TOOLSANDBOX: A Stateful, Conversational, Interactive Evaluation Benchmark for LLM Tool Use Capabilities

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer
Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li,
Guoli Yin, Zirui Wang, Ruoming Pang
Apple
{jiarui_lu, tholleis, yizhe_zhang, baumayer
f_nan, haoping_bai, shuang_ma2, sma7, mengyu_li2
gyin, ziruiw, r_pang}@apple.com

Abstract

Recent large language models (LLMs) advancements sparked a growing research interest in tool assisted LLMs solving real-world challenges, which calls for comprehensive evaluation of tool-use capabilities. While previous works focused on either evaluating over stateless web services (RESTful API), based on a single turn user prompt, or an off-policy dialog trajectory, TOOLSANDBOX[1] includes stateful tool execution, implicit state dependencies between tools, a built-in user simulator supporting on-policy conversational evaluation and a dynamic evaluation strategy for intermediate and final milestones over an arbitrary trajectory. We show that open source and proprietary models have a significant performance gap, and complex tasks like State Dependency, Canonicalization and Insufficient Information defined in TOOLSANDBOX are challenging even the most capable SOTA LLMs, providing brandnew insights into tool-use LLM capabilities.

1 Introduction

	State Dependency	Conversational	Interactive	Human Authored Ground Truth
TOOLSANDBOX	✔	✔	✔	✔
BFCL	✗	✗	✗	✔
ToolEval	✗	✗	✔	✗
API-Bank	✗	✔	✗	✔
ToolTalk	✗	✔	✔	✗
τ-bench	✗	✔	✔	✗

Table 1: A comparison of TOOLSANDBOX and other tool-use benchmarks.

Recent advancements in Large Language Models (LLMs) brought forth new opportunities treating LLMs as autonomous agents, capable of observing real-world environments and deciding upcoming actions. Among which, tool-use agents (Schick et al., 2023; Qin et al., 2023a; Patil et al., 2023; Qin et al., 2024) follow human instructions and utilize

real-world APIs to complete complex tasks. Contrary to prior approaches like dialog state tracking (Henderson et al., 2014; Budzianowski et al., 2018; Rastogi et al., 2020), which require the model to explicitly generate dialog states and actions under a predefined ontology, and derive a tool call from those structured outputs, tool-use studies allow the model to directly generate tool calls based on its observations, while keeping dialog and world state tracking implicit.

Despite the paradigm shift towards a more simplified problem formulation, the **stateful**, **conversational** and **interactive** nature of task oriented dialog remains, and poses a significant challenge for systematic and accurate evaluation of tool-using LLMs. Existing benchmarks like the Berkeley Function Calling Leaderboard (BFCL) (Yan et al., 2024), ToolEval (Qin et al., 2024), API-Bank (Li et al., 2023), ToolTalk (Farn and Shin, 2023) and τ-bench (Yao et al., 2024) attempted to tackle some of these challenges, but there is yet to be an all encompassing solution.

Stateful Task oriented dialog often involves tools that are strongly coupled with a *World State*, e.g. a database. This can be a tool that can alter the world state, like turning on internet connection. More interestingly, there can be a tool that implicitly depends on a world state, for example, one cannot search for a nearby restaurant when internet connection is off. Sometimes, actions that deal with both of these scenarios need to be taken to complete a task, even if the user is agnostic to the underlying world state and only gives general instructions. The agent needs to use its own knowledge about the world and environment feedback to come up with a plan to modify the world state and complete the task. An example can be found in Figure 1.

BFCL (Yan et al., 2024) and ToolEval (Qin et al., 2024) both rely on stateless tools interacting with web services (through RESTful APIs). As such,
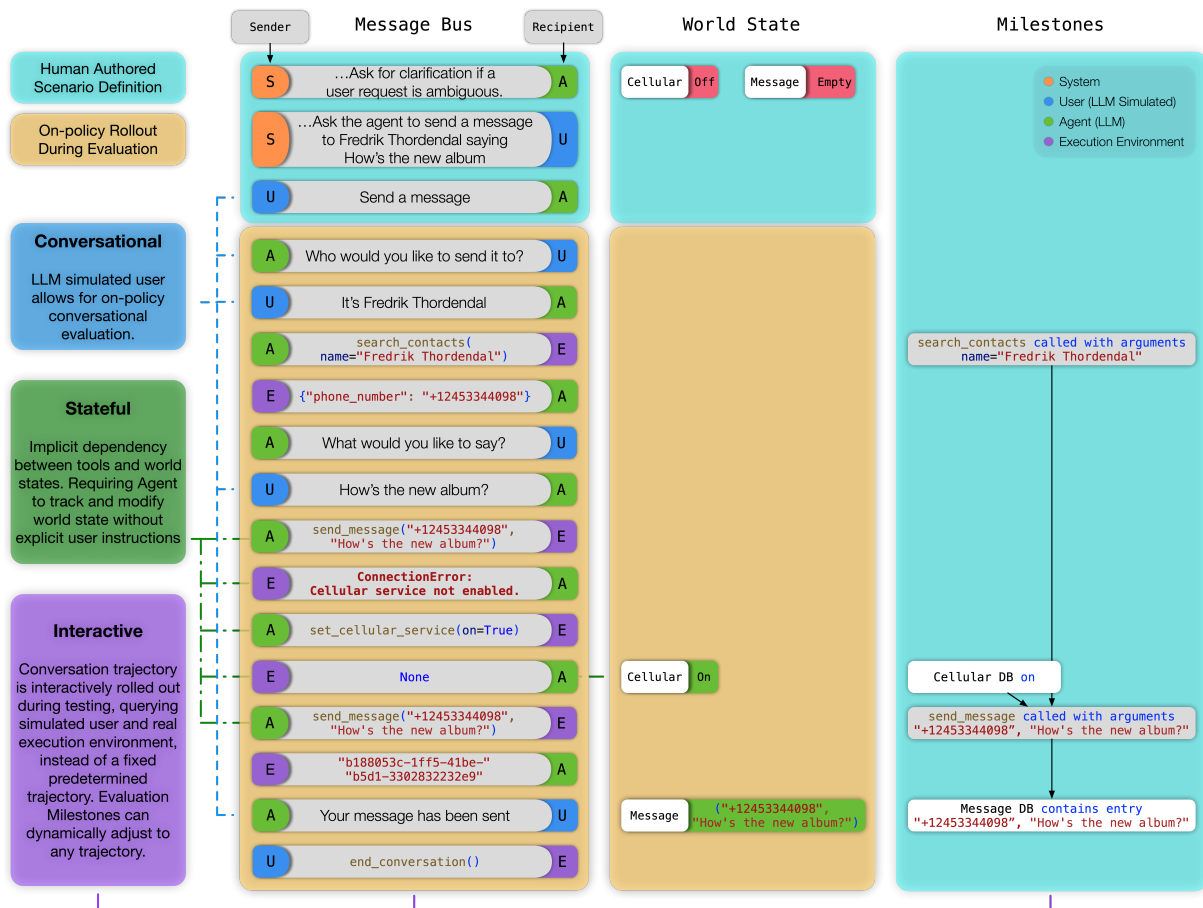

Figure 1: An example evaluation trajectory from TOOLSANDBOX. Some message contents and milestones were truncated and streamlined for visual clarity. The *Message Bus* represents a full dialog history between the *User*, the *Agent* and the *Execution Environment*. The *World State* represents mutable database snapshots at a given turn. The *Milestones* represent predefined key events that need to happen in this trajectory. The light blue boxes represent the human authored scenario definition, including the user goal, initial user utterance, initial world state and milestones. The light yellow boxes represent on policy rollout collected during an interactive evaluation run. In this example, the *User* intended to send a message, while cellular service is turned off. The *Agent* should first understand the *User*'s intent, and prompt for necessary arguments from the *User*. After collecting all arguments with the help of the *search_contacts* tool, the *Agent* attempted to send the message, figured out it needs to enable cellular service upon failure, and retried. To evaluate this trajectory, we find the best match for all *Milestones* against *Message Bus* and *World State* in each turn while maintaining topological order.

these evaluation benchmarks are designed to assess how agents make trials with a static environment. API-Bank (Li et al., 2023), ToolTalk (Farn and Shin, 2023) and τ-bench (Yao et al., 2024) does include a set of tools to modify world states, but does not study the impact of state dependencies.

Conversational   Conversational evaluation is crucial yet challenging when assessing a dialog policy, due to the interdependency between a user and said policy, as well as the ambiguous nature of natural language. To facilitate automated conversational evaluation, a common practice is to implement a simulated user (Zhang et al., 2024; Sekulic et al., 2024). However, BFCL and ToolEval only evaluate self-contained, unambiguous single-turn

user queries, which is hardly realistic. API-Bank and ToolTalk evaluates on unrolled predefined off-policy dialog trajectories, and thus cannot assess the agent's performance based on its own policy.

Interactive   Real world scenarios are full of surprises. The agent could issue an erroneous tool call. Tool execution could raise an unexpected exception. And the user could issue a follow-up correcting a previous statement. An interactive evaluation framework assessing the immediate return of key interactions with user or environment would be necessary to capture the intricate interaction between different roles. Such an interactive evaluation should provide full spectrum and fine-grained evaluation of any multi-turn session. In

this regard, BFCL, API-Bank and ToolTalk rely on a predefined trajectory, and by extension relying on static turn wise evaluation metrics. τ-bench requires agent action to match a single predetermined sequence, leaving no room for error correction in the model, or multiple possible answer sequences. Even though ToolEval allows multiple rounds of interaction between the Agent and tools, it relies solely on an LLM evaluator to judge the final pass rate and win rate of trajectories, which raises questions to its reliability and interpretability.

Driven by these motivations, we propose TOOL-SANDBOX, a stateful, conversational and interactive tool-use benchmark. To the best of our knowledge, TOOLSANDBOX is the first LLM tool-use benchmark which

- Includes implicit state dependencies between stateful tools, allowing the agent to track and alter the world state based on its world/commonsense knowledge, which is implicit from the user query;
- Includes an LLM simulated user, allowing for realistic, on-policy conversational evaluation to measure the agent's ability on implicit dialog state tracking;
- Allows for fully interactive, dynamic trajectory collection with a representative set of highly composable tools, and a human authored, milestone / minefield based system for intermediate and final execution evaluation.

A comparison between TOOLSANDBOX and other benchmarks can be found in Table 1.
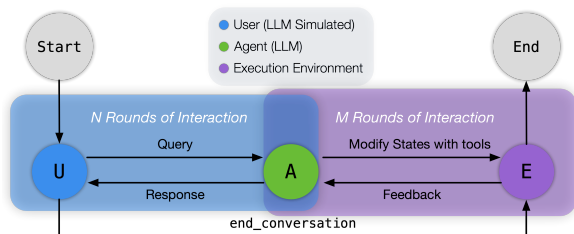
2 TOOLSANDBOX Design

Figure 2: Interaction between the User, Agent and the Execution Environment. Boxes represent multiple rounds of interaction between involved roles.

At its core, TOOLSANDBOX is a Python native LLM testing environment, with *Execution Context* as world state abstraction and Python functions as *Tools*, where *User*, *Agent* and *Execution Environment* communicate with each other through a

Message Bus to complete a task, which is evaluated against predefined *Milestones* and *Minefields*. As shown in Figure 2, a typical test case starts with the *User* speaking to the *Agent*. From then on, the role being addressed gets to speak next, until the end state is reached. Upon receiving a *User* request, an *Agent* can decide to respond to the *User* asking for more information, or inform the *Execution Environment* to execute a *Tool*, providing intended tool name and arguments. The *Execution Environment* executes the *Tool* in an *code.InteractiveConsole*, (Foundation, 2024), which depending on the *Tool* modifies the world state stored in the *Execution Context*, and responds to the *Agent*. Once the *User* decides the task has been completed, it informs the *Execution Environment* to execute the end_conversation tool, which puts the system in the end state, ready to be evaluated based on the dialog's similarity to *Milestones* and *Minefields*. A thorough walk-through of TOOLSANDBOX implementation details can be found in Appendix A. The remainder of this section focuses on the key components that enables **stateful**, **conversational** and **interactive** evaluation.

2.1 Stateful

To construct challenging reasoning scenarios, TOOLSANDBOX includes a set of carefully designed stateful tools, defined as tools that inspects, depends on or manipulates world states. These world states include:

Cellular service Tools that require cellular service (e.g., send_message) depend on it to be true.

Wifi RapidAPI tools (e.g., search_stock) depend on it to be true.

Location Service All tools that utilize current location (e.g., get_current_location) depend on it to be true.

Low battery mode All tools that turn on cellular service, wifi, and location service status depend on it to be false, creating nested state dependency.

Stateful tools cover 44% of TOOLSANDBOX toolbox. Naturally these tools forms an implicit dependency between each other, testing the agent's ability to maintain an internal image of world states and tool call stack. As an example, in Figure 1, when send_message tool is called while cellular service is off, a ConnectionError is raised, and the *Agent* should utilize set_cellular_service

to resolve the error, understand that cellular service should be turned on now, and retry `send_message`.

2.2 Conversational

On-policy conversational roll-out is supported by an LLM (GPT-4o) powered user simulator and carefully calibrated prompting design. The simulator represents a human interacting with an Agent, hoping to complete a task through possibly multiple rounds of conversation. When the User simulator decides the task has been completed, or could not be completed, it can terminate the conversation using the `end_conversation` tool, which is the single tool available to it. As related studies in user simulation (Zhang et al., 2024; Sekulic et al., 2024) suggest, one should include the user's overall goal in the simulator's system prompt. However, we found this is often insufficient for the complex interactions in TOOLSANDBOX, and can lead to two categories of failures. In some cases, it is infeasible for an LLM simulated user to judge task completion, or provide follow-up information with only access to the user goal, and not the expected result, which could lead to hallucination. Also, with only a single system prompt, the simulated user could be derailed by the tool-use agent, failing to follow instructions. Examples of these failures can be found in Appendix A.4.

In light of this, we propose two additional components in user simulator prompts: *Knowledge Boundary*, which inform the user simulator what it should and should not know, providing partial access to expected result, combating hallucination. And *Demonstration*, which provides few shot example dialogs to the user simulator. Prompt examples can be found in Appendix A.4. Note that demonstration is only visible to the user simulator and not the agent. We performed an ablation study for these components in Table 2. With both approaches combined, the LLM simulated user achieves the lowest error rate in all categories. User simulator error rate is also found to be consistent across agents, shown in Table 3, which should not affect the agent accuracy comparison in Table 5.

2.3 Interactive

With an stateful, conversational and interactive environment, evaluation trajectories are highly dynamic. Multiple trajectories can lead to the same outcome. A given task may be completed using different tools, the same tools in a different order, or through trial and error, and the evaluation strat-

	Hallucination ↓	IF ↓
User Goal	12.4	6.20
+ Knowledge Boundary	7.75	3.88
+ Demonstration	**6.97**	**0.77**

Table 2: Percentage of user simulation failures in each failure category for each user simulator prompting setup. IF stands for instruction following error. Statistics derived from 1032 manually annotated trajectories using GPT-4o user simulator and GPT-4o agent.

	GPT-4o	Claude-3-Opus	Gemini-1.5-Pro
Hallucination	6.90±1.45	6.40±0.97	7.15±0.71
IF	1.11±0.84	1.38±0.69	0.92±0.49
Total Error	8.02±1.36	7.78±0.52	8.07±1.03

Table 3: Percentage of user simulation failures in each failure category for each agent model. Mean and std collected from 4 repeated trials each containing 1032 trajectories. Both Knowledge Boundary and Demonstration are applied.

egy has to be flexible enough to accommodate for that. To combat this, we developed an evaluation strategy based on *Milestones* and *Minefields*, which defines key events that must or must not happen in a trajectory, allowing us to evaluate any trajectory with rich intermediate and final execution signals, providing deeper understanding of the model performance. An example can be found in Figure 10.

In specific, *Milestones* are the critical steps needed to achieve a goal. An example is shown in Figure 1, where cellular service is turned off and the user asks the agent to send a text message. The milestones, in this example, would be defined as:

1. The cellular status in the settings database must be changed to `True`.
2. The Agent must issue a tool call using the `search_contacts` tool and the correct arguments, before or after milestone 1.
3. The Agent must issue a tool call using the `send_message` tool and the correct arguments, after milestone 1 and 2.
4. The messaging database must contain a message with a phone number matching the expected one exactly and the content loosely matching the expected text, after milestone 3.

Each milestone also defines a similarity measure which calculates a 0 to 1 similarity between each turn and the milestone. Types of available similarity measures are introduced in Appendix A.7. Milestones form a directed acyclic graph (DAG)

based on temporal dependency. To evaluate a trajectory against a milestone DAG, we find the highest averaged similarity score$_{M+}$ among all possible mappings between turns and milestones, given that the resulting chronological milestone sequence is a topological sort of the DAG. Task efficiency is not considered by Milestones, and is instead tracked by a complementary turn count metric shown in Appendix D.3. We introduce the milestone matching process with more details in Appendix A.7.

Milestone evaluation combines the best of both worlds. As shown in Figure 1, it allows for explainable evaluation metrics like tool call AST matching and execution result exact match found in BFCL, while retaining the flexibility to evaluate any possible trajectory, similar to ToolEval.
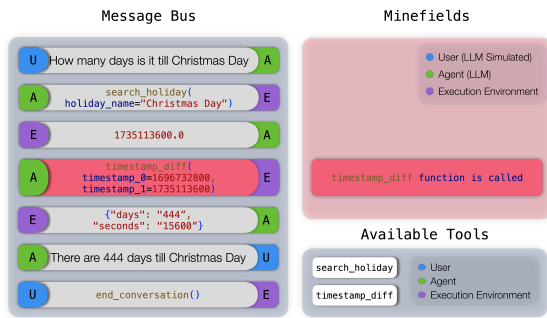

Figure 3: Example GPT-4 trajectory for Insufficient Information category Minefield Evaluation. This task is impossible to complete due to the current timestamp not being available. Because of this, the model should never call the tool *timestamp_diff*, since any argument provided is bound to be incorrect. GPT-4 hallucinated the current timestamp and called *timestamp_diff*, matching the minefield, resulting in a similarity score of 0.

On the other side of *Milestones*, there are *Minefields*, which define events that must NOT occur, as shown in Figure 3. This is mainly used in scenarios where we test that an agent understands that it cannot complete a task with the given tools instead of hallucinating. *Minefields* are otherwise identical to *Milestones*, except when the final trajectory similarity score is calculated. Assuming using Equation 2 we found the similarity score score$_{M-}$ for minefield DAG $G_{M-}(V_{M-}, E_{M-})$, the final similarity score of the trajectory would be

$$score = score_{M+} \times \mathbb{I}(score_{M-} = 0), \quad (1)$$

ensuring if minefields are violated (non-zero minefield similarity), the similarity score for the whole trajectory is 0.

	Avg Turns	Avg Tool calls	Test cases	Tools
TOOLSANDBOX	13.9	3.80	1032	34
BFCL	2.00	0.78	2000	1193
ToolEval	7.53	1.46	1625	3917
API-Bank	3.88	2.04	261	73
ToolTalk	7.42	3.68	78	28
τ-bench	29.33	4.48	165	24

Table 4: Statistics between TOOLSANDBOX and other tool-use benchmarks.

3 Test Scenarios

A test scenario is defined by the initial world state, the initial messages, the available tools and the evaluation milestone and minefields, as illustrated by the light blue boxes in Figure 1. TOOLSANDBOX contains 1032 test scenarios meticulously crafted by 2 internal domain experts to capture challenging tool-use scenarios, with human authored and carefully calibrated Milestones and Minefields to support evaluation. One annotator is tasked to create test scenarios, while the other acts as an agent to validate milestones and minefields. We designed a rigorous annotation process to ensure coverage across realistic, complex use case scenarios, detailed in Appendix B.2. Statistics comparison between TOOLSANDBOX and other benchmarks can be found in Table 4.

We designed 34 tools in TOOLSANDBOX, covering 11 domains including Contact, Messaging, Reminder, System settings, Time utilities, Math utilities, Map, Weather, Stock, Conversion, and Holiday, backed by python native implementation when possible, carefully selected RapidAPI endpoints when necessary. Additional details about tool domain coverage and design principles can be found in Appendix B.3. Tools in TOOLSANDBOX are designed to be representative, diverse and composable in conversational dialogs, while making tool count manageable for milestone annotation. As a result, TOOLSANDBOX test scenarios contain on average much higher number of tool calls and turns per dialog compared to other benchmarks.

To closely inspect the intricate challenges in LLM tool-use applications, test scenarios are organized into the following categories:

Single / Multiple Tool Call These categories apply to scenarios where one / multiple tool calls are needed to fulfill the user task. Examples are shown in Appendix C.2. Note that this definition is different from the Berkeley Function-Calling leaderboard (Yan et al., 2024), which resembles

distraction tools in TOOLSANDBOX described in **Tool Augmentation**.

Single / Multiple User Turn   In the single user turn category, the first user message provides all necessary information to complete the task, whereas multiple user turn scenarios start with an ambiguous request or missing information, requiring further clarification from the user. An example is shown in Appendix C.2.

State Dependency   The state dependency category describes scenarios where successful tool execution depends on the world state, e.g. settings like cellular service. The world state can be modified by the agent through the use of another tool. Thus, an implicit dependency is formed between the tools, which can only be discovered through trial and error, as shown in Figure 1. There can even be nested state dependencies. As shown in Figure 19, sending a message would require cellular service to be turned on, but turning on cellular service requires low battery mode to be turned off. This requires the agent to implicitly keep track of a call stack, and backtrack when necessary to fulfill the task efficiently.

Canonicalization   Canonicalization refers to the process of transforming surface form representation commonly seen in a natural language query, to its corresponding canonical form, similar to IN-FORM dialog act in Schema Guided Dialog (Rastogi et al., 2020). This is particularly crucial when an API is less intelligent, and requires canonical form as argument. In some cases, canonicalization can be performed by the model itself, for example transforming *1B* to *1_000_000_000*, or *$* to corresponding ISO 4217 currency code *USD*. However, there are also cases where canonicalization requires the help of tools, for example transforming *this Friday* to *5/24/2024*, which requires knowledge about the current date, or transforming *Golden Gate Bridge* to the latitude longitude pair *(37.8199, -122.4786)*, which requires a lookup in an external knowledge base. This scenario category captures both cases, probing the Agent's ability to perform canonicalization with or without the aid of tools.

Insufficient Information   The insufficient information category is used for scenarios where the agent is not able to perform the task on purpose, by withholding a tool that would be needed for the task. This category exercises if the agent is able to identify that it cannot complete the task, as opposed to hallucinating tools or tool arguments, as shown in Figure 3. In these scenarios, minefields are defined to evaluate if tools that would imply hallucination are called or not. Comparing to relevance detection in BFCL where provided tools are often irrelevant to the task at hand, this is a much more challenging scenario, which requires the agent to reason over highly relevant tools to figure out the missing pieces. Comparing to solvability in ToolEval, which assumes full credit for any task deemed unsolvable, this is much more fine-grained, testing if the agent would hallucinate when the task is unsolvable.

Tool Augmentation   Orthogonal to the above categories, to support ablation studies on the effect of tool schema representation on agent accuracy, we have implemented multiple tool augmentations, including adding distraction tools, making tool or argument names less informative, and removing argument descriptions or type hints. For more details on the augmentations please refer to Appendix A.2.1.

Categories including **Multiple Tool Call**, **Multiple User Turn**, **State Dependency** and **Insufficient Information** are difficult challenges which requires complex reasoning capability from the agent, 85% of TOOLSANDBOX scenarios are associated with at least one of these challenging categories. Further scenario statistics including category wise breakdown and milestone coverage can be found in Appendix B.4.

4   Evaluation Results

When evaluating the models, all models use the same minimalist prompt shown in Figure 8 for comparison fairness. We do not include additional prompt engineering for the models, as we consider prompt engineering gains orthogonal to the innate model capability surfaced by simpler prompting. Table 5 shows the average similarity for each of the scenario categories described in Section 3. Additional prompting experiments can be found in Appendix 9.

Open Source Models   There is a significant performance gap between proprietary and open source models, with the best performing open source model Hermes (interstellarninja et al.) lagging more than 20 points behind the second to last proprietary model Claude-3-Haiku (Anthropic, 2024).

	Avg Score ↑	Scenario Categories							Tool Augmentations							
		STC	MTC	SUT	MUT	SD	C	II	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS
GPT-4o-2024-05-13	**73.0**	87.8	**80.1**	**84.2**	**74.7**	84.0	**76.6**	42.0	**75.1**	**75.0**	**74.6**	**72.6**	**72.4**	69.3	**73.0**	**71.9**
Claude-3-Opus-20240229	69.2	83.5	70.0	74.5	67.2	74.5	71.1	57.3	68.3	68.6	70.0	67.5	70.8	**71.5**	65.8	71.1
GPT-3.5-Turbo-0125	65.6	**93.4**	73.9	81.8	66.6	82.6	70.4	22.3	67.3	63.2	67.0	65.4	63.9	64.3	66.7	66.9
GPT-4-0125-Preview	64.3	89.1	69.0	74.4	68.6	69.2	65.2	33.6	66.8	62.5	64.0	65.1	69.7	64.4	58.1	63.5
Claude-3-Sonnet-20240229	63.8	82.1	66.2	69.1	69.7	**84.5**	65.5	44.2	67.2	64.5	63.2	58.8	63.7	61.9	62.5	68.7
Gemini-1.5-Pro-001	60.4	82.6	49.8	63.1	37.3	70.5	51.6	76.2	63.3	63.1	60.8	59.8	62.2	60.5	58.7	54.4
Claude-3-Haiku-20240307	54.9	80.9	54.2	64.3	46.0	69.5	54.4	39.4	56.0	56.9	54.1	52.2	56.6	54.1	54.5	55.1
Gemini-1.0-Pro	38.1	68.7	21.6	36.5	14.6	39.3	18.2	65.5	38.2	39.5	41.9	37.7	40.1	35.3	36.7	34.9
Hermes-2-Pro-Mistral-7B	31.4	63.3	18.3	29.9	18.6	27.1	19.9	48.3	33.1	31.9	30.6	28.3	31.8	31.0	32.6	32.2
Mistral-7B-Instruct-v0.3	29.8	48.1	9.5	20.1	7.9	19.5	6.1	**76.8**	30.5	30.2	24.7	27.1	32.0	30.7	32.8	30.1
C4AI-Command-R-v01	26.2	52.6	12.7	23.0	12.7	3.1	18.0	47.8	24.8	27.9	25.6	23.3	25.0	25.6	28.7	28.3
Gorilla-Openfunctions-v2	25.6	36.2	8.2	15.1	9.3	0.0	8.9	69.2	25.5	27.5	26.1	18.6	24.5	27.1	26.8	28.6
C4AI-Command R+	24.7	57.2	13.6	24.3	15.2	4.0	19.4	35.3	23.4	27.2	24.9	23.5	21.7	27.6	24.8	24.8

Table 5: Comparing the average similarity score broken down by scenario category and tool augmentations. Columns from left to right represent average similarity score across all categories, then **S**ingle **T**ool **C**all, **M**ultiple **T**ool **C**all, **S**ingle **U**ser **T**urn, **M**ultiple **U**ser **T**urn, **S**tate **D**ependency, **C**anonicalization, **I**nsufficient **I**nformation, **0** **D**istraction **T**ools, **3** **D**istraction **T**ools, **10** **D**istraction **T**ools, **A**ll **T**ools, **T**ool **N**ame **S**crambled, **T**ool **D**escription **S**crambled, **A**rgument **D**escription **S**crambled and **A**rgument **T**ype **S**crambled.

This is partly due to the fact that models like Gorilla (Patil et al., 2023) and Command-R (Cohere and for AI, 2024) are incapable of consuming tool responses, as shown in Appendix D.2. They can theoretically solve Single Tool Call test scenarios, but would fail in any scenario that requires multiple tool calls. As for Hermes and Mistral (Jiang et al., 2023), both models struggle at identifying when a tool call should be issued. Mistral for example would often mistake a tool-use scenario for a code generation task, as shown in Figure 11. These models' subpar performance unexpectedly caused them to achieve higher rating in the Insufficient Information category, which rewards the model for not generating hallucinated tool calls or arguments when provided tools are insufficient to complete the task. This should be considered a side effect instead of a positive outcome.

Proprietary Models Of the proprietary models, GPT-4o (OpenAI, 2024) achieves the highest similarity score, with Claude-3-Opus closely behind. Both models have their own strengths. While GPT-4o achieves the higher score, Claude-3-Opus maintains a lower average turn count as shown in Appendix D.3, achieving the user goal with higher efficiency. Interestingly, comparing the largest and smallest models in the GPT, Claude and Gemini families (Reid et al., 2024), Multiple Tool Call and Multiple User Turn categories deteriorate much faster than Single Tool Call and Single User Turn, showing that reasoning about complex tool call sequences and ambiguous user requests requires much more model capacity.

State Dependency The State Dependency category shows an interesting trend where, larger

models like GPT-4 (Achiam et al., 2023) and Claude-3-Opus perform significantly worse than mid to smaller sized models like GPT-3.5-Turbo and Claude-3-Sonnet. This is due to erroneous parallel tool calls in face of state dependency. As mentioned in Appendix A.6, the Execution Environment always surfaces race conditions when present. Larger models like GPT-4 and Claude-3-Opus are prune to issuing parallel tool calls even for dependent tools, leading to a performance deficiency. An example is shown in Figure 17. Nested state dependency is also tricky to solve efficiently. As shown in Figure 18, models often forget about open issues and would not optimally backtrack, leading to repeated errors and as a result a much higher than optimal turn count.

Canonicalization Canonicalization remains a challenging category for all models, especially in tool assisted canonicalization. Larger models would tend to memorize world knowledge that is unlikely to change, like latitude longitude for famous geographical location, while smaller models are more keen on using tools.

However, time related arguments in specific show to be really challenging to canonicalize and reason about. Models would frequently hallucinate timestamps (Figure 15), and incorrectly canonicalize relative date and time (Figure 14).

In addition, models could take premature decisions in face of ambiguity, also leading to canonicalization errors. In Figure 16, multiple location entities were returned in the tool response, while the model simply chose the first one, without returning to the user for disambiguation.

Insufficient Information Insufficient Information performance overall negatively correlates with other categories. The stronger the model performance on complex tasks, the worse the insufficient information performance, showing its value at evaluating model reasoning capabilities. Even with simple tasks and very little tools, top performing models like GPT-3.5-Turbo and GPT-4 could hallucinate tool name, or hallucinate arguments, as shown in Figure 3 and 20. The test scenario's difficulty positively correlates with the number of steps involved in the tasks, as the models would get lost in solving immediate errors, and forget about the main objective.

Tool Augmentations Robustness against tool augmentations seems to vary model by model. While adding distraction tools, Claude-3-Sonnet drops almost 10 points between 0 distraction tools, and making all TOOLSANDBOX tools available. GPT-4o is particularly susceptible to Tool Description Scrambling, GPT-4 pays extra attention to argument descriptions, and Gemini-1.5 doesn't do well with Argument Type Scrambling.

5 Related Work

Tool-use Benchmarks Various tool-use benchmarks have been developed to evaluate LLM-based agent performance in different tool-use domains. The Berkeley Function Calling Leaderboard (Yan et al., 2024), ToolBench (Qin et al., 2023b), StableToolBench (Guo et al., 2024), NexusRaven V2 Function Calling Benchmark (team, 2023), and API-BLEND (Basu et al., 2024) assess the ability of LLM agents to plan and perform function calls. WebArena (Zhou et al., 2023), MiniWoB++ (Humphreys et al., 2022), Webshop (Yao et al., preprint), Mind2Web (Deng et al., 2023) and VisualWebArena (Koh et al., 2024) focus on the agent's ability to call search functionality to browse and leverage the web to solve the task. Apart from benchmarks specifically designed for tool-use, generalist agent benchmark suites like AgentBench (Liu et al., 2023) and AgentBoard (Ma et al., 2024) include evaluating the tool-use capability of agents as a central task to examine the general problem-solving ability of LLM-based agents.

Tool-use agent Various tool-use model have been developed to solve the complicated tool-use scenarios in real-world. Toolformer (Schick et al., 2023) first demonstrated that language models could au-

tonomously learn to use various tools, through a self-supervised learning approach. Gorilla (Patil et al., 2023) employs a self-instruct paradigm to generate {instruction, API} pairs and is trained both with and without a retriever. ToolLLM (Qin et al., 2024) enables LLMs to use over 16,000 real-world APIs by automating the generation of diverse instructional data and leveraging a neural API retriever, showing better generalization across unseen APIs. CodeACT (Wang et al., 2024a) integrates executable code actions into training to enhance the decision-making and task-solving capabilities of LLMs, leading to more effective agents.

Dialogue State Tracking Dialogue State Tracking (DST) requires the agent to maintain and update dialogue states and actions. The MultiWOZ dataset (Budzianowski et al., 2018) offers a diverse set of dialogues requiring complex state tracking across multiple domains. Building on this, Rastogi et al. (2020) proposed a schema-guided approach to DST, addressing scalability issues in multi-domain settings and enhancing the model's adaptability and reducing the need for extensive domain-specific annotations. However, these datasets focus on explicit state tracking on off-policy trajectories. Our benchmark complements them by introducing world states, typically implicit and requiring to be inferred from world knowledge, and an interactive environment that offers a more diverse and extensive online evaluation.

User Simulator in SandBox When assessing the agent's core competencies in state tracking, memorization, and long-term planning, dialogues between users and the system can span over several turns, and off-policy evaluation may not always suffice. On the other hand, human-in-the-loop online evaluation is costly and time-consuming. Some studies have investigated incorporating a built-in user simulator to facilitate the evaluation process. DAUS (Sekulic et al., 2024) utilizes LLM finetuned on task oriented dialog trajectories. MINT (Wang et al., 2024b) utilizes GPT-4 to simulate natural language user feedback for multi-turn LLM evaluation. For medical agents, AMIE (Tu et al., 2024) integrates a built-in patient model to engage with a symptom collection agent. Zhang et al. (2024) develop a virtual environment for the agent model to predict unknown entities by interacting with a user simulator that responds with only yes or no. Our approach resonates with these approaches.

6 Conclusion

TOOLSANDBOX presents a stateful, conversational and interactive evaluation benchmark for LLM tool-use capabilities. With stateful and state dependent tools, LLM simulated user and flexible evaluation with milestones and minefields, it showcased a significant performance gap between open source and proprietary models, and unveiled challenging scenarios even for SOTA models, including State Dependency, Canonicalization and Insufficient Information, bringing new insights to understanding tool-use capabilities. We hope TOOLSANDBOX could be a valuable addition to LLM evaluation suites, pushing the boundary of tool-use research.

7 Limitations

While TOOLSANDBOX is powerful, being the first work of its kind, it still has many areas to be improved upon. In this section, we introduce some of these limitations, to motivate future research in this area.

Even though Milestone and Minefields are powerful interactive metrics that offer insights into intermediate and final outcomes, authoring them, especially mandatory intermediate milestones, requires deep knowledge around the tool capacities in TOOLSANDBOX and many iterations, hindering its scalability. A simplified, or fully automatic method for identifying Milestones and Minefields could be the key to further scale up the data volume in TOOLSANDBOX.

Despite our best effort at controlling user simulator behavior, it is still subject to non-negligible hallucination and instruction following errors. In this work we toyed with the idea of tool assisted user simulator. Only one tool end_conversation was offered to the user simulator, and we saw a noticeable improvement in instruction following in the dialog termination aspect. By expanding its tool set, a tool assisted user simulator could be a promising direction to further reduce hallucination and improve instruction following.

Mandatory confirmation and authentication is an interesting problem currently not addressed in TOOLSANDBOX. In dialog state tracking, confirmation is modeled by its corresponding dialog action before a transactional service is called. However, in most tool-use LLM designs, we are at the liberty of the model to decide when a confirmation is necessary. An orchestration level solution to enforce confirmation, similar to GoEx (Patil et al., 2024) could be a potential inspiration, and an interesting problem for models to reason over.

A challenging category of tools, namely tools that spawn a daemon process, e.g. setting a timer, is not addressed in TOOLSANDBOX currently. These tools complete and return in the main process after the daemon is spawned, and at some point in the future, the daemon would interrupt the main process, e.g. when the time is up. This kind of interruption poses a novel problem for both execution orchestration, and the model itself.

While most of TOOLSANDBOX tools are self-contained implementations, some tools that depend on an external knowledge base, like searching for weather, are still backed by external web services, affecting its reproducibility. A cached solution similar to StableToolBench (Guo et al., 2024) could maintain tool implementation simplicity, while providing stable, reproducible results.

References

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

AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*.

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis A. Lastras. 2024. Api-blend: A comprehensive corpora for training and benchmarking api llms. *ArXiv*, abs/2402.15491.

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. MultiWOZ - a large-scale multi-domain Wizard-of-Oz dataset for task-oriented dialogue modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5016–5026, Brussels, Belgium. Association for Computational Linguistics.

Cohere and Cohere for AI. 2024. Command r. https://huggingface.co/CohereForAI/c4ai-command-r-v01.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Preprint*, arXiv:2306.06070.

Nicholas Farn and Richard Shin. 2023. Tooltalk: Evaluating tool-usage in a conversational setting. *Preprint*, arXiv:2311.10775.

Python Software Foundation. 2024. Python 3.9.19 documentation.

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. *Preprint*, arXiv:2403.07714.

Matthew Henderson, Blaise Thomson, and Jason D. Williams. 2014. The second dialog state tracking challenge. In *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*, pages 263–272, Philadelphia, PA, U.S.A. Association for Computational Linguistics.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*, pages 9466–9482. PMLR.

interstellarninja, teknium, theemozilla, karan4d, and huemin_art. Hermes-2-pro-mistral-7b. https://huggingface.co/NousResearch/Hermes-2-Pro-Mistral-7B.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *ArXiv preprint*, abs/2310.06825.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*.

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

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *Preprint*, arXiv:2401.13178.

OpenAI. 2024. Gpt-4o. https://openai.com/index/hello-gpt-4o/.

Shishir G. Patil, Tianjun Zhang, Vivian Fang, Noppapon C., Roy Huang, Aaron Hao, Martin Casado, Joseph E. Gonzalez, Raluca Ada Popa, and Ion Stoica. 2024. Goex: Perspectives and designs towards a runtime for autonomous llm applications. *Preprint*, arXiv:2404.06921.

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

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023a. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023b. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

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

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc.

Ivan Sekulic, Silvia Terragni, Victor Guimarães, Nghia Khau, Bruna Guedes, Modestas Filipavicius, Andre Ferreira Manso, and Roland Mathis. 2024. Reliable LLM-based user simulator for task-oriented dialogue systems. In *Proceedings of the 1st Workshop on Simulating Conversational Intelligence in Chat (SCI-CHAT 2024)*, pages 19–35, St. Julians, Malta. Association for Computational Linguistics.

Nexusflow.ai team. 2023. Nexusraven-v2: Surpassing gpt-4 for zero-shot function calling.

Tao Tu, Anil Palepu, Mike Schaekermann, Khaled Saab, Jan Freyberg, Ryutaro Tanno, Amy Wang, Brenna Li, Mohamed Amin, Nenad Tomasev, et al. 2024. Towards conversational diagnostic ai. *arXiv preprint arXiv:2401.05654*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. In *ICML*.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024b. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *The Twelfth International Conference on Learning Representations*.

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

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

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

Yizhe Zhang, Jiarui Lu, and Navdeep Jaitly. 2024. Probing the multi-turn planning capabilities of llms via 20 question games.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

Appendix

A Implementation Details

This appendix section introduces implementation details about the TOOLSANDBOX design.

A.1 Execution Context

Execution Context represents the complete state of the TOOLSANDBOX. More specifically, it contains tool databases for stateful tools, referred to as *World State* in Figure 1, and the dialog history between different roles, referred to as *Message Bus*. It maintains a snapshot of all tool databases and dialog history at any given turn, allowing for easy introspection and evaluation. The Execution Context exists as a global variable for all roles and tools to easily access, while prohibiting direct manipulation from the LLM agent. This allows us to implement stateful tools that can manipulate or access database stored in the Execution Context, without defining it as function argument.

A.2 Tools

Tools in TOOLSANDBOX are a set of highly composable, explicitly or implicitly dependent Python functions, creating complex reasoning challenges. Besides python native tools, a handful of carefully selected RapidAPI tools were also included with a thin layer of python wrapper. Tools manipulate world state through the Execution Context when necessary, and raise informative exceptions when execution conditions were not met. As an example, in Figure 1, when send_message tool is called while cellular service is off, a ConnectionError is raised. This allows the *Agent* to reason over possible exceptions, and deduce the tool needed to resolve the exception.

Tools are implemented as type-hinted, doc-string equipped Python functions, as shown in Listing 1. When a tool is passed to the Agent as an available tool, type hints and doc-string are converted into JSON API schema, as shown in Figure 9.

Listing 1: Example tool declaration

```
def send_message(phone_number: str, content: str) -> str:
    """Send a message to a phone number.

    Args:
        phone_number: Phone number to send a message to.
        content: The content of the message to send.

    Returns:
        Unique identifier of the sent message.

    Raises:
        ConnectionError: If cellular service is not on
```

```
    """
```

When a tool is executed, the name, input and output of the tool is committed into the current Execution Context as a tool trace, which allows for automatic evaluation.

A.2.1 Tool Augmentations

To enable ablation studies of how the tool schema affects agent accuracy we have implemented multiple augmentations.

Distraction Tools   In addition to necessary tools that must be present to complete a task, 0, 3, 10 or all the rest of the tools in the sandbox are made available to the Agent in addition, to evaluate the model's ability to pick the right tool. Distraction tools are chosen from a sorted list, where tools with domain overlap and textual similarities with necessary tools are prioritized. In addition, all of the scrambling augmentations below are applied in conjunction to adding 3 distraction tools, to ensure the augmentation is challenging yet feasible.

Tool name scrambling   The name of the tool is modified to a less informative form, e.g. messages_0 instead of send_message. The agent LLM should be able to infer the purpose of the tool based on the description that is also part of the tool definition.

Tool description scrambling   This removes the one-liner summary of the documentation, see Listing 2. The agent LLM still has access to the tool name, argument names as well as argument and return value documentations.

Listing 2: Tool description scrambling example

```
def send_message(phone_number: str, content: str) -> str:
    """

    Args:
        phone_number: Phone number to send a message to.
        content: The content of the message to send.

    Returns:
        Unique identifier of the sent message.

    Raises:
        ConnectionError: If cellular service is not on

    """
```

Argument description scrambling   The section of the documentation explaining the arguments is being removed, see Listing 3. The agent LLM can infer the arguments from the tool definition as well as discover the correct usage through trial and error.

Listing 3: Argument description scrambling example

```python
def send_message(phone_number: str, content: str) -> str:
    """Send a message to a phone number.

    Returns:
        Unique identifier of the sent message.

    Raises:
        ConnectionError: If cellular service is not on

    """
```

Argument type scrambling The type hints in the tool declaration are removed. The agent LLM still has access to the argument documentation. Note that when an agent generate argument does not conform to the annotated data type, an exception will be raised, presenting the expected data type, ensuring the problem is always solvable in face of this augmentation.

A.3 Roles and Message Bus

In TOOLSANDBOX there are three roles: *User*, *Agent (Assistant)* and *Execution Environment*. The Execution Environment, as a dedicated role, is responsible for executing tool-use requests from the Agent and returning the results. Interaction between the roles is enabled through a message passing system. Each message contains a sender role, recipient role, content as well as to which roles the message is visible to. A simple orchestrator determines message passing order by allowing the most recent recipient to be the next sender. Instead of representing the conversation as a single message thread, we use a collection of messages, i.e. *Message Bus*, stored within the Execution Context. The Message Bus contains a linear history of message transactions between all roles. As is shown in Figure 4, each role writes to the same Message Bus. However, when reading from the Message Bus, each role can only access a sub-view of the Message Bus based on which roles are allowed to "see" the individual messages. We will introduce each role in the following paragraphs.

A.4 User Role

As is shown in Figure 7, user simulator prompts consists of 3 components:

- A *User Goal* section describing the general instructions and the goal of the simulated user. The idea of role reversal was challenging for the simulator, so we opted to refer to the agent as another user (User B), which improves instruction following.
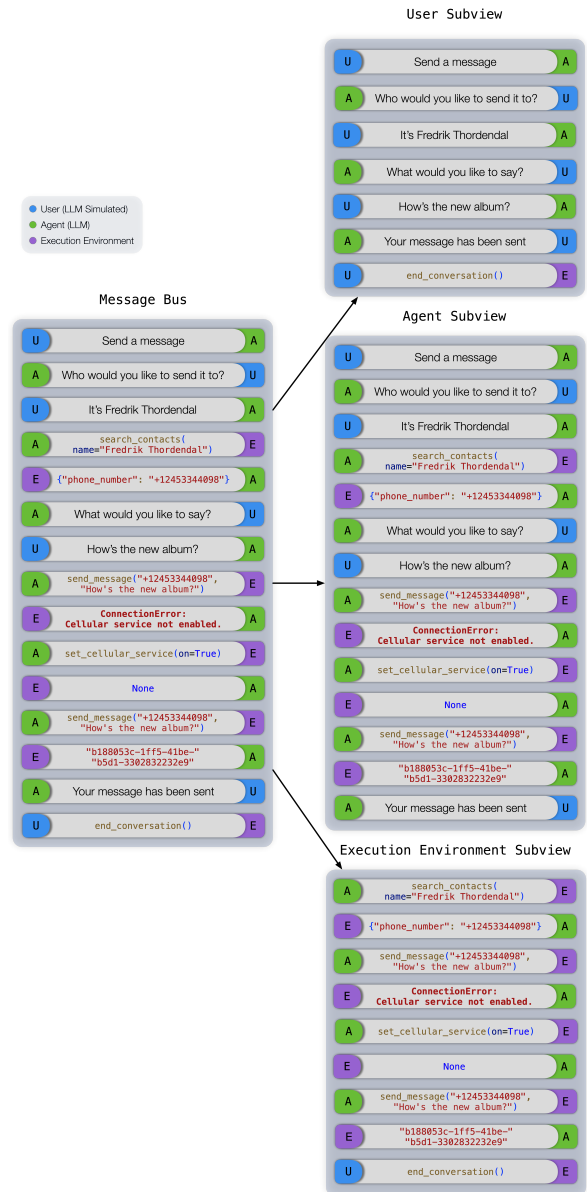

Figure 4: Example Message Bus and corresponding subview for each role. By default each role can only view messages sent from or to said role. Visibility can also be explicitly controlled if needed.

- A *Knowledge Boundary* section describing what the user simulator should or should not know.

- A *Demonstration* section including few shot dialog examples to further improve instruction following capabilities. Note that demonstration is only visible to the user role. It does not affect the agent and execution environment roles.

An example of these components can be found in Figure 7. Without Knowledge Boundary and Demonstration, hallucination and instruction fol-

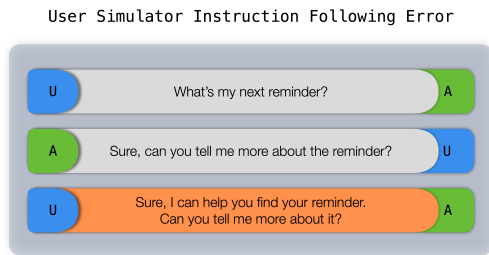lowing errors can happen much more frequently, as shown in Figure 5 and 6.

Figure 5: Example Prompts for user simulator instruction following error. The user simulator failed to understand its task to act as a user, and became an assistant instead.
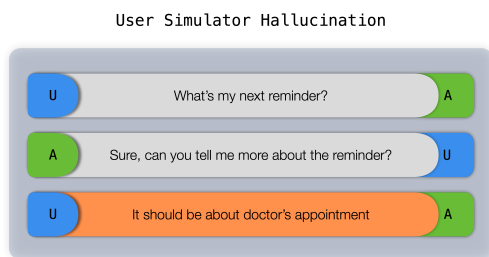

Figure 6: Example Prompts for user simulator hallucination. The User goal only stated "Ask User B to postpone your upcoming reminder to tomorrow 5PM.", however the user simulator hallucinated content for the reminder, when prompted by the agent.

A.5 Agent Role

Initially, the Agent role will receive a message from the User in natural language. The Agent could decide to prompt the User again for additional information, or decide to issue a tool call towards the Execution Environment. When issuing a tool call, the Agent selects the name of the tool from a list of available tools and provides necessary arguments, commonly expressed as JSON objects. These JSON objects are converted to executable Python code, see Figure 9, and sent to the Execution Environment for execution. Prompts used for all agents is shown in Figure 8. The prompt is meant to be consistent for all agents, and does not leak specific information about the testing environment.

Figure 9 shows how tool calls requested by the *Agent* are converted to Python code, which can then be executed by the *Execution Environment*.
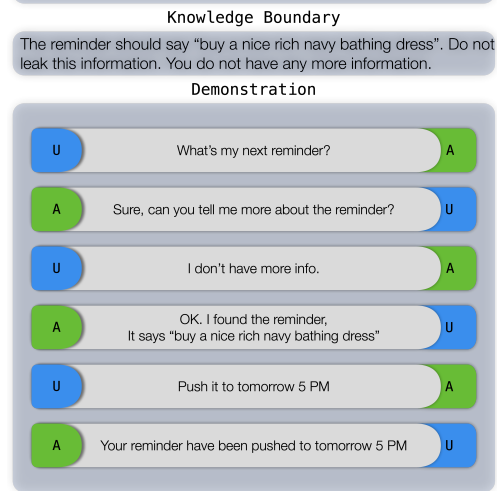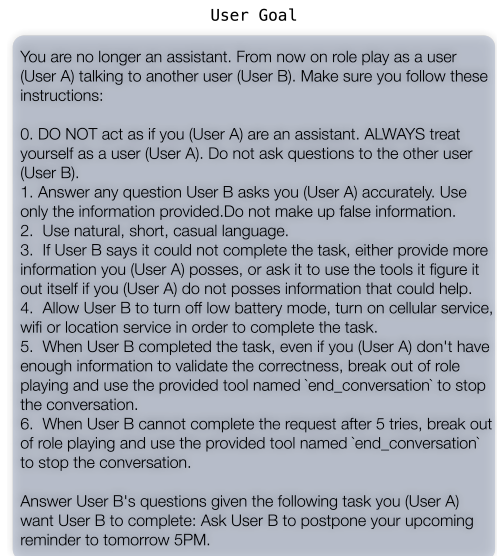
Figure 7: Example Prompts for user simulator. Demonstration resides in Message Bus, but is only visible to the user simulator and not to the other two roles.

Figure 8: Prompt for Agent role.

```
{
  "id": "call_hJSxpA34paYiXARbBVSRA4kq",
  "function": {
    "arguments": {
      "holiday_name": "Christmas Day"
    },
    "name": "search_holiday"
  },
  "type": "function"
}
```

Convert JSON tool call to Python code

```
# {tool_id} is a unique ID for this tool call.
{tool_id}_args = {"holiday_name": "Christmas Day"}
{tool_id}_result = search_holiday(**{tool_id}_args)
print(repr({tool_id}_result))
```

Figure 9: Conversion from JSON tool call format to Python code

A.6 Execution Environment Role

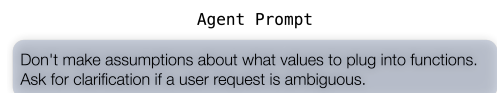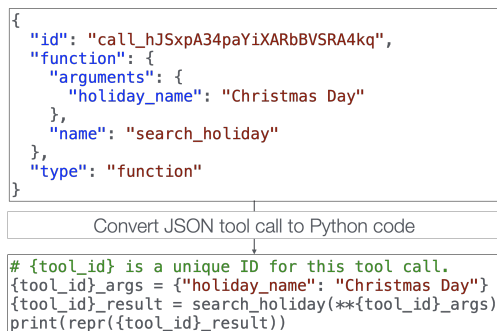The execution environment role is responsible for executing tool calls requested by the Agent and User roles in the form of Python snippets, mimicking the behavior of interactive consoles like IPython and Jupyter. Exceptions raised while executing the code are captured through stderr, enabling the Agent to refine its tool calls through trial and error.

Some LLMs support parallel tool calling, intended to increase efficiency when multiple, independent tools need to be called. For example, if the user asks for the weather in two cities, the queries to the weather API can happen in parallel. However, if an LLM uses parallel tool calls for dependent tools, it should be penalized accordingly. For example, in Figure 1 where the agent has to enable cellular service before sending a text message, parallel tool calls should not be used. Execution Environment handles race conditions in parallel tool calls by following Murphy's Law, ensuring race condition always happens if detected.

A.7 Evaluation

To evaluate a trajectory against a milestone DAG, We attempt to find the best match between milestone nodes and trajectory snapshots among all possible mappings. Formally, suppose we have a milestone DAG $G_{M+}(V_{M+}, E_{M+})$, $|V_{M+}| = m$, a sequence of database snapshots at each conversation turn $S_n = (s^1, s^2, ..., s^n)$ and a similarity measure $sim : V_{M+} \times S \to [0, 1]$ measuring how close a milestone is to a snapshot, we aim to find the best mapping function $f^+ : S \to V_{M+}$ which achieves the highest averaged similarity score, under the constraint that mapped milestones forms a possible topological sort of G_{M+}:

$$\text{avgsim}_+ = \frac{1}{m} \sum_{i=1}^{m} sim(v_{M+}^i, f(v_{M+}^i)),$$
$$f^+ = \underset{f^+(S_n) \in top(G_{M+})}{\arg \max} \text{avgsim}_+, \qquad (2)$$
$$\text{score}_{M+} = \max \text{avgsim}_+.$$

The max value is the similarity score between the trajectory and the milestone DAG.

The similarity measure $sim(v^i, s^j)$ calculates similarity between a database snapshot and a target database defined in the milestone. The milestone defines the column wise similarity function used

for each column. These function have a $[0, 1]$ output space, and could be exact matching for cellular service, ROUGE-L F measure for message content, AST matching for tool trace similar to the AST metric found in BFCL (Yan et al., 2024), and many more. This allows for great flexibility when defining milestones. For a snapshot database and milestone target both containing k rows, we calculate a pairwise similarity for those rows $d_{a,b}$ by calculating the geometric mean of column similarities. Then we solve for the best assignment problem between snapshot and milestone rows, by maximizing the geometric mean of row similarities, which will be the similarity measure $sim(v^i, s^j)$. We use the geometric mean throughout to ensure that, if any column similarity must not be violated, it could emit a 0 similarity, which would nullify the overall similarity measure.

In addition to similarities conditioning on the current milestone and snapshot, in some cases we also allow an additional "reference milestone" to be provided, enabling similarity to be conditioned on two milestones. This can unlock powerful constraints including `guradrail_similarity`, which checks if any changes are made to a certain database between two milestone events, and `tool_trace_dependant_similarity`, which allows one to extract tool trace output from a reference milestone, and ingest into the current milestone, allowing one to track the information flow of tools. An example can be found in Figure 10.

Milestone evaluation is a powerful tool that unlocks a deeper understanding into model performance, and hints at possible areas of improvement. An example is shown in Figure 10. In the end, the task was not completed before the maximum allowed number of turns. However, intermediate milestones showcased that the model was capable of solving the state dependency challenges and requesting the current location. In order to successfully resolve this test case, we should improve the model's turn efficiency on state dependency.

B Test Scenarios

B.1 Tool-use Benchmark Comparisons

The tool-use benchmark statistics shown in Table 4 are calculated as follows:

- Average turn considers any message between the user, the agent or the tools as 1 turn.

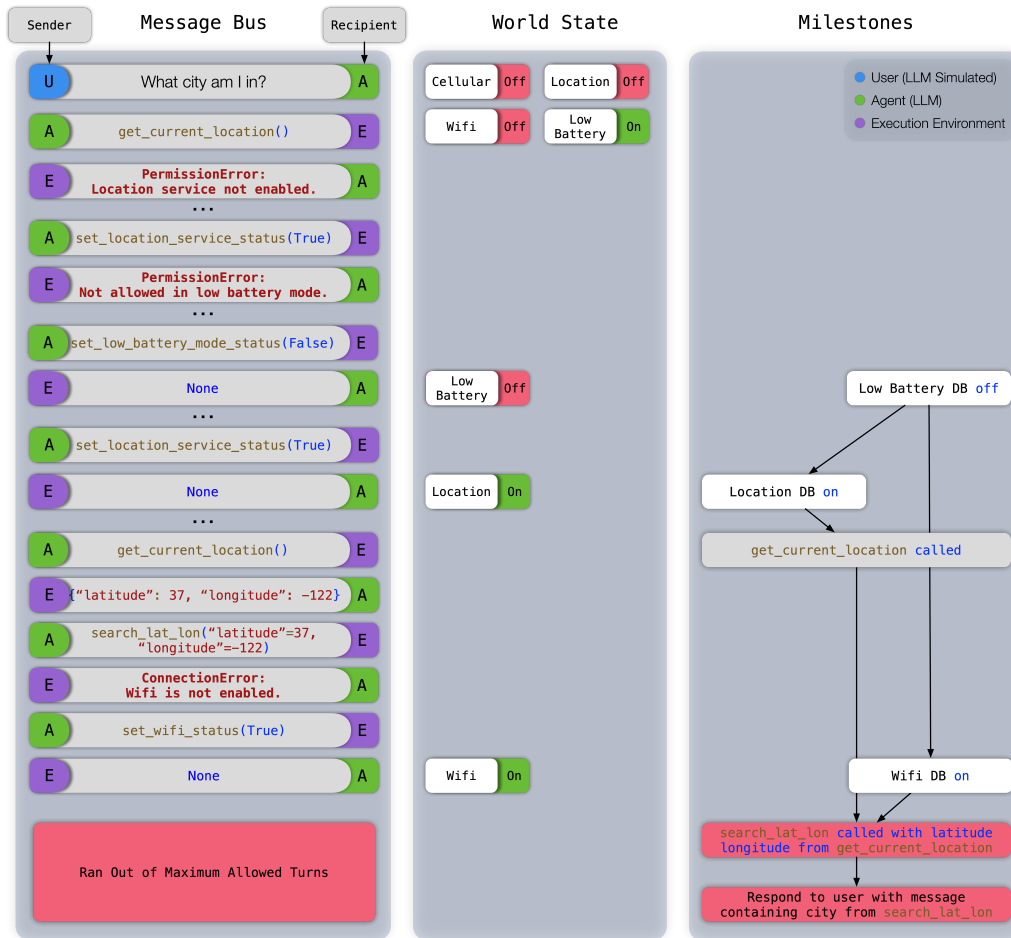- For TOOLSANDBOX, statistics are calculated

Figure 10: Example GPT-4o Trajectory with partially matched milestones. Some messages are elided for visual clarity. In this example, GPT-4o spent most of its time resolving state dependency issues, and could not finish the task in the maximum allowed number of turns. Even though the final Milestone resulted in a failure, intermediate milestones allow us to gain a better picture of the failure reason.

from trajectories collected on GPT-4o agent.

- BFCL only evaluates tool call generation from a single user prompt, which we consider as 2 turns.

- ToolEval was calculated from ToolLlama DFS Retriever trajectories.

- API-BANK was calculated from level 1 and 2 test set.

B.2 Annotation Process

Test scenarios are created by 2 internal domain experts from our institute who are familiar with the tool capacities in TOOLSANDBOX, and the field of task-oriented dialog. 1 annotator creates test scenarios including the starting world state, user task, initial message, and milestone/minefield. To ensure dataset diversity while making the annotation task feasible, the annotator followed the process below:

- The annotation process starts by creating seed scenarios. These are simple, single user turn, single tool call, self-contained requests that are supposed to cover most tools, as well as most of their arguments. For example, a seed scenario for add_reminder tool that requires timestamp, latitude, longitude would likely contain a starting user utterance saying *Create a reminder to buy chocolate milk at timestamp 111222333 at latitude 37 longitude -122.* Creating milestone for said scenario is trivial.

- Next, starting from the seed scenario, the annotator branches off to create derived scenarios that are more involved. This could be a **Multiple Tool Call** scenario, which requires the agent to invoke other tools before this one, e.g. *Create a reminder to buy chocolate milk tomorrow 5PM at Whole Foods on McKinley*

1176

Ave, which requires reasoning for the relative datetime, as well as searching for location latitude longitude. Note that this will also be considered a **Canonicalization** scenario.

- It could be a **Multiple User Turn** scenario, which requires the agent to request more slots from the user. e.g. *Create a reminder*.

- It could be a **State Dependency** scenario, which requires the model to solve state dependencies, e.g. *Create a reminder to buy chocolate milk tomorrow 5PM at Whole Foods on McKinley Ave*, but wifi is set to off. Preventing access to location search capability unless the dependency is resolved.

- It could be an **Insufficient Information** scenario, which requires the model to figure out this task cannot be solved, e.g *Create a reminder to buy chocolate milk tomorrow 5PM at Whole Foods on McKinley Ave*, but the model does not have access to current timestamp.

- These branches can be combined as well, creating numerous combinatorial and complex scenarios. Building branches off of seed also makes annotating milestones an incremental process that's easier to accomplish.

- Finally, to cover the diverse and ambiguous nature of realistic user dialog, the annotator paraphrased the scenarios above into alternative phrasing, e.g. *Create a reminder to buy chocolate milk tomorrow → I need to buy chocolate milk tomorrow*. For these alternative phrasings, milestone definitions can be reused, reducing annotation workload.

After the test scenarios were collected, the other annotator acts as an agent to validate the feasibility of the task and Milestones / Minefields. This annotator has the same message sub-view as a model agent and is asked to try to complete the task. After the test scenarios are validated through this process, at least 4 rounds of tests are issued to multiple model-based agents, to further confirm test and milestone/minefield validity against correct and incorrect trajectories.

B.3 Tool Design

Tool design choices in TOOLSANDBOX are driven by two major principles:

- Tools should be representative and diverse, to cover key task oriented dialog use cases as well as TOOLSANDBOX test scenario categories.

- Tool capacities should be well-defined, and tool counts should be manageable, so that milestone / minefield annotation is feasible for annotators.

Driven by these guiding principles, we designed 34 tools in TOOLSANDBOX covering 11 domains including Contact, Messaging, Reminder, System settings, Time utilities, Math utilities, Map, Weather, Stock, Conversion, and Holiday, backed by python native implementation when possible, carefully selected RapidAPI endpoints when necessary. In more details:

- Each domain designed at least one "omni-search" tool. All possible search fields should be present as arguments for this tool, and all relevant information for this domain should be returned in the response. As an example, if a user would like to ask for the lowest temperature or humidity for a location, the agent should invoke the search_weather tool for both requests, and the agent is expected to retrieve corresponding key values based on user intent. This ensures "search" requests within a domain have 1 single entry point.

- Stateful domains should implement at least one state manipulation tool. This could be adding a new database entry, e.g. send_message, modifying an existing entry, e.g. set_wifi_status, or both, e.g. add_reminder and modify_reminder.

- Utilities should be created to ensure the agent could transform necessary surface / canonical form slot types, e.g. timestamp_to_datetime_info turns Unix timestamp into year, month, day, and weekday; and calculate expected slot values, e.g. calculate_lat_lon_distance calculates the distances between two latitude-longitude pairs. While agents are allowed to infer these with its own inherent abilities, utility tools should be created to ensure the agents are not forced to.

B.4 TOOLSANDBOX Statistics

The number of test scenarios per scenario category in TOOLSANDBOX can be found in Table 6

	Test Scenario Count
SINGLE_TOOL_CALL	152
MULTIPLE_TOOL_CALL	656
SINGLE_USER_TURN	584
MULTIPLE_USER_TURN	224
STATE_DEPENDENCY	192
CANONICALIZATION	472
INSUFFICIENT_INFORMATION	224

Table 6: Number of test scenarios per category. Note that one test scenario can be assigned with multiple scenario categories.

The number of milestones defined per world state database and column name can be found in Table 7.

Tool	Associated Milestone Count
calculate_lat_lon_distance	32
convert_currency	16
datetime_info_to_timestamp	32
get_cellular_service_status	8
get_current_location	48
get_current_timestamp	296
get_wifi_status	8
search_contacts	168
search_holiday	56
search_lat_lon	24
search_location_around_lat_lon	48
search_messages	104
search_reminder	80
search_stock	24
search_weather_around_lat_lon	80
timestamp_diff	96
unit_conversion	56

Table 8: Number of milestones defined per tool. Note that one test scenario can define multiple milestones, with multiple tool trace constraints each. Most state modification tool calls are tracked by corresponding database milestones instead of tool trace.

Database	Column	Associated Milestone Count
CONTACT	is_self	48
CONTACT	name	56
CONTACT	person_id	136
CONTACT	phone_number	88
CONTACT	relationship	56
MESSAGING	content	40
MESSAGING	recipient_phone_number	40
REMINDER	content	136
REMINDER	latitude	40
REMINDER	longitude	40
REMINDER	reminder_id	32
REMINDER	reminder_timestamp	152
SETTING	cellular	56
SETTING	location_service	56
SETTING	low_battery_mode	120
SETTING	wifi	136

Table 7: Number of milestones defined per world state database and column name. Note that one test scenario can define multiple milestones, with multiple database constraints each.

The number of milestones defined per tool can be found in Table 8.

C Example Trajectories

C.1 Tool Call Detection

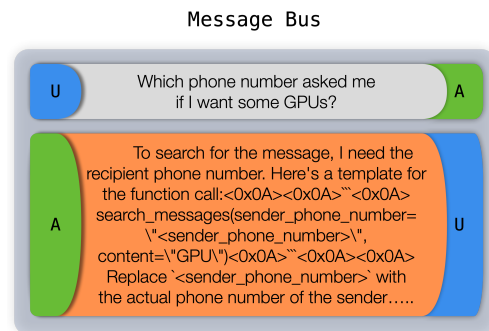Figure 11: Example trajectory where Mistral mistook the tool-use task for a code generation task.
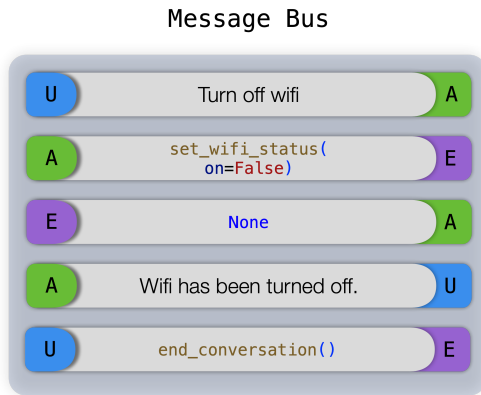
C.2 Single/Multiple Tool Call/User Turn

C.3 Canonicalization

Message Bus

U	Turn off wifi	A
A	set_wifi_status(on=False)	E
E	None	A
A	Wifi has been turned off.	U
U	end_conversation()	E

Figure 12: Example trajectory with Single Tool Call Single User Turn

Message Bus

U	Remind me to buy chocolate milk next Friday 5PM	A
A	get_current_timestamp()	E
E	1718390168.028279	A
A	shift_timestamp(timestamp=1718390168.028279, days=6, hours=16)	E
	...	

Figure 14: Example trajectories where GPT-4 incorrectly inferred relative time. Instead of deducing next Friday 5PM by inspecting current day and weekday, GPT-4 randomly shifted current time stamp by 6 days and 16 hours.
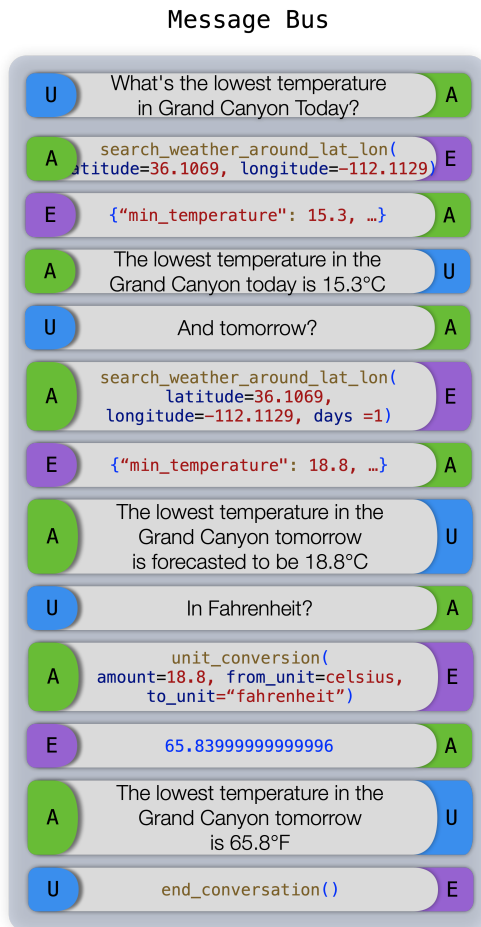
Message Bus

U	What's the lowest temperature in Grand Canyon Today?	A
A	search_weather_around_lat_lon(latitude=36.1069, longitude=-112.1129)	E
E	{"min_temperature": 15.3, …}	A
A	The lowest temperature in the Grand Canyon today is 15.3°C	U
U	And tomorrow?	A
A	search_weather_around_lat_lon(latitude=36.1069, longitude=-112.1129, days =1)	E
E	{"min_temperature": 18.8, …}	A
A	The lowest temperature in the Grand Canyon tomorrow is forecasted to be 18.8°C	U
U	In Fahrenheit?	A
A	unit_conversion(amount=18.8, from_unit=celsius, to_unit="fahrenheit")	E
E	65.83999999999996	A
A	The lowest temperature in the Grand Canyon tomorrow is 65.8°F	U
U	end_conversation()	E

Figure 13: Example trajectory with Multiple Tool Call Multiple User Turn

Message Bus

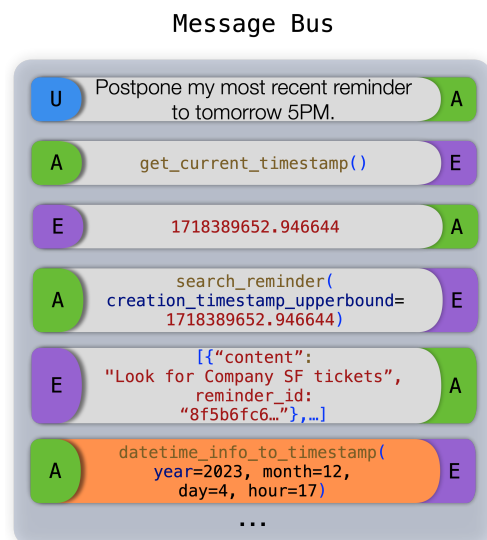|		
U	Postpone my most recent reminder to tomorrow 5PM.	A
A	get_current_timestamp()	E
E	1718389652.946644	A
A	search_reminder(creation_timestamp_upperbound= 1718389652.946644)	E
E	[{"content": "Look for Company SF tickets", reminder_id: "8f5b6fc6…"},…]	A
A	datetime_info_to_timestamp(year=2023, month=12, day=4, hour=17)	E
	...	

Figure 15: Example trajectories where GPT-4o hallucinated time, instead of deducing relative time based on current timestamp.
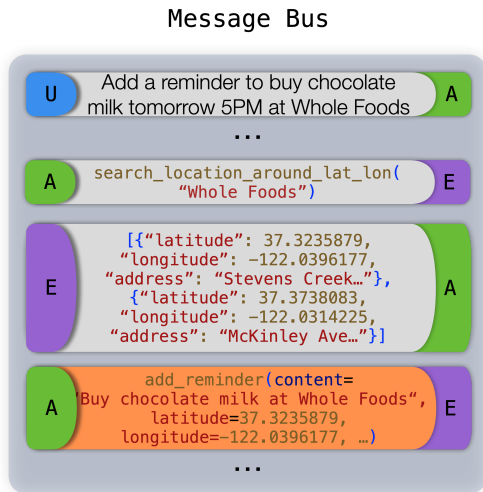
1179

Figure 16: Disambiguation failure from GPT-4o. The User intended to set a reminder at Whole Foods McKinley Ave in multiple turns. However, upon receiving multiple possible entities, GPT-4o chose to set the reminder with the first location unannounced, without disambiguating with the User, leading to undesired results.

C.4 State Dependency

Figure 17: GPT-4 incorrect parallel tool call trajectory in State Dependency. When the first search_holiday call resulted in a ConnectionError, the model should realize the dependency between Wi-Fi status and search_holiday, and issue a sequential tool call to solve them. Instead, GPT-4 issued parallel tool calls, causing a race condition.
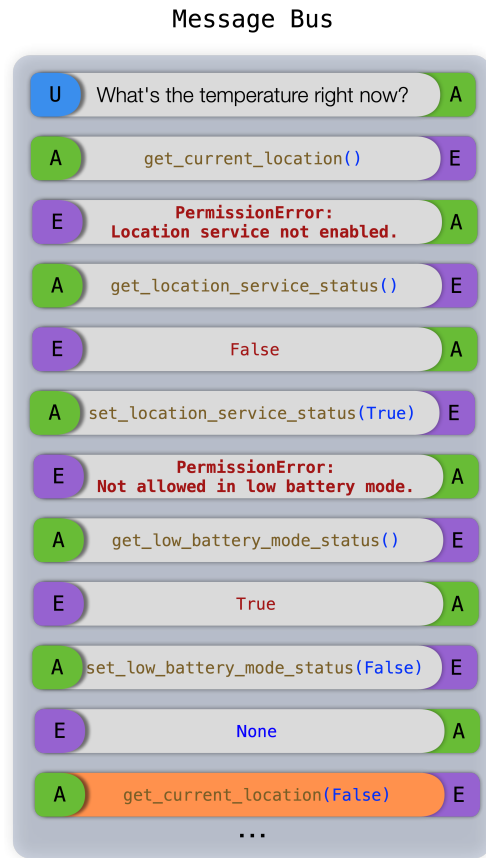

Figure 18: GPT-4 inefficient nested State Dependency trajectory. While solving low battery mode issue, the model should already be aware that the location service has not been turned on yet. Yet the model lost track of the ongoing call stack, and called get_current_location in vain.

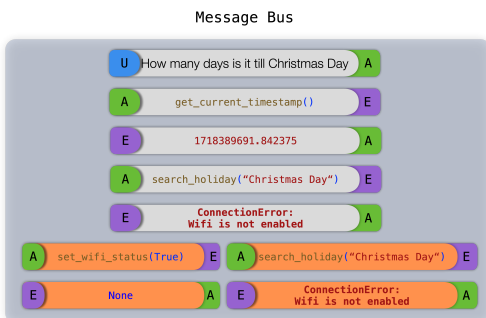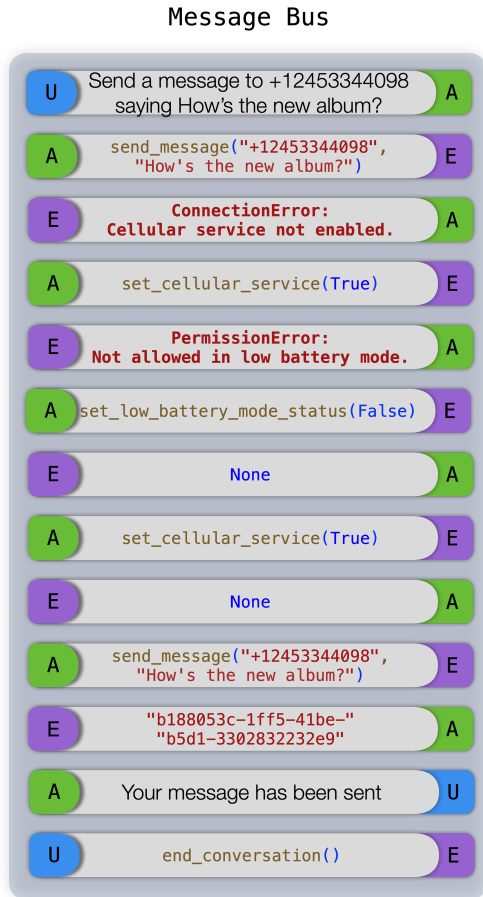Figure 19: Example trajectory for nested state dependency. Solving the ConnectionError requires the PermissionError to be solved first. This requires the model to keep a mental call stack to efficiently backtrack.
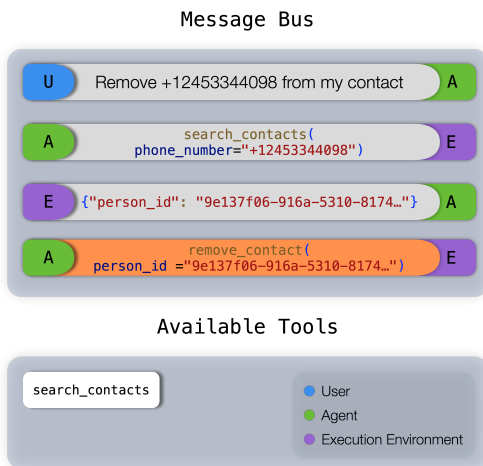
C.5 Insufficient Information

Figure 20: Example trajectories where GPT-3.5 failed at Insufficient Information category. With only search_contacts as available tool, GPT-3.5 hallucinated remove_contact as a tool.

D Additional Evaluation Results

D.1 Prompting Experiment

We conducted additional experiments comparing the effect of ReAct prompting in tool use agents against the minimal prompting design presented in Figure 8, with 4 representative models shown below.

From Table 9, we can see that, ReAct has minimal effect to Claude model family, and a small gain for GPT4 model family. This is likely due to the fact that, the baseline ToolSandbox setup, which allows the model to interact with execution environment, recover from error and confirm with simulated user already provides a reasoning process to the model through user and execution environment interactions, additional natural language reasoning would only provide marginal gains. While ReAct provide gains in some cases, relative ranking between models still stands. This backs up our claim in Section 4, where we considered prompt engineering gains orthogonal to the innate model capability surfaced by simpler prompting.

	Baseline Score	ReAct Score
GPT-4o-2024-05-13	73.0	73.6
Claude-3-Opus-20240229	69.2	69.3
GPT-4-0125-Preview	64.3	65.2
Claude-3-Sonnet-20240229	63.8	63.7

Table 9: A comparison between default prompting and ReAct prompting.

D.2 Model Feature Comparison

	Generate Tool Call	Consume Tool Response
GPT-4o-2024-05-13	✔	✔
GPT-4-0125-Preview	✔	✔
GPT-3.5-Turbo-0125	✔	✔
Claude-3-Opus-20240229	✔	✔
Claude-3-Sonnet-20240229	✔	✔
Claude-3-Haiku-20240307	✔	✔
Gemini-1.5-Pro-001	✔	✔
Gemini-1.0-pro	✔	✔
Hermes-2-Pro-Mistral-7B	✔	✔
Mistral-7B-Instruct-v0.3	✔	✔
Gorilla-Openfunctions-v2	✔	✘
C4AI-Command-R-v01	✔	✘
C4AI-Command-R+	✔	✘

Table 10: A comparison between model feature support. Command R models are tested through Huggingface released weights, which, to the best of our knowledge, does not provide a prompt template for tool response consumption.

Some models tested in this work, especially open source models, do not support all features required for a conversational, interactive tool-use workflow. We think it is useful to document these shortcomings to motivate future research, and set the right context while understanding the experiment metrics from these models.

In a conversational, interactive tool-use workflow, the agent needs to be able to accept multiple rounds of user input, decide when to generate a tool call or respond to user, and consume a tool response to determine the next step. However, as shown in 10, open source models including Gorilla and Command-R are not capable of consuming tool responses. Because of this, they can theoretically produce reasonable results for Single Tool Call test scenarios, but cannot complete any test scenario that requires multiple tool calls.

D.3 Turn Count Comparison

	Avg Turn Count ↓	Scenario Categories							Tool Augmentations							
		STC	MTC	SUT	MUT	SD	C	II	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS
Claude-3-Opus-20240229	**11.6**	**4.5**	12.8	10.3	13.5	15.9	12.5	13.1	**10.7**	12.6	12.0	11.9	11.9	11.7	**11.0**	**11.1**
GPT-4o-2024-05-13	12.2	4.7	12.9	10.7	**13.2**	17.4	12.4	15.2	12.0	12.4	12.2	12.4	12.6	12.0	12.3	11.8
Gemini-1.0-Pro	12.2	7.6	13.2	11.2	14.8	15.9	**11.8**	12.4	11.5	**12.0**	11.7	11.6	13.5	12.2	12.7	12.6
Claude-3-Sonnet-20240229	12.4	5.3	13.1	**10.3**	15.0	16.1	12.9	15.1	11.9	12.3	12.5	13.7	12.0	**11.7**	11.9	12.9
GPT-4-0125-Preview	13.0	4.5	14.0	11.1	15.0	19.5	13.8	16.0	13.1	13.4	11.8	13.8	13.0	12.4	13.5	13.1
Claude-3-Haiku-20240307	13.6	5.1	14.4	12.0	14.4	16.6	14.4	17.1	13.0	13.9	13.2	14.5	13.4	13.9	13.1	14.0
GPT-3.5-Turbo-0125	13.7	4.8	14.2	11.7	14.3	18.5	14.1	18.5	13.1	13.6	13.6	13.5	13.8	14.0	14.1	14.2
Gemini-1.5-Pro-001	15.0	6.1	17.5	13.5	20.1	17.7	17.9	13.6	14.0	14.4	15.0	14.6	15.2	15.8	15.4	15.3
Mistral-7B-Instruct-v0.3	11.8	8.4	**12.4**	10.8	14.0	**13.9**	12.2	**12.2**	14.4	12.4	**10.4**	**8.5**	**11.6**	13.0	12.3	11.6
Hermes-2-Pro-Mistral-7B	15.3	9.6	16.1	14.7	15.2	22.8	14.0	17.0	14.0	14.6	14.6	15.6	16.0	15.6	16.3	15.8
Gorilla-Openfunctions-v2	24.2	26.6	23.9	23.9	25.8	23.8	24.4	23.5	26.6	26.5	24.1	13.8	25.0	25.4	26.3	26.1
C4AI-Command-R-v01	29.7	30.0	29.7	29.6	30.0	29.9	29.6	29.5	29.4	29.8	29.7	29.8	29.2	29.7	29.9	29.9
C4AI-Command-R+	30.0	30.0	30.1	30.1	30.0	30.0	30.1	30.0	30.0	30.0	30.0	30.0	30.1	30.0	30.0	30.0

Table 11: Comparing the average turn count broken down by scenario category and tool augmentations. Columns from left to right represent average turn count across all categories, then **S**ingle **T**ool **C**all, **M**ultiple **T**ool **C**all, **S**ingle **U**ser **T**urn, **M**ultiple **U**ser **T**urn, **S**tate **D**ependency, **C**anonicalization, **I**nsufficient **I**nformation, **0 D**istraction **T**ools, **3 D**istraction **T**ools, **10 D**istraction **T**ools, **A**ll **T**ools, **T**ool **N**ame **S**crambled, **T**ool **D**escription **S**crambled, **A**rgument **D**escription **S**crambled and **A**rgument **T**ype **S**crambled. Note that turn count should not be interpreted in isolation, considering that a model could also be confidently wrong, finishing a task early without properly completing the user goal. As such, one should compare turn count between similarly similarity scored models, to showcase their difference in efficiency.