario: one clean (error-free) version, and ten error-injected versions—covering 5 distinct tool-use errors, with half originating from the user and half from the assistant. Figure 4 shows the domain and subdomain distribution.

4 Experiments

In this section, we present a comprehensive comparison of different closed and open source models under two distinct tasks in ReflecTool-Bench.

4.1 Experimental Setup

We evaluate a total of 12 models on ReflecTool-Bench, encompassing both commercial API-based LLMs and open-source models to provide a comprehensive assessment of self-reflective tool-use capabilities. (1) For API-based models, we include six widely used systems: GPT-o1, GPT o4-mini, and GPT-5-mini from OpenAI (Achiam et al., 2023), along with Claude Haiku, Claude Sonnet, and Claude Opus from Anthropic (Anthropic, 2025). (2) For open-source models, we select a diverse range spanning various model sizes and architectures, including Mistral (Jiang et al., 2023), Qwen 2.5 (Team, 2024), Qwen 3 (Team, 2025), LLaMA 3.1 (Dubey et al., 2024), and DeepSeek R1-Distillation (DeepSeek-AI, 2025).

4.2 Implementation Details

All the experiments across different models were conducted on eight NVIDIA A100-SXM4 (80GB) GPUs with a CUDA version of 12.4.

4.3 Main Results

In this section, we evaluate how model performance shifts from the Critique task to the Self-Reflection task using the ReflecTool-Bench benchmark (Table 1 vs. Table 2). Our goal is to answer: **Among both API-based and open-source models, which exhibit greater robustness when shifting from critique to self-reflection task?** Across the board, models show reduced accuracy in detection, correction, and overall detect-and-correct metrics under self-reflection, reflecting the increased complexity of reflective reasoning. To ensure fair comparisons, since models do not perform error classification explicitly in self-reflection, we average detection and correction scores.

Among the API-based models, Claude Opus and GPT o4-mini continue to perform best, but both show noticeable declines when moving to the self-reflection setting. Their ability to detect errors becomes less reliable, though they compensate with some improvement in correction, suggesting they attempt more fixes but with mixed success. By contrast, weaker models like Claude Haiku become better at spotting errors but make only limited progress in actually correcting them.

The contrast is even more obvious for open-source models. LLaMA shows a steep decline in its ability to detect errors and makes little progress in correction, while Mistral struggles across both dimensions. In contrast, Qwen 2.5-7B demonstrates the strongest resilience, maintaining stable detection and showing substantial gains in correction, outperforming the other open-source models. Overall, these results highlight that **self-reflection imposes greater challenges on weaker or less adaptable models**, whereas more robust systems like Qwen 2.5 manage to adapt more effectively.

5 Discussion

Our curated benchmark serves as a valuable resource for evaluating the self-reflective and self-corrective capabilities of LLMs in tool-augmented multi-turn dialogues. In this section, we address key research questions that are crucial for advancing the development of more reliable and accurate tool-using LLM agents. Additional analysis can be found in Appendix B.

5.1 User-Origin. vs. Assistant-Origin. Errors

Here, the research question we aim to answer is: **How do LLMs handle errors originating from the user versus those caused by the assistant?**

Model	Error Detect. Acc (%) (↑)			Correction Acc (%) (↑)			Detect & Correct Avg (%) (↑)	Error Class. Acc (%) (↑)			Reasoning Rationale (1–5) (↑)		
	User	Asst.	Avg	User	Asst.	Avg		User	Asst.	Avg	User	Asst.	Avg
API-based Models													
GPT o1	89.88	82.13	86.00	15.47	20.32	17.89	51.95	**91.16**	**85.60**	**88.38**	4.22	3.99	4.10
GPT o4-mini	91.92	87.30	89.61	**17.82**	**21.69**	**19.75**	**54.68**	81.98	75.24	78.61	4.33	**4.14**	4.24
GPT 5-mini	90.91	88.55	89.73	13.08	18.74	15.91	52.82	88.01	83.83	85.92	4.27	4.15	4.21
Claude Opus	**97.53**	87.88	**92.71**	12.10	19.39	15.75	54.23	81.59	75.84	81.59	4.44	44.00	4.22
Claude Sonnet	94.73	**89.15**	91.94	13.81	17.58	15.69	53.82	88.76	**80.65**	**84.71**	**4.46**	**4.14**	**4.30**
Claude Haiku	67.39	42.14	54.76	15.50	19.19	17.34	36.05	76.96	72.19	74.57	3.29	2.35	2.82
Open-Source Models													
Mistral-7B-Instruct	41.93	32.09	37.01	7.29	15.74	11.52	24.26	28.15	25.21	26.68	2.04	1.59	1.81
Qwen2.5-7B-Instruct	82.35	59.66	71.01	15.30	20.60	17.95	44.48	74.70	73.31	74.01	3.69	2.86	3.27
Qwen3-4B-Instruct	76.96	55.40	66.18	16.96	24.81	20.89	43.53	20.18	37.09	28.63	3.78	3.02	3.40
LLaMA3.1-8B-Instruct	**94.69**	**86.46**	**90.58**	14.82	17.60	16.21	**53.39**	81.58	68.18	74.88	3.68	3.32	3.50
Qwen3-32B	80.32	72.27	76.29	**17.53**	20.66	**19.09**	47.69	65.13	58.91	62.02	**3.93**	**3.61**	**3.77**
DeepSeek-R1-Qwen3-8B	86.28	79.18	82.73	15.08	**20.86**	17.97	50.35	81.71	**79.50**	**80.61**	3.90	**3.61**	3.75

Table 1: Overall results (across all domains) of various API-based and open-source models on ReflecTool-Bench under the **Critique task** (non-self-reflection setting). **Bold** indicates the best performance, and underline marks the second best. To align with the self-reflection setting—where models do not explicitly classify identified errors—we report the average of error detection and correction scores as a comparable measure. Metrics include error detection accuracy, correction accuracy, detect-and-correct average, error classification accuracy, and reasoning rationale, each reported separately for user- and assistant-caused errors. ↑ indicates that higher values are better.

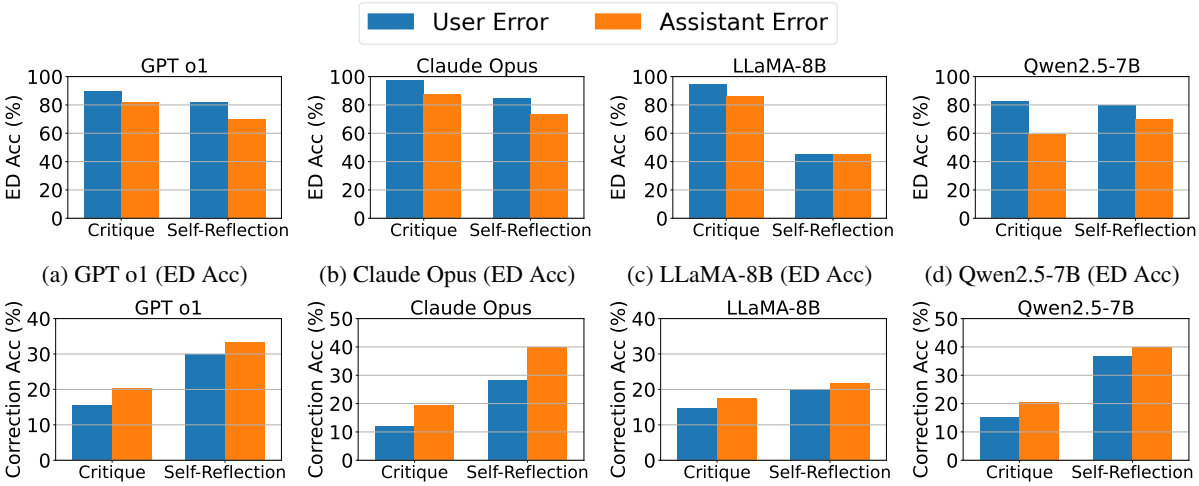
Model	Error Detect. Acc (%) (↑)			Correction Acc (%) (↑)			Detect & Correct Avg (%) (↑)	Reasoning Rationale (1–5) (↑)		
	User	Asst.	Avg	User	Asst.	Avg		User	Asst.	Avg
API-based Models										
GPT o1	81.67	70.00	75.84	**30.00**	33.33	31.67	53.75	3.93	3.53	3.73
GPT o4-mini	**85.00**	**75.00**	**80.00**	**30.00**	33.33	31.67	55.83	**4.27**	3.67	**3.97**
GPT 5-mini	75.95	52.25	64.10	22.27	24.33	23.30	43.70	3.95	3.29	3.62
Claude Opus	**85.00**	73.33	79.17	28.33	**40.00**	**34.17**	**56.66**	4.03	**3.72**	3.88
Claude Sonnet	83.17	70.66	76.92	18.33	26.67	22.50	49.71	3.57	3.50	3.53
Claude Haiku	81.67	63.33	72.50	**30.00**	26.67	28.33	50.41	3.85	3.17	3.51
Open-Source Models										
Mistral-7B-Instruct	20.00	3.33	11.67	1.67	0.00	0.83	6.25	1.87	1.77	1.82
Qwen2.5-7B-Instruct	**80.00**	**70.00**	**75.00**	**36.67**	**40.00**	**38.33**	**56.67**	2.97	**3.17**	**3.07**
Qwen3-4B-Instruct	49.32	28.12	38.72	18.66	15.59	17.13	27.92	2.51	2.28	2.40
LLaMA3.1-8B-Instruct	45.00	45.00	45.00	20.00	21.67	20.83	32.92	2.33	2.55	2.44
Qwen3-32B	48.89	31.03	39.96	30.81	36.80	33.81	36.88	2.09	1.86	1.97
DeepSeek-R1-Qwen3-8B	65.38	55.59	60.49	13.17	17.03	15.10	37.79	**3.08**	2.88	2.98

Table 2: Overall results (across all domains) of various API-based and open-source models on ReflecTool-Bench under the **Self-Reflection** task (self-reflection setting). **Bold** indicates the best performance, and underline marks the second best. Models are evaluated using error detection accuracy, correction accuracy, the average of detection and correction scores, error classification accuracy, and reasoning rationale, each reported separately for user- and assistant-caused errors. ↑ denotes that higher values indicate better performance.

User-originated errors often arise from vague or underspecified queries, while assistant-originated errors involve incorrect tool choice, or malformed calls despite clear intent. To investigate this, we analyze model performance across both the Critique and Self-Reflection tasks, comparing their ability to detect, explain, and correct errors from each source (see Figure 5).

5.1.1 Critique task

We begin by examining model behavior in the Critique task, where the model evaluates completed dialogues as an external observer. As shown in Figure 5a-5d, the average error detection accuracy is consistently higher for user-originated errors than for assistant-originated ones. This trend is evident across both API-based and open-source models. For instance, models such as GPT o4-mini and Claude Opus exceed 90% in user error detection, but performance drops markedly—by over 10 percentage points in some cases—when detecting assistant errors. **This suggests that LLMs are more sensitive to identifying input-level flaws (e.g.,**



(a) GPT o1 (ED Acc) (b) Claude Opus (ED Acc) (c) LLaMA-8B (ED Acc) (d) Qwen2.5-7B (ED Acc)

(e) GPT o1 (Corr. Acc) (f) Claude Opus (Corr. Acc) (g) LLaMA-8B (Corr. Acc) (h) Qwen2.5-7B (Corr. Acc)

Figure 5: Error Detection (ED) accuracy and Correction accuracy across different models. In subplots 5a, 5b, 5c, 5d, the y-axis denotes the ED accuracy. In the rest of subplots, the the y-axis represents the correction accuracy. The x-axis reflects performance across two different tasks.

vague or malformed user queries) than failures in their own reasoning or tool usage. In contrast, correction accuracy is substantially lower overall and shows even greater disparity. While some models can effectively correct user errors, most struggle with assistant-originated ones, often falling below 20% accuracy. This indicates that even when a model can detect a flaw, it frequently lacks the ability to suggest or execute an appropriate correction, particularly when the error stems from internal reasoning faults rather than external user inputs. The average of detection and correct score reflects a consistent gap: even top models often detect errors but fall short in correcting them, particularly for assistant-side mistakes. This further highlights the challenge of translating recognition into effective repair. Hence, while models are adept at detecting user-originated errors, their ability to correct assistant-originated errors remains limited, highlighting a fundamental gap between recognition and reflective repair.

5.1.2 Self-Reflection Task

In the self-reflection setting, the model is directly involved in the dialogue and must recognize and repair its own errors within the context. As shown in Figures 5e–5h, detection remains stronger for user-originated errors than assistant-originated ones, though the gap is narrower than in the Critique task. For example, GPT o4-mini detects 85% of user errors but only 75% of assistant errors. Open-source models such as Qwen2.5-7B-Instruct also achieve solid detection rates (around 80% on user errors), while other models like Claude Sonnet and Mistral-7B struggle with assistant errors. On the other hand, the Correction accuracy is generally lower than in the Critique task: even top performers like Claude Opus and GPT o4-mini barely exceed 30% on user errors and often fall below that on assistant errors. Open-source models fare worse, sometimes failing entirely. These results show that while models can partially identify issues in real-time, being both actor and evaluator hinders effective repair. The detect and correct score reflects this compounding challenge, with many models failing to surpass 40% on average. **Thus, in self-reflective settings, models demonstrate moderate detection but remain fundamentally limited in self-corrective execution, especially for their own mistakes.**

6 Conclusion

In this work, we introduced ReflecTool-Bench, a benchmark for evaluating LLM agents' ability to engage in self-reflective tool use. Unlike prior efforts that focus on static accuracy or external critique, our benchmark emphasizes scenarios where models must detect and correct tool-use errors. Specifically, ReflecTool-Bench was constructed through a multi-agent pipeline with systematic error injection, supporting fine-grained evaluation across both Critique and Self-Reflection tasks. Experiments across various models show that while LLM agents can reliably detect user-originated errors, they struggle with correcting assistant-originated ones, especially in self-reflective settings. This gap highlights a fundamental weakness in translating recognition into effective repair.

Limitation

While ReflecTool-Bench advances the evaluation of reflective reasoning in tool-augmented dialogues, it also carries several limitations that should be acknowledged. First, the dataset is synthetically generated through a multi-agent simulation framework. This design ensures systematic control over error types and coverage, but it may not fully capture the variability, unpredictability, and noise of genuine human–assistant interactions. As a result, model performance on ReflecTool-Bench may not directly translate to real-world deployment scenarios.

Second, ReflecTool-Bench operates in a static and simulated environment rather than a sandboxed execution setting. In our benchmark, tool calls are validated against schemas but not executed in real or emulated environments. This prevents the benchmark from assessing whether a corrected call truly resolves the task end-to-end, and it limits feedback to structural validity rather than practical outcome. A sandbox environment would allow more realistic evaluation by simulating tool executions and their downstream effects.

Finally, our assessment of corrections emphasizes schema validity and tool-call accuracy, which are crucial for execution but do not capture the broader quality of the assistant's final response to the user. A corrected tool call may be technically valid but still fail to provide a satisfying or contextually helpful answer. Future work should incorporate dynamic evaluation methods that assess answer quality at the dialogue level.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2025. The claude 3 model family: Opus, sonnet, haiku. https://assets.anthropic.com/m/61e7d27f8c8f5919/original/Claude-3-Model-Card.pdf.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *ICLR*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023a. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and 1 others. 2023b. T-eval: Evaluating the tool utilization capability of large language models step by step. *arXiv preprint arXiv:2312.14033*.

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

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. In *ICML*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Yonggan Fu, Zhongzhi Yu, Junwei Li, Jiayi Qian, Yongan Zhang, Xiangchi Yuan, Dachuan Shi, Roman Yakunin, and Yingyan Celine Lin. 2024. Amoeballm: Constructing any-shape large language models for efficient and instant deployment. *arXiv preprint arXiv:2411.10606*.

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

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *NeurIPS*.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, and 1 others. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *NeurIPS*.

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. *arXiv preprint arXiv:2304.08244*.

Zheyuan Liu, Guangyao Dou, Zhaoxuan Tan, Yijun Tian, and Meng Jiang. 2024. Towards safer large language models through machine unlearning. *arXiv preprint arXiv:2402.10058*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *NeurIPS*.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, and 1 others. 2023. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Neurips*.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *NeurIPS*.

Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023a. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023b. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

RapidAPI. Rapidapi – the next generation api platform. https://rapidapi.com/.

Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, and 1 others. 2023. Tptu: Task planning and tool usage of large language model-based ai agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*.

Xiaoxi Sun, Jinpeng Li, Yan Zhong, Dongyan Zhao, and Rui Yan. 2025. Towards detecting llms hallucination via markov chain-based multi-agent debate framework. In *ICASSP*.

Zhaoxuan Tan, Qingkai Zeng, Yijun Tian, Zheyuan Liu, Bing Yin, and Meng Jiang. 2024. Democratizing large language models via personalized parameter-efficient fine-tuning. *arXiv preprint arXiv:2402.04401*.

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

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

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

TM Forum. Tm forum open api directory. https://www.tmforum.org/oda/open-apis/directory.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, and 1 others. 2023. Retroformer: Retrospective large language agents with policy gradient optimization. *arXiv preprint arXiv:2308.02151*.

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 *ACL*.

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

A Appendix: Data Creation

A.1 Generated Error Types

In Section 3.4, we introduced the error injection stage, which systematically perturbs clean dialogues to simulate realistic tool-use mistakes. Below we provide detailed descriptions of the five error types used in ReflecTool-Bench, each reflecting common failure modes in tool-augmented reasoning.

A.1.1 Parameter Mismatch

This error occurs when the assistant provides values that do not conform to the tool's expected format. Such mismatches include type errors (e.g., providing a string instead of a boolean), incorrect value ranges, or irrelevant parameter values. For example, passing `include_waitlisted="yes"` instead of `true` results in a schema violation. Parameter mismatch errors highlight whether a model can recognize inconsistencies between its tool call and the required specification.

A.1.2 Missing Required Parameter

In this case, the assistant omits a field that the tool schema designates as mandatory. For instance, an `order_status` tool call without the `orderName` parameter cannot be executed successfully. These errors are particularly important for testing whether models can notice incompleteness in their own calls and either prompt the user for clarification or repair the call proactively.

A.1.3 Incorrect Tool Selection

Here, the assistant calls the wrong tool despite clear user intent. This may stem from semantic confusion between tools with overlapping functionality or from overgeneralization during reasoning. For example, when asked to "track my package," the assistant might incorrectly call `CancelOrder()` instead of `TrackOrder()`. Such errors test whether models can align user intent with the appropriate tool under reflective evaluation.

A.1.4 Ambiguous Reference

Ambiguity errors arise when the user provides vague, underspecified, or conflicting input, leading the assistant to construct an imprecise or incorrect tool call. For example, a request like "check my last booking" may lack sufficient detail to uniquely identify the correct record. Although these originate from user behavior, the reflective challenge lies in whether the assistant can detect the ambiguity and request clarification, rather than proceeding with a faulty call.

A.1.5 Over/Under Specification

This category captures cases where either too much irrelevant detail or too little necessary information is included in the tool call. Overspecification might involve adding unnecessary parameters that constrain the result incorrectly, while underspecification leaves out details needed for accurate execution. For example, asking for "the cheapest nonstop flight arriving before 5pm with free meals and lounge access" may overload the system with conflicting constraints, whereas requesting simply "book a ticket" provides too little information. These errors test the model's ability to balance completeness with precision when repairing calls.

B Appendix: User vs. Assistant. Errors

In this section, we provide additionally analysis on models' performances across user-originated errors against assistant-originated errors. As shown in Figure 6, error detection consistently outpaces correction for almost model, reinforcing that recognition is substantially easier than reflective repair. For example, GPT o4-mini and Claude Sonnet sustain detection rates above 80% in both critique and self-reflection, yet their correction scores rarely exceed 30%. By contrast, weaker open-source models such as Mistral-7B collapse under self-reflection, with detection dropping below 20% and correction approaching zero. **This highlights that while leading API-based models maintain strong recognition capabilities, they remain limited in turning those judgments into actionable corrections.**

The asymmetry between user- and assistant-originated errors persists across models. For instance, Qwen3-32B detects nearly 90% of user errors under Critique task but drops to around 70% on assistant errors in self-reflection. Similarly, Claude Haiku shows stable user-side detection yet struggles with assistant errors, while correction accuracy remains below 20%. **This suggests that LLMs are consistently more adept at flagging vague or underspecified user inputs than recognizing and repairing their own faulty reasoning.**

Notably, some models demonstrate relative robustness compared to their peers. Qwen2.5-7B maintains balanced performance across tasks, achieving detection around 75% and correction close to 40% in self-reflection, making it the

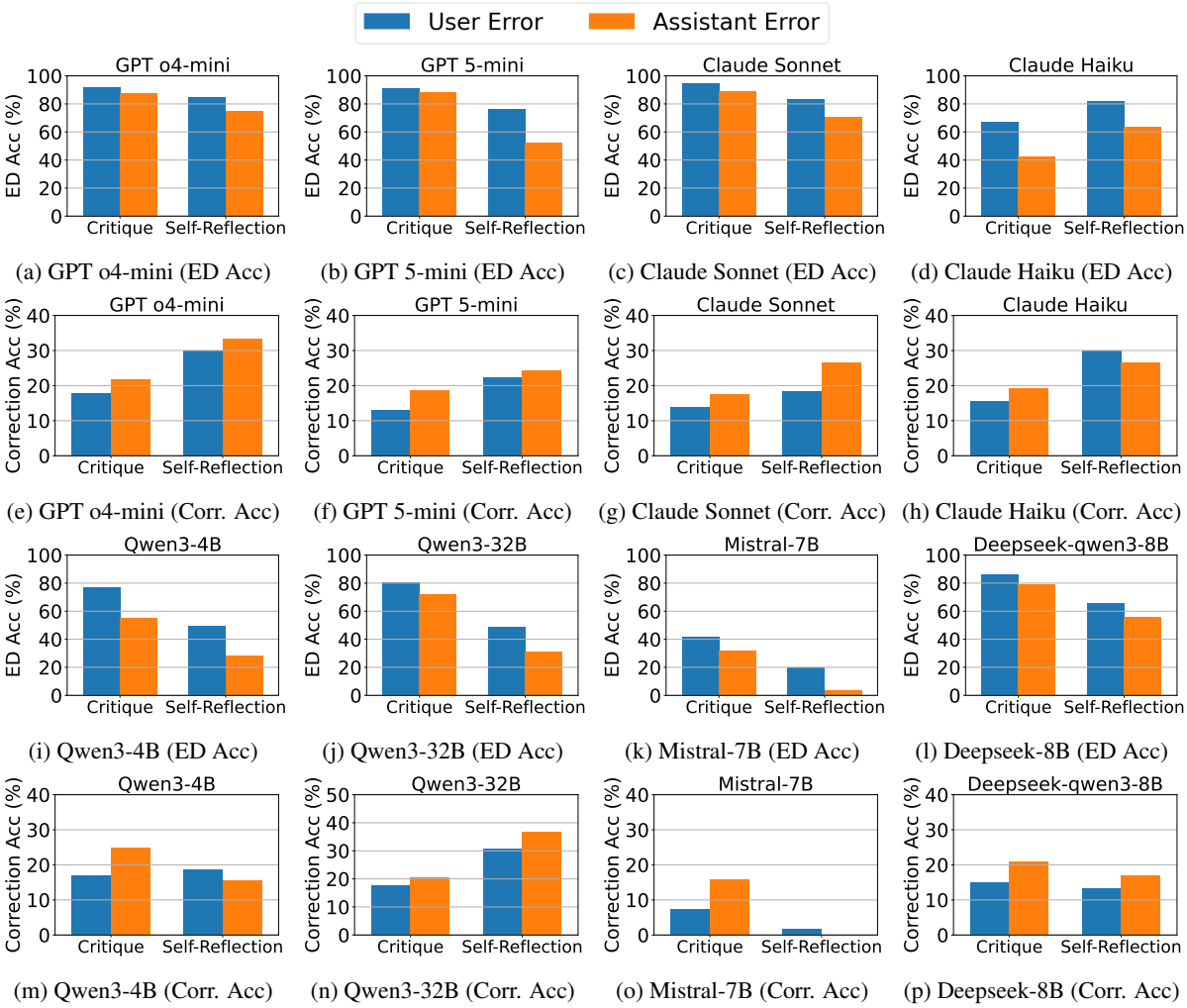


Figure 6: Error Detection (ED) accuracy and Correction accuracy across different models. In subplots 6a, 6g, 6j, 6k, 6l, 6d, 6b, 6i, the y-axis denotes the ED accuracy. In the rest of subplots, the the y-axis represents the correction accuracy. The x-axis reflects performance across two different tasks.

strongest among open-source baselines. In contrast, DeepSeek-8B and Mistral-7B show sharp degradation across both metrics, underscoring their limited reflective reasoning. Overall, while top API-based systems lead in absolute performance, even they struggle to bridge the gap between detecting and correcting errors, and open-source models suffer even more severe declines. **Thus, across models, the central bottleneck lies not in recognizing mistakes but in executing reliable self-correction.**

B.1 Error Case Analysis

To better understand how API-based models and open-source models handle different error types under both the Critique and Self-Reflection tasks, we present heatmaps in Figures 8, 9, 10, and 11. Here, PM denotes *Parameter Mismatch*, MRP stands for *Missing Required Parameter*, ITS represents *Incorrect Tool Selection*, AR indicates *Ambiguous Reference*, and O/US refers to *Over-/Under-Specification*. The heatmaps visualize model performance in terms of **Error Detection (ED)** and **Correction Accuracy** for each error type.

From these figures, we observe consistent patterns across model families. API-based models (Figures 8 and 10) demonstrate stronger and more stable performance across error types—especially in error detection. During the Critique task, top models like GPT o4-mini and Claude Opus achieve near-perfect detection for most categories, though their correction accuracy remains low across the board, underscoring the common gap between recognizing an error and applying a valid fix. This trend is even more evident under Self-Reflection, where ED accuracy drops moderately, but correction scores improve slightly—suggesting increased effort in attempting repair, albeit not always successfully. Notably, Claude Haiku shows a significant jump in correction accuracy under self-reflection, pointing to more aggressive corrections.

Example Dialogue with Tool Use and Internal Reasoning

User:	Hey, I think I have a package that was supposed to arrive a few days ago. I placed the order last week but haven't seen any updates. Can you help me figure out where it is?
Assistant (Think):	The user mentioned ordering last week but didn't specify the product name. To check the order status, I need the precise `orderName` before calling the `order_status` API.
Assistant:	Sure—could you tell me which product do you want to check?
User:	The wireless earbuds.
Assistant (Think):	We now have the product name, and I can see the records of this product, so we can call the `order_status` tool to retrieve the current status of the wireless earbuds order.
Assistant:	Let me check the status of your wireless earbuds order placed on July 3. Just a moment, please.
Tool Call:	`order_status(orderName="wireless earbuds", orderDate="July 3")`
Tool Result:	`{status: "success", orderName: "wireless earbuds", orderDate: "2025-07-03", currentStatus: "In Transit", estimatedDeliveryDate: "2025-07-10", trackingNumber: "1Z999AA10123456784", carrier: "UPS", lastUpdated: "2025-07-07T14:32:00Z", shippingHistory: [...]}`

Figure 7: Example assistant-user dialogue in the e-commerce domain involving multi-turn clarification and tool use. The assistant reasons about missing information, gathers necessary parameters, and executes the correct tool call with structured internal planning.

On the other hand, Open-source models (Figures 9 and 11) struggle more, with greater variability and overall lower scores. While some models like LLaMA 3.1-8B and DeepSeek-7B perform well on easier error types (e.g., PM or MRP) during Critique, they deteriorate sharply in both ED and correction under Self-Reflection—especially for harder categories like ITS or AR. Mistral-7B collapses almost entirely under self-reflection, failing to detect or correct even simple mistakes. However, Qwen 2.5-7B consistently emerges as a more robust open-source baseline: it maintains strong ED performance across both tasks and shows notable correction gains under Self-Reflection, even outperforming some API models in specific categories (e.g., O/US correction).

Overall, these results reinforce two key insights: (1) recognition does not imply repair—most models, even top-tier ones, exhibit a persistent drop from detection to correction; and (2) assistant-originated errors, which demand deeper reflection and tool-use reasoning, present the greatest challenge—exposing limitations in model adaptability and schema understanding. API-based models remain more reliable, but closing the correction gap remains a central challenge across the board.

C Appendix: Evaluation Decomposition

C.1 Reasoning Quality

In addition to correctness-based metrics, we evaluate the quality of the model's explanation for its decision. Each explanation is rated on a 1–5 scale based on two criteria: *correctness*, which assesses how accurately the explanation aligns with the ground truth reasoning, and *helpfulness*, which measures its clarity, informativeness, and usefulness for understanding or correcting the error.

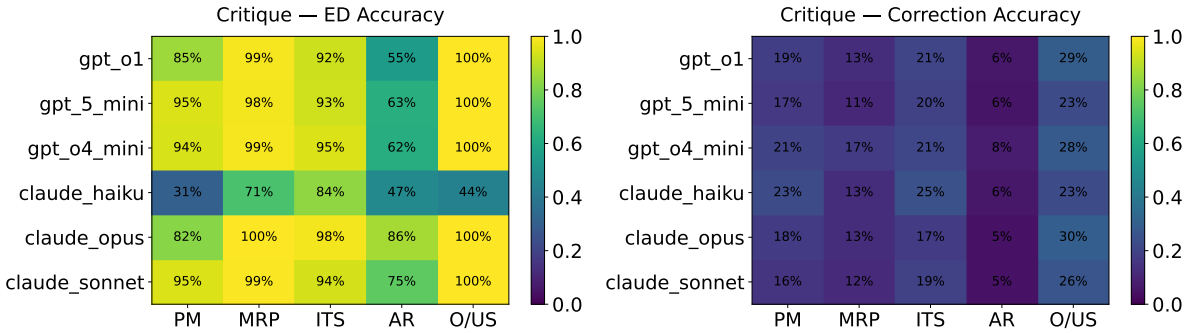
A score of 5 indicates an explanation that is fully accurate, comprehensive, and clearly articulated—closely matching the reasoning provided in the ground truth reference. A score of 4 reflects a mostly correct and helpful explanation, with only minor issues in clarity or completeness. A score of 3 represents a partially correct explanation that is somewhat helpful but either incomplete or misaligned in key aspects. Scores of 2 and 1 reflect increasingly poor performance, with a score of 1 assigned to explanations that are incorrect, misleading, or entirely unhelpful. Scoring is performed automatically using GPT-based evaluation with a fixed rubric, and we report the average explanation score separately for user-induced and assistant-induced errors. The prompt for reasoning evaluation under Critique task and self-reflection task is shown in Figure 12 and Figure 13, respectively.

D Multi-Agents Prompting Strategy

Here, we display the prompting strategies that we have leveraged to create the multi-agent systems.

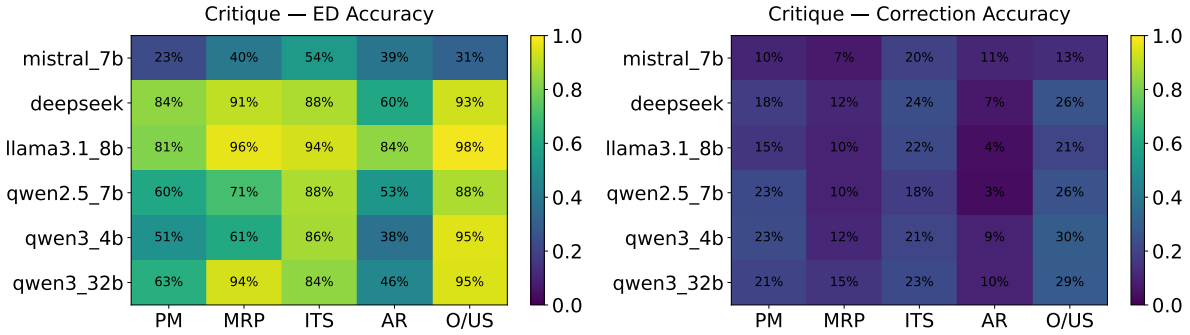
D.1 Description Agent

The detailed prompting strategy for description agent is shown in Figure 14.



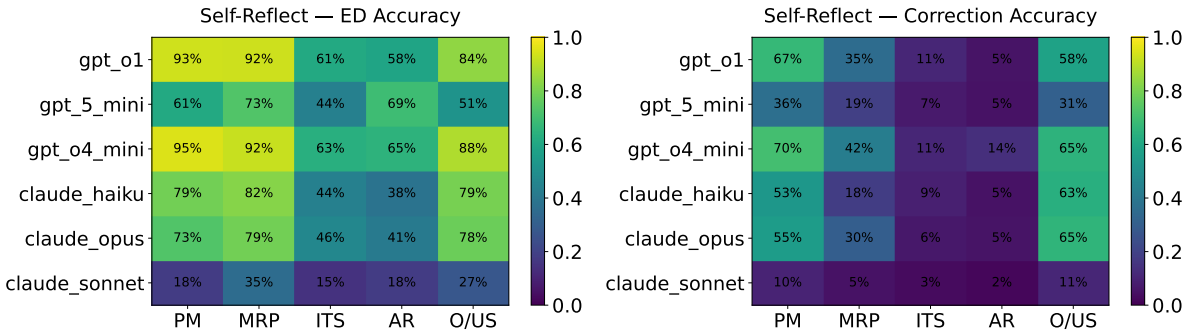
(a) API-Based Model Critique (ED Acc) (b) API-Based Model Critique (Corr. Acc)

Figure 8: Critique task error case analysis for API-based models. The x-axis indicates different error types, while the left y-axis lists model names and the right y-axis shows corresponding accuracy scores. Cell shading encodes performance: lighter colors indicate higher accuracy, and darker colors represent lower accuracy.



(a) API-Based Model Critique (ED Acc) (b) API-Based Model Critique (Corr. Acc)

Figure 9: Critique task error case analysis for open source models. The x-axis indicates different error types, while the left y-axis lists model names and the right y-axis shows corresponding accuracy scores. Cell shading encodes performance: lighter colors indicate higher accuracy, and darker colors represent lower accuracy.



(a) API-Based Model Critique (ED Acc) (b) API-Based Model Critique (Corr. Acc)

Figure 10: Self-Reflect task error case analysis for API-based models. The x-axis indicates different error types, while the left y-axis lists model names and the right y-axis shows corresponding accuracy scores. Cell shading encodes performance: lighter colors indicate higher accuracy, and darker colors represent lower accuracy.

D.2 User Agent

The detailed prompting strategy for User agent is shown in Figure 15 and Figure 16.

D.3 Assistant Agent

The detailed prompting strategy for Assistant agent is shown in Figure 17.

D.4 User-Error Agent

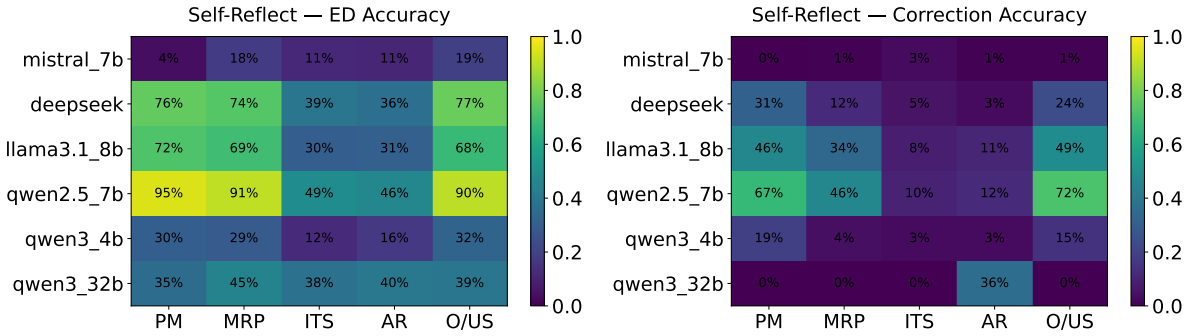
The detailed prompting strategy for User-Error agent is shown in Figure 18.

D.5 Assistant-Error Agent

The detailed prompting strategy for Assistant-Error agent is shown in Figure 19.

E Appendix: Failed Case Analysis

It is noteworthy that even stronger models such as GPT-o1 and Claude Opus often failed to fully correct tool-use mistakes, despite successfully detecting them in many cases. This highlights a persistent gap between *error recognition* and *error repair*. To better illustrate this limitation, we present



(a) API-Based Model Critique (ED Acc)

(b) API-Based Model Critique (Corr. Acc)

Figure 11: Self-Reflect task error case analysis for open source models. The x-axis indicates different error types, while the left y-axis lists model names and the right y-axis shows corresponding accuracy scores. Cell shading encodes performance: lighter colors indicate higher accuracy, and darker colors represent lower accuracy.

an abbreviated set of failed cases from ReflecTool-Bench. These examples demonstrate how models frequently produce invalid or incomplete tool calls, fail to adhere to schema requirements, or hallucinate tools altogether. We classify the errors into four representative categories.

E.1 Hallucinated Tools

In some cases, the model invokes a non-existent tool rather than the correct one. As shown in Figure 20, the assistant chose `get_registered_courses` instead of the correct `list_classes`, producing incomplete or invalid results. Such hallucinations reveal a failure to ground tool calls in the provided schema, even when the user intent is clear.

E.2 Invalid or Malformed Tool Calls

Another common error type arises when the model generates a tool call with incorrect parameter formats or values. In Figure 21, the assistant propagated a user's mistake by passing the string `"yes"` instead of the expected boolean `true`, resulting in a type mismatch. This demonstrates how models struggle to enforce schema constraints when user inputs are noisy.

E.3 Missing API Call Fields

Models also fail by omitting required tool-call fields. As shown in Figure 22, the assistant bypassed the API entirely and produced a direct textual response, despite the schema requiring a structured tool invocation. Such omissions often occur when the model prioritizes surface-level fluency over adherence to execution protocols.

E.4 Redundant or Irrelevant Follow-up Questions

Finally, we observe errors where the assistant introduces irrelevant or redundant questions instead of repairing the original mistake. Figure 23 illustrates a case where the assistant repeated the user's query in natural language rather than producing the corrected tool call. These behaviors reflect a tendency to stall or divert rather than engage in genuine self-correction.

Overall, these examples highlight the challenges that remain even for advanced LLMs. While detection of user-side mistakes is often successful, assistant-originated errors persist, with corrections frequently malformed, incomplete, or missing. By curating and categorizing these cases, ReflecTool-Bench provides a valuable lens for analyzing failure modes in reflective reasoning.

Reasoning Evaluation Prompt — Critique task

You are an expert evaluator reviewing an assistant's explanation toward an API usage case in the "{domain}" domain. Please rate the explanation for its **correctness** and **helpfulness** on a scale from 1 to 5, using the provided reference explanation as context.

=== Definitions ===

- **Correctness**: Does the explanation align with the provided ground truth? Are there logical errors, false claims, or critical omissions?
- **Helpfulness**: Is the explanation clear, precise, and actionable for understanding the issue? Does it guide toward identifying or fixing the error?

=== Scoring Guidelines ===

- **1 (Poor)**: Completely incorrect or misleading; no alignment with the ground truth; unhelpful.
- **2 (Fair)**: Contains major inaccuracies or is too vague to be helpful. Partial alignment but still inadequate.
- **3 (Average)**: Partially correct and helpful. Acceptable, but incomplete or misaligned.
- **4 (Good)**: Mostly correct and helpful. Aligns well with the ground truth, with only minor flaws.
- **5 (Excellent)**: Fully correct, clear, and highly helpful. Closely matches the ground truth with strong reasoning.

—

Ground Truth Reference Explanation: "{groundtruth_error_explanation}"

Assistant's Explanation to be Scored: "{explanation}"

—

Reply with a **single integer (1–5)** only. Do not include any additional text or commentary.

Figure 12: Prompt used in the Critique task for reasoning evaluation. An expert judge scores the assistant's explanation for correctness and helpfulness based on a reference explanation.

Self-Reflection Evaluation Prompt

You are an expert evaluator assessing an AI assistant's performance in a task-oriented dialogue. You are given: 1. The full dialogue history between user and assistant. 2. The assistant's final structured response (including <think>, <api_call>, <response>, etc.). 3. A ground-truth tool call and explanation describing what the assistant should have done.

—

=== Criteria ===

1. Error Detection (Yes/No) Did the assistant correctly identify that an error (by either the user or assistant) occurred?

- Check the <think>, <api_call>, and <response> blocks for evidence of recognition.
- If the assistant identifies the mistake in <think> but does not attempt a correction, still mark Error Detection as "Yes", but mark Correction Accuracy as "No".
- If no indication of error is present in any part, mark Error Detection as "No".

2. Correction Accuracy (Yes/No) Did the assistant provide a correct revised <api_call> based on the identified error?

- Answer "Yes" only if the assistant (1) detected the error and (2) attempted a fix via <api_call>.
- The revised tool call must match the ground truth in both structure and semantics.
- Be strict: mismatches in casing, field names, or formats should count as "No".

3. Reasoning Score (1–5) Evaluate the assistant's explanation inside <think>.

- 5: Excellent — deep, precise reasoning clearly aligned with ground truth.
- 4: Good — mostly correct, minor issues in clarity or depth.
- 3: Average — partially correct but incomplete or generic.
- 2: Poor — vague, unhelpful, or logically flawed.
- 1: Very Poor — incorrect or nonsensical.

—

Dialogue History: {dialogue_history (JSON-formatted)}

—

Assistant Response: {model_reply}

—

Ground Truth Tool Call: "{ground_truth_call}"
Ground Truth Explanation: "{groundtruth_error_explanation}"

—

Reply with a JSON object in the following format only:

```
{
  "error_detected": "Yes" or "No",
  "correction_accurate": "Yes" or "No",
  "reasoning_score": 1-5
}
```

Figure 13: Evaluation prompt for scoring assistant behavior in the Self-Reflection Task. Evaluators assess error detection, correction accuracy, and reasoning quality from the assistant's structured response.

Description Agent Prompt with Integrated Examples

You are an expert in designing realistic and high-impact tool functions for AI assistant applications in the e-commerce domain: **{domain}**, specifically the subdomain: **{subdomain}**.
Your task: - Generate a tool **name** in CamelCase (concise and descriptive). - Provide a realistic and assistant-aware **tool_description** in 1–2 sentences. - Avoid assuming users always provide technical IDs (like order ID or ASIN). - Assume assistants may have access to prior user context (e.g., past orders, viewed items, account preferences). - Design descriptions that reflect real conversations where users describe needs naturally (e.g., "I bought a coffee table last week").
Do NOT repeat or slightly modify these functionalities. Instead, diversify the scope by introducing creative, complementary tools that align with natural user behaviors.
Output a single JSON object with the fields: tool_name, tool_description, and subdomain.
Below are several examples:

- **Example 1:**
  { "tool_name": "TrackOrderStatus",
  "tool_description": "Fetches delivery and shipping status of a recent order, using either product name, delivery window, or order ID if needed. Designed for assistants with session context.",
  "subdomain": "order_management" }
- **Example 2:**
  { "tool_name": "CancelPendingOrder",
  "tool_description": "Allows users to cancel their own recent orders if still in pending or processing state. Uses inferred order details from user context when possible.",
  "subdomain": "order_management" }
- **Example 3:**
  { "tool_name": "RefineProductSearch",
  "tool_description": "Helps users narrow product results by filtering based on features like price range, brand, rating, or delivery speed.",
  "subdomain": "product search" }
- **Example 4:**
  { "tool_name": "SuggestSimilarItems",
  "tool_description": "Given a product description or previously viewed item, this tool recommends similar alternatives based on user interests and product metadata.",
  "subdomain": "product search" }

Now generate a NEW tool under subdomain **{subdomain}**, avoiding overlap with the tools above.

Figure 14: Prompt used for the Description Agent. The agent is instructed to generate novel tools within a subdomain, guided by natural user behavior and realistic assistant capabilities. Few-shot examples are embedded for clarity.

User Agent Prompt — Initial Request

You are simulating a user speaking to a virtual assistant.
You are in the subdomain: "{domain}" with the following tools:
{ {tool_schema (in JSON format)} }
Your task: - Generate a realistic and varied **first message** a user might say in this context. - Avoid generic openings like "Hi, I have a question...". - The request should feel natural, a little vague, and conversational to encourage the assistant to ask clarifying questions. - Return only the user's message in **plain text**, without JSON or Markdown formatting.

Figure 15: Prompt used for the User Agent to generate the initial user message in a tool-driven conversation. The agent is encouraged to simulate under-specified but realistic behavior.

User Agent Prompt — Follow-up Turn

```
You are simulating the user in an ongoing dialogue in the {subdomain} under the {domain} domain.
- This is Turn {turn_id} out of {num_turns} total. - The assistant just said: {assistant_content}
- Previously, you said: {user_reminder}
Your task: - Generate the user's next message in response to the assistant. - Respond directly,
naturally, and in a human-like tone. - If the assistant asked for missing info, just give it
(synthesize something plausible). - DO NOT include technical identifiers like order IDs, user
IDs, etc. - DO NOT return JSON or Markdown formatting — just a plain user message. - Try to
diversify your request so the assistant cannot just reuse the same tool repeatedly.
Here are the tools available to the assistant. Your reply should align with what these tools
can address:
{ {tool_schema (in JSON format)} }
```

Figure 16: Prompt for the User Agent to generate follow-up user replies during multi-turn dialogue. The user is simulated to be cooperative but natural, and variation is encouraged.

Assistant Agent Prompt — Step-by-Step Tool Planning

You are a highly capable assistant specialized in the domain of {domain}. Your task is to resolve the user's request step-by-step using **ONLY** the APIs provided below:
{ {tool_schema (in JSON format)} }
=== RULES ===

- Gather missing details naturally by asking clarifying questions (only ask for **one detail per turn**).
- Prioritize required parameters before optional ones.
- **Do NOT** call a tool until all required parameters are available.
- Avoid calling tools too early; focus on gathering context in early turns.
- If only one tool is provided:
  - Use the tool only at the **final turn**.
- If multiple tools are provided:
  - Use at least **two different tools** in the conversation.
  - Always call tools in the **final turn** once all info is gathered.
- NEVER invent tools, parameters, or user data.
- DO NOT ask for user IDs (e.g., order number, user ID, etc.).
- If the user gives vague dates (e.g., "last week"), ask them to clarify.
- If the user says their issue is resolved, end politely.
- NEVER leave the conversation unresolved.

=== RESPONSE FORMAT ===
Return a valid JSON object with the following fields:
{ "think": "Explain your internal reasoning here.", "content": "What you say directly to the user.", "tool_call": { "name": "<tool_name>", "parameters": { "param1": "...", "param2": "..." } } }
=== EXAMPLE (Non-final turn) ===
{ "think": "The user asked about their package delivery date. I need their order date before I can call the TrackOrderStatus tool.", "content": "Could you tell me when you placed the order so I can check its status for you?" }
=== EXAMPLE (Final turn with tool call) ===
{ "think": "I now have the order date and product name. Time to call TrackOrderStatus to fetch the delivery date.", "content": "Let me check the current delivery status for your order.", "tool_call": { "name": "TrackOrderStatus", "parameters": { "orderName": "headphones", "orderDate": "2025-06-01" } } }
=== CONTEXT ===

- Turn {turn_id + 1} of {num_turns}
- User's original goal: "{user_goal}"
- Tools already used: {used_tools}

Think carefully. Ask questions early to collect missing info. Only call tools at the **right moment**.

Figure 17: Prompt for the Assistant Agent to conduct step-by-step tool planning. The agent must reason before acting, delay tool use until all required parameters are collected, and respond in structured JSON.

User Error Injection Agent Prompt

You are an expert assistant for simulating realistic **user-side** mistakes in tool-based interactions.
Your task is to inject **plausible user-caused errors** (e.g., vague requests, misstatements, incorrect formats) into an otherwise valid assistant-user dialogue. These user-originated errors should lead to common tool call failures such as parameter_mismatch, missing_required_parameter, and others.
Key constraints:

- Use the **EXACT SAME** tool call function and base dialogue as the given planner sample.
- Do **NOT** create or modify tool functions beyond the ones provided.
- Modify 1-2 user turns while preserving realistic dialogue structure.
- The assistant should remain cooperative and unaware of the user's mistake.
- Tool errors must stem from user behavior; assistant reasoning should remain sound and plausible.

Each output must include:

- "error_type": the type of user-caused error.
- "dialogue": realistic conversation turns with assistant think, content, and optional tool_call and tool_call_result.
- "groundtruth_tool_call": the correct tool call based on an ideal user query (should match planner sample).
- "Explanation": what user did wrong, why the assistant misinterpreted it, and why the tool call failed.

Supported Error Types (caused by user behavior):

- **parameter_mismatch**: User gives incorrect or ambiguous values (e.g., "yesterday" for date).
- **missing_required_parameter**: User omits required fields (e.g., product name).
- **incorrect_tool_selection**: User's phrasing leads assistant to call wrong tool (e.g., says "return" when they meant to cancel).
- **vague_query / ambiguous_reference**: User refers to "it", "that thing" or uses unclear descriptors.
- **over / under-specification**: User gives too much (unsupported) or too little (non-informative) detail.

Final Output Format (one per error type): Return a list of JSON objects, one for each error type, with the structure shown above.
Reference Planner Sample (for modification):

{planner_sample}

Reminder: Do not modify the tool call function. Inject errors only into user dialogue content.

Figure 18: Prompt used by the User Error Injection Agent to create error-injected dialogue samples caused by vague or faulty user input. The assistant logic must remain sound.

Assistant Error Injection Agent Prompt

You are an expert assistant for simulating realistic **assistant-side** failures in tool-based interactions.
Your task is to inject **plausible mistakes made by the assistant** during a tool-driven conversation, even when the user input is clear and complete.
Key Constraints:

- Use the **EXACT SAME** tool call function and dialogue structure as the provided planner sample.
- Do **NOT** create or modify tool functions beyond the given schema.
- Inject the error into the assistant behavior, not the user.
- The assistant should appear logical and cooperative — it must not be aware that it made a mistake.

Each output must include:

- "error_type": the type of assistant-caused error.
- "dialogue": naturalistic dialogue with one tool call at the final assistant turn.
- "groundtruth_tool_call": the correct tool call from the planner sample.
- "Explanation": what the assistant misunderstood, and why the tool call was incorrect.

Supported Error Types (caused by assistant behavior):

- **parameter_mismatch** — Assistant misformats or misinterprets parameter values (e.g., uses a boolean instead of a string).
- **missing_required_parameter** — Assistant fails to include a required field, even though the user provided it.
- **incorrect_tool_selection** — Assistant selects a tool that does not match the user's intent.
- **ambiguous_reference** — Assistant proceeds with assumptions without clarification.
- **over / under-specification** — Assistant adds unsupported fields or omits relevant optional ones.

Reference Planner Sample:

{planner_sample}

Reminder: Only modify assistant behavior — the user dialogue remains correct. Ensure assistant's internal logic remains plausible even when tool usage is incorrect.

Figure 19: Prompt used for the Assistant Error Injection Agent. The assistant is guided to simulate realistic tool-use errors such as parameter formatting issues or incorrect tool selection, while maintaining natural and cooperative behavior.

Error Analysis: Hallucinated tool

User:	What courses am I registered for in Fall 2025?
Assistant (Think):	The user wants to see their course registrations for Fall 2025. I should call the course listing tool with the correct term.
Ground Truth:	`list_classes(term_name="Fall 2025")`
Model Generation:	`get_registered_courses(term="Fall 2025")`

Figure 20: Error caused by hallucinated tool. Assistant error caused by choosing a non-existent tool `get_registered_courses` instead of the correct `list_classes`. This results in incomplete or invalid results.

Error Analysis: invalid or malformed tool calls

User:	I'm trying to figure out what courses I've got lined up next semester—could you include the ones I'm on the waitlist for too?
Assistant (Think):	The user asked for their class schedule in Fall 2025 and also requested to include waitlisted courses. I will call the tool with the correct parameters.
Ground Truth:	`list_classes(term_name="Fall 2025", include_waitlisted=true)`
Model Generation:	`list_classes(term_name="Fall 2025", include_waitlisted="yes")`

Figure 21: Error caused by invalid or malformed tool calls. User error due to passing a string `"yes"` instead of a boolean `true`. The assistant propagated this mistake, resulting in a type mismatch.

Error Analysis: Missing API Call field

User:	I'm trying to figure out what courses I've got lined up for Fall 2025—could you include the ones I'm on the waitlist for too?
Assistant (Think):	The user is asking about their Fall 2025 schedule and wants to include waitlisted courses. I'll retrieve this using the appropriate tool.
Ground Truth:	`list_classes(term_name="Fall 2025", include_waitlisted=true)`
Model Generation:	`Here's your class schedule for Fall 2025: 1) ENGL 205: Introduction to Creative Writing (Section 01) "Credits: 3 ....`

Figure 22: Error caused by missing API call field. The assistant skips the tool calling field and directly produces an answer, which is fabricated.

Error Analysis: Redundant or irrelevant follow-up questions

User:	I'm trying to figure out what courses I've got lined up for Fall 2025—could you include the ones I'm on the waitlist for too?
Assistant (Think):	Formatting the final response with the schedule details from the API output.
Ground Truth:	`list_classes(term_name="Fall 2025", include_waitlisted=true)`
Model Generation:	`Just to confirm—would you like me to include the waitlisted courses as well?`

Figure 23: Error caused by redundant or irrelevant follow-up questions. The assistant unnecessarily asks about including waitlisted courses, which the user already specified. This leads to wasted interaction turns and delays task completion.