WRIT: Write-Read Intensive Trajectory Synthesis for Multi-Turn User-Facing Agents

Hengrui Gu[1] **Xiaotian Han**[2] **Kaixiong Zhou**[1]

[1]North Carolina State University [2]Case Western Reserve University

hgu6@ncsu.edu xhan@case.edu kzhou22@ncsu.edu

Homepage: https://hengrui-gu.github.io/WRIT/

Dataset: https://huggingface.co/datasets/Henryoung/WRIT-2K

Model: https://huggingface.co/collections/Henryoung/WRIT-models

Abstract

Multi-turn user-facing agents must infer user intent from incomplete requests, collect missing information through dialogue and tools, and execute valid actions. A training trajectory records this process as an interleaved sequence of user messages, agent responses, tool calls, etc. Synthesizing sufficiently complex trajectory has become a central route to train agents: existing pipelines often increase difficulty by composing multiple user requests into longer tasks, producing write-intensive trajectories that train sequential execution.

We argue that a single write decision can itself be difficult when the agent must gather and compare substantial read-tool evidence before its arguments become identifiable, a challenge that write-intensive data alone cannot address. Guided by this insight, we propose WRIT (Write-Read Intensive Trajectory Synthesis), a pipeline for synthesizing multi-turn agent training trajectories along two complexity axes: the number of write decisions in a task and the evidence burden of each individual decision. WRIT first generates write-intensive and read-heavy tasks. It then diversifies user behavior instructions to reflect realistic conversational variation, and finally simulates agent-user interactions in an executable environment to produce complete training trajectories. The resulting data trains agents not only for longer task execution, but also for robust, evidence-grounded decision making under high information load. With only 2K synthesized trajectories, a 4B model trained on WRIT outperforms GPT-5.1 no-think on τ^2-bench and substantially reduces inference-time token usage, showing that compact SFT data can convert part of expensive test-time reasoning into efficient agent behavior.

1 Introduction

Language agents equipped with tools are becoming a practical interface for automating user-facing workflows, from booking flights to changing reservations and processing returns (Lu et al., 2025; Drouin et al., 2024; Wang et al., 2026a; Fang et al., 2026; Barres et al., 2025; Qian et al., 2025; Cheng et al., 2026; Qin et al., 2026). In these multi-turn settings, an agent must infer an incomplete or evolving user intent, ask clarifying questions, read external records, follow domain policy, and execute valid state-changing actions (Lu et al., 2025; Zhao et al., 2025; Rana et al., 2025; Burdisso et al., 2026; Zhang et al., 2025). A training trajectory records this process as an interleaved sequence of user messages, agent responses, tool calls, and tool observations. High-quality trajectories are therefore the supervision that teaches an agent when to ask, when to read, which tool to call, what evidence to trust, and when it is safe to write (Zeng et al., 2026; Xu et al., 2025; Gao et al., 2026).

Since collecting such trajectories from humans is expensive, synthetic trajectory generation has become a central route for training tool-using agents. Existing work follows several routes: executable simulation pipelines roll out interactions between user and agent models (Prabhakar et al., 2025; Chen et al., 2026; Wang et al., 2026b); LLM-driven pipelines synthesize trajectories or simulate environment feedback without a complete backend (Li et al., 2025); and environment-scaling approaches construct many tool-use environments from which trajectories can be collected (Fang et al., 2026). Together, these methods expand the quantity and diversity of training data and improve benchmark performance for multi-turn tool-use agents.

Most existing synthesis pipelines increase complexity by composing multiple user requests or state-changing actions into longer tasks. This trains agents for multi-step execution, sequential decision making, and long-horizon stability. Yet these pipelines mainly teach agents to do more, while overlooking difficulty that arises before any action is taken. In realistic service scenarios, the hard part

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 12115–12134
October 24-29, 2026 ©2026 Association for Computational Linguistics

Write Action	Reason	Tool Calls

`book_reservation(`
`user_id="emma_johnson_7098",`
`origin="EWR",`
`destination="IAH",`
`flight_type="one_way",`
`cabin="business",`
`flights=[{date="2024-05-25",`
`flight_number="HAT188"}],`
`...)`
	Simple task: "I need to book a one-way business class flight from Newark to Houston on May 25. Please book the direct flight that departs at 8:00 AM and arrives at 11:30 AM."	1 x get_user_details
		1 x search_direct_flight
		1 x book_reservation

	Read-heavy task: "I need to book a one-way business class flight from the New York area to Houston. I'm flexible between May 25 and May 26, and I can depart from either Newark or LaGuardia. Please book the fastest overall flight."	1 x get_user_details 4 x search_direct_flight 4 x search_onestop_flight 1 x book_reservation

Table 1: A simple task and a read-heavy task can share the same gold write action, while differing in the amount of read evidence required to determine its arguments. The simple task uses 2 read-tool calls before the write action, whereas the read-heavy task uses 9 read-tool calls before executing the same booking action. Read tools are shown in blue and write tools in orange.

is often gathering and comparing enough read-tool evidence to determine what arguments an action should carry. Users rarely provide all necessary identifiers; instead, they express preferences and descriptions, leaving the agent to search broadly before committing a state change. This motivate a new data synthesis question:

> *Beyond teaching agents to act for longer, can we synthesize trajectories that teach them to read more carefully before they act?*

Table 1 makes this distinction concrete. Both tasks share the same gold write action, `book_reservation(...)`, so from a write-action perspective they are identical. The difference is what the agent must do before writing. In the simple task, the user specifies the target flight by departure and arrival time, so one local search is enough. In the read-heavy task, the user asks for the fastest overall flight across multiple dates and departure airports, so the agent must search every airport-date combination, compare all returned candidates, and recover the correct `flight_number`; the read-tool count rises from 2 to 9. An agent trained only on shallow lookups may fail on such requests because it never learned to plan broad search, integrate evidence, and defer commitment until the arguments are grounded. Read-heavy trajectories are therefore a structurally distinct form of training complexity.

Motivated by this observation, we propose WRIT (Write-Read Intensive Trajectory Synthesis), a pipeline that synthesizes training trajectories covering both action execution and evidence-intensive decision making. First, WRIT generates service tasks with verifiable correct outcomes, spanning tasks with multiple sequential actions (i.e., write-intensive) and tasks where one action requires extensive reading and comparison (i.e., read-intensive). Second, WRIT varies how users express and reveal the same request, so training data reflects realistic conversational behaviors rather than only cooperative, fully specified interactions. Third, WRIT runs the agent and user through each task in an executable environment and retains successful interactions as complete training trajectories. Figure 1 summarizes this pipeline.

We evaluate WRIT on τ^2-bench using a controlled 2K-trajectory training budget against strong synthetic-data baselines.

- WRIT consistently outperforms prior trajectory synthesis methods across all three tested models (Qwen3-4B-Instruct-2507, Llama-3.1-8B-Instruct, Qwen2.5-14B-Instruct), with especially large gains on read-heavy task subsets.
- A 4B model trained with only 2K WRIT trajectories outperforms GPT-5.1 no-think on τ^2-bench and substantially narrows the gap to GPT-5.1 thinking, while using far fewer output tokens at inference time.
- Ablations confirm that both read-heavy task synthesis and user-behavior diversification contribute independently.

These results show that a small, carefully structured set of trajectories balancing write-intensive and read-intensive complexity can produce more capable and reliable agents than much larger but less structured datasets. Synthetic data should teach agents not only to act more, but also to know more before they act.

2 Problem Setup and Design Rationale

2.1 Problem Setup. We consider a user-facing operational domain, such as airline customer service, where an agent interacts with

a user while operating over a database, a set of tools, and domain policy rules (Yao et al., 2024; Barres et al., 2025). The tools include read tools, which observe the environment without changing it, such as `search_direct_flight(origin,destination,date)` for retrieving matching flight candidates. They also include write tools, which update the environment state, such as `book_reservation(user_id, origin,destination,flights,...)` for creating a flight reservation. Domain policy rules constrain when write tools may be used, including rules such as *"All reservations can be cancelled within 24 hours of booking."*

A task specifies what the user wants the agent to accomplish and what a correct outcome looks like. We formalize a task as a tuple consisting of a user request u, an initial database state s_{init}, a gold write-action sequence A_{gold}, and a gold final database state s_{gold}. Here, u is the natural-language goal, s_{init} gives the starting conditions, A_{gold} specifies the correct state-changing actions, and s_{gold} is obtained by executing A_{gold} from s_{init} in a sandboxed environment. For a booking task, for example, s_{gold} is the database state after the correct reservation has been created, and task success is evaluated by checking whether the executed outcome matches s_{gold}.

While the task defines what the agent must do, a training trajectory defines how the agent does it in a real conversation. A trajectory τ is the complete multi-turn interaction record generated by simulating the task, interleaving user messages, agent responses, tool calls, and tool observations across conversation turns. As supervised fine-tuning data, a trajectory teaches the agent when to ask for more information, which tool to call and with what arguments, how to interpret tool outputs, and when to execute a write action. Our pipeline first synthesizes tasks, then uses each task to simulate a trajectory, which lets us control task difficulty independently from how the trajectory unfolds.

2.2 Two-Axis Trajectory Complexities. To synthesize useful training trajectories, we need to understand what makes a write decision difficult for the agent. The challenge is not only choosing the right write tool, but resolving the correct argument values from the user request, the conversation context, and tool observations; we call this process ***argument grounding***. For example, to book the right flight, the agent must determine the specific `flight_number` by reading flight search results, rather than being told it directly. Each write action is therefore a *decision point*: before committing an action to environment, the agent must fully ground both the tool choice and its argument values.

This framing yields two independent ways to make agent training harder and more comprehensive. The first axis is the number of write decisions in a task: increasing it produces *write-heavy trajectories* that train the agent on long-horizon sequential decision making. The second axis is the evidence burden of a single decision: increasing this axis produces *read-heavy trajectories*, where one write action requires the agent to collect and compare multiple read-tool outputs before grounding its arguments. This second axis is important and comparatively underexplored: without read-heavy trajectories, an agent trained only on simple decisions may learn to act after a single lookup and fail when a real user's request requires searching across multiple options, dates, or alternatives before any valid write can be taken.

Our synthesis objective is therefore to generate training trajectories along both axes. Together, write- and read-heavy trajectories teach the agent both long-horizon execution stability and evidence-intensive grounding under high information load.

3 WRIT for Multi-turn Agent Training

Guided by this goal, we propose WRIT (Write-Read Intensive Trajectory Synthesis), a pipeline for generating multi-turn agent training data in three stages. First, WRIT synthesizes write-read intensive tasks with known correct outcomes, covering both write-intensive service requests and read-heavy requests that require substantial evidence gathering. Second, WRIT designs user behavior instructions that diversify how the user expresses and reveals the same underlying task across trajectories, so that training data reflects realistic conversational variation. Third, WRIT runs the agent and user simulator through each task and behavior instruction in an executable environment, collecting successful interactions as complete supervised fine-tuning trajectories. In this workflow, the first two stages prepare the inputs, namely the task and behavior instruction, and the final stage turns them into training trajectories.

3.1 Write-Read Intensive Task Synthesis

WRIT first synthesizes tasks, each consisting of a user request u, an initial database state s_{init}, a

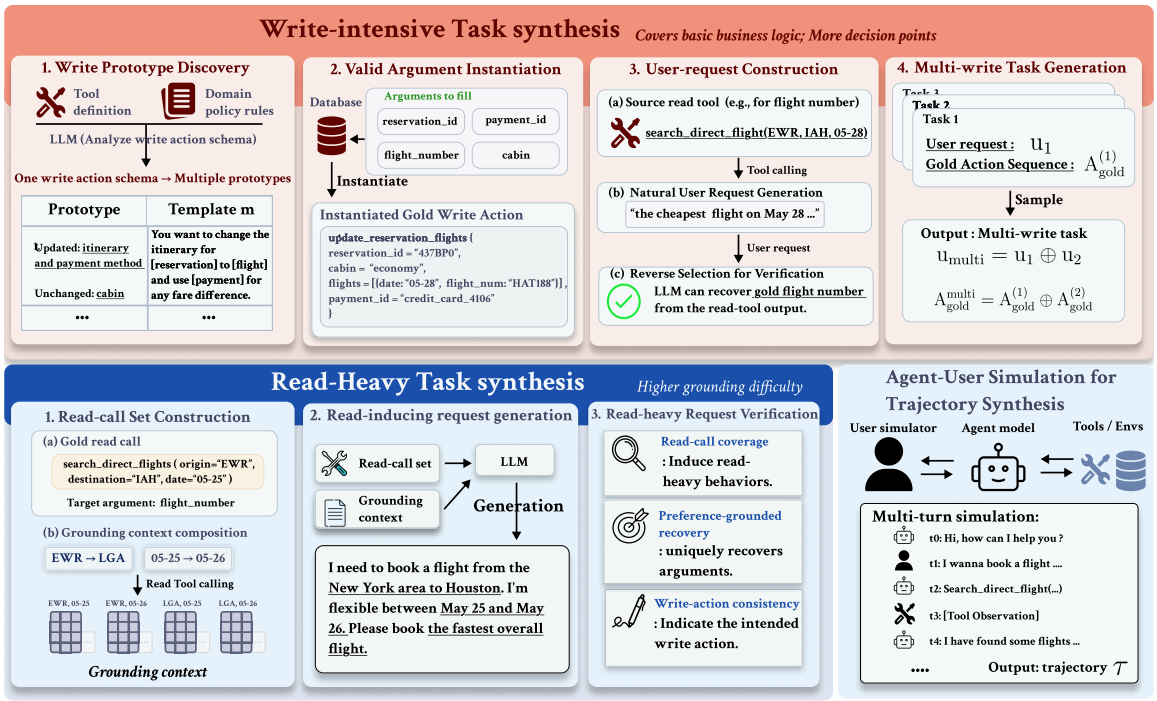

Figure 1: Overview of the WRIT pipeline.

gold write-action sequence A_{gold}, and a gold final state s_{gold}. This subsection focuses entirely on task synthesis; the simulation that turns tasks into trajectories is introduced later in Section 3.3. We control task complexity through following two branches.

3.1.1 Write-intensive task synthesis. This branch synthesizes trajectories that cover the core write operations of the domain. Each trajectory trains the agent to identify common user intents, follow domain policy, and execute write actions with correctly grounded arguments. We describe the process in four steps.

Step 1: Write prototype discovery. The synthesis starts from identifying the popular write operations and user-facing scenarios the agent should learn to handle. We use an LLM to analyze the tool definitions and domain policy rules, and automatically derive a set of operation prototypes: each prototype captures a meaningful usage pattern for a write action and is paired with a natural-language template m that describes the corresponding user intent with slots for grounded argument values. For example, one prototype for `update_reservation_flights(reservation_id,cabin,flights,payment_id)` captures the pattern where the user wants to change the itinerary and payment method, producing a template such as "You want to change the itinerary for [reservation] to [flight] and use [payment] for any fare difference." These templates keep the generated user requests stable and semantically aligned with the target write action.

Step 2: Valid argument instantiation. This step populates each prototype with concrete, valid argument values drawn from the current database state. For each prototype, we sample a feasible combination of database records that satisfies the prototype's constraints, such as selecting a user, one of the user's reservations, and a target cabin class that differs from the current one. This produces a fully instantiated gold write action A_{gold}.

Step 3: User-request construction. Based on the sampled write tool and arguments, we can construct a natural user request that expresses the intent behind the gold write action without directly exposing backend identifiers. Rather than inserting raw argument values, such as a flight number, into the request (Prabhakar et al., 2025), we describe each argument through a natural preference, such as "the cheapest flight" instead of a literal flight ID (Chen et al., 2026), so the agent must read the environment to resolve it. The verified descriptions are accepted and filled into the natural-language template m to form the final user request u.

Step 4: Multi-write task generation. We finally extend single-decision trajectories into multi-step trajectories that require the agent to complete several sequential write actions. For example, we combine two write-intensive trajectories by concatenating their user requests and gold write sequences into a single compound trajectory, i.e., $u_{\text{multi}} = u_1 \oplus u_2$ and $A^{\text{multi}}_{\text{gold}} = A^{(1)}_{\text{gold}} \oplus A^{(2)}_{\text{gold}}$, with programmatic

checks to prevent unintended execution conflicts. The resulting multi-write trajectories challenge the agent to sustain correct decision making across multiple decision points without losing track of the user's overall goal.

3.1.2 Read-heavy task synthesis. This branch synthesizes tasks in which a single write decision requires the agent to gather and compare evidence from multiple read-tool calls before the correct argument can be determined. Unlike write-intensive tasks, where arguments can be resolved through a small number of direct lookups, read-heavy tasks force the agent to search broadly, compare candidates across multiple tool outputs, and select the correct argument based on the user's preference. The construction proceeds in three steps.

Step 1: Read-call set construction. The synthesis process must first determines the full set of read-tool calls the agent should make to resolve the target write argument, and collect their outputs as an evidence pool. Starting from an instantiated gold write action, we identify one argument as the read-heavy target, i.e., the value the agent must discover through tool use. We identify the single read-tool call that contains this argument, which we call the gold read call, e.g., `search_direct_flight(origin="EWR", destination="IAH",date="2024-05-25")` in Table 1, then generate perturbed variants of it by varying parameters such as date or departure airport. The outputs of all these calls form the *grounding context*: the evidence pool the agent must compare to find the correct value.

Step 2: Read-inducing request generation. It then generates a natural user request that requires the agent to consult the full evidence pool rather than stopping at a single lookup. An LLM generates the user request given the read-call set and grounding context, under two requirements: the stated user preference must lead the agent to consult all specified read-tool outputs, and it must uniquely identify the correct gold argument from the returned evidence. For example, "fastest overall flight" requires comparing candidates across all the searched airport-date combinations, as shown in Table 1.

Step 3: Read-heavy request verification. Finally, we verify that the generated request actually induces the intended evidence-gathering behavior and remains solvable. An LLM verifier checks following three properties:

- *Read-call coverage:* the request should imply all lookup operations in the read-call tool set.
- *Preference-grounded recovery:* the verifier should recover gold argument from grounding context based on the stated user preference.
- *Write-action consistency:* the request should clearly indicate the intended write action while leaving the read-heavy target argument to be resolved from evidence.

Requests that fail any check are discarded. The read-heavy task synthesis branch has now produced a verified user request u and gold write-action sequence A_{gold}.

3.1.3 Gold-state construction. After either synthesis branch produces a user request u and a gold write-action sequence A_{gold}, both branches enter the same gold-state construction step. We execute A_{gold} in a sandboxed environment initialized with s_{init} to obtain the gold final state s_{gold}.The resulting state provides the executable supervision signal used later to verify whether a simulated trajectory actually completes the intended task.

3.2 User Behavior Diversification

Diversifying user behavior across trajectories is essential for training agents that remain robust when real users express the same request in different ways (Ferreira et al., 2024). The same task can unfold into many different conversations depending on how the user behaves: a user may reveal information gradually, correct a mistake mid-conversation, or add irrelevant small talk (Hu et al., 2025). If the training trajectories always assume a cooperative and information-complete user, the agent may be fragile at test time. User behavioral variation changes the conversational path but not the underlying goal or correct write action (Hu et al., 2025). This means we can explicitly diversify user behavior without changing the task's supervision signal: A_{gold} and s_{gold} stay fixed.

WRIT maintains a library of reusable behavior instruction primitives, each describing a specific user behavior pattern. General task-completion primitives cover behaviors that arise in ordinary service conversations, including progressive disclosure, where the user reveals information gradually; self-correction, where the user fixes a stated value after a challenge; confirmation hesitation, where the user verifies the agent's summary before agreeing; mild emotion; and irrelevant asides (Algherairy and Ahmed, 2025). Policy-robustness primitives cover behaviors that specifically pres-

sure the agent's policy boundary, including false-premise assertions, assume-style pressure, prior-agent approval claims, complaint pressure, and social flattery (Hu et al., 2025). These are needed because policy-sensitive tasks require the agent to refuse or redirect gracefully under adversarial user strategies that ordinary task-completion primitives do not cover.

For each synthesized task, we select a small number of compatible primitives from the library and prompts an LLM to instantiate them as concrete user-simulator instructions tailored to that task; for example, the instruction may ask the user simulator to initially give the wrong date and correct it only after the agent challenges it. These instructions govern only interaction style, namely how and when the user reveals information, not task content, namely what the user wants or which write action should be executed. The instructions are passed to the user simulator alongside the task request u in Section 3.3. Appendix J lists the script primitives used in our implementation and provides concrete examples of instantiated scripts.

3.3 Trajectory Simulation and Filtering

This is where the synthesized task and user behavior instructions come together to produce complete training trajectories (Prabhakar et al., 2025; Fang et al., 2026). Section 3.1 provides the task $(u, s_{\text{init}}, A_{\text{gold}}, s_{\text{gold}})$, and Section 3.2 provides the user behavior instructions. With these inputs, we initialize the executable environment at s_{init} and run two models simultaneously: a user simulator guided by the task request u and the behavior instructions, and an agent model given the domain policy and tool definitions. They interact turn by turn: the user expresses requests according to the behavior instructions, the agent responds and issues tool calls, and the environment executes those calls until the task is completed or refused. The output is a complete trajectory τ interleaving user messages, agent responses, tool calls, and tool observations.

Because the agent model may make errors during simulation, not all trajectories successfully realize the intended task, so we filter the data to keep only correct and complete demonstrations. The retained trajectories form the training corpus $\mathcal{T}$ used for supervised fine-tuning. Since each retained trajectory comes from either a write-intensive or read-heavy task with a verified gold outcome, $\mathcal{T}$ systematically covers both axes of complexity defined in Section 2. Additional simulation details are provided in Appendix I.

4 Experiments

Experimental setting. We study in-domain specialization for agentic tasks, where training trajectories are synthesized using the predefined tool definitions and domain policies of an operational environment, and evaluation is conducted on unseen tasks within the same environment. This setup reflects deployment scenarios in which agents are specialized to known operational requirements before serving new user requests. We conduct our experiments under this setting, using the retail and airline domains of τ^2-bench as the operational environments for synthesizing training trajectories. Both domains provide executable tools, domain policies, database states, and state-based success checks (Barres et al., 2025).

Training data. Using WRIT, we synthesize 2K training trajectories with balanced domain coverage, including 1K trajectories for the τ^2-bench retail domain and 1K trajectories for the airline domain. Details of the trajectory synthesis setup are provided in Appendix I.

Baselines. We compare WRIT with four synthetic trajectory generation methods for multi-turn user-facing agents: APIGen-MT (Prabhakar et al., 2025), Simia (Li et al., 2025), CoVe (Chen et al., 2026), and AReaL (Gao et al., 2026). For each baseline, we use its publicly released dataset synthesized for the τ^2-bench retail and airline domains using the corresponding tool definitions and domain policy rules. These methods represent different synthesis strategies, including simulated agent–user interaction, seed-set expansion with simulated environment feedback, rule-based argument transformation, and LLM-controlled synthetic data generation. To compare data synthesis recipes under the same supervised fine-tuning budget, we use a controlled 2K trajectory-level setting for all main experiments. For public baselines with larger released datasets, we uniformly sample 2K trajectories at the trajectory level. This protocol isolates the effect of trajectory quality and task composition from the effect of dataset scale; we additionally report full-size baseline comparisons in Appendix B.

Evaluation. We evaluate on τ^2-bench, covering both retail and airline domains. In addition to the full task sets, we report performance on fixed read-heavy subsets where the hardest decision point requires about six or more read/search calls; the sub-

Model	Dataset	τ^2 Retail		τ^2 Airline		τ^2 Average		τ^2 Retail-Hard		τ^2 Airline-Hard	
		Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4
Qwen3-4B -Instruct-2507	APIGen-MT	50.00±2.77	23.68	20.00±4.90	6.00	40.85±2.28	18.29	43.15±2.03	14.52	17.50±6.45	5.00
	Simia	53.73±2.62	25.44	31.00±6.63	10.00	46.80±2.36	20.73	42.74±4.06	14.52	21.25±7.50	0.00
	CoVe	59.65±1.60	31.58	37.50±6.40	20.00	52.90±2.46	28.05	53.63±2.42	22.58	33.75±6.29	20.00
	AReaL	59.43±5.13	32.46	47.00±3.46	36.00	55.64±3.60	33.54	52.42±11.82	25.81	42.50±8.66	25.00
	WRIT	**71.05**±1.24	**47.37**	**61.00**±3.83	**42.00**	**67.99**±1.90	**45.73**	**66.13**±2.28	**38.71**	**57.50**±6.45	**40.00**
Llama-3.1-8B -Instruct	APIGen-MT	42.98±1.75	18.42	20.50±4.12	6.00	36.13±2.19	14.63	34.27±0.81	9.68	22.50±5.00	5.00
	Simia	40.79±3.40	17.54	23.00±2.00	8.00	35.37±2.49	14.63	33.47±3.58	12.90	16.25±4.79	10.00
	CoVe	52.19±1.13	27.19	32.00±4.90	14.00	46.04±1.17	23.17	**51.21**±1.54	25.81	27.50±2.89	15.00
	AReaL	45.18±0.51	20.18	43.50±3.42	24.00	44.66±0.77	21.34	37.50±4.63	17.74	38.75±2.50	15.00
	WRIT	**54.61**±2.90	**31.58**	**50.00**±5.89	**32.00**	**53.20**±3.32	**31.71**	47.58±5.01	**27.42**	**46.25**±7.50	**30.00**
Qwen2.5-14B -Instruct	APIGen-MT	50.00±3.72	24.56	27.00±2.00	12.00	42.99±2.84	20.73	43.15±5.65	22.58	17.50±5.00	5.00
	Simia	51.10±2.42	28.95	34.50±1.91	18.00	46.04±1.76	25.61	40.32±3.95	17.74	23.75±4.79	10.00
	CoVe	58.11±4.08	31.58	34.50±3.00	16.00	50.91±3.61	26.83	53.63±5.49	27.42	30.00±4.08	10.00
	AReaL	57.68±5.03	31.58	43.00±2.58	28.00	53.20±3.90	30.49	50.81±6.52	29.03	30.00±5.77	10.00
	WRIT	**72.37**±1.68	**47.37**	**57.50**±4.43	**38.00**	**67.84**±2.51	**44.51**	**66.13**±2.28	**37.10**	**46.25**±10.31	**20.00**

Table 2: Tau2-bench evaluation results. τ^2 Retail/Airline report success over all domain tasks, τ^2 Average is task-count weighted across Retail and Airline, and Retail-Hard/Airline-Hard report fixed read-heavy subsets where the hardest decision point requires about six or more read/search calls. The exact hard-subset task groupings are listed in Appendix D. Pass^1 includes sample standard deviation across four trials. All numbers are percentages.

set definitions are provided in Appendix D. We use the Pass^k reliability metric (Yao et al., 2024). For each task i, we run n independent trials and let c_i denote the number of successful trials. The Pass^k score is computed as $\frac{1}{|\mathcal{Q}|}\sum_{i\in\mathcal{Q}}\binom{c_i}{k}/\binom{n}{k}$, where $\mathcal{Q}$ is the evaluated task set. Intuitively, Pass^1 is the average success rate over repeated trials, while Pass^k estimates the probability that k randomly sampled trials for the same task all succeed. Larger k therefore gives a stricter measure of reliability, because a model must solve the same task consistently rather than succeed only occasionally. In our experiments, we run each task four times and report Pass^1 and Pass^4. Higher Pass^k indicates more consistent behavior across repeated attempts.

Models and implementation details. We focus on non-thinking agent settings, where the deployed model must act directly without explicit long-form reasoning. We therefore fine-tune multiple instruction-tuned base models, including Qwen3-4B-Instruct-2507, Llama-3.1-8B-Instruct, and Qwen2.5-14B-Instruct. For each base model and dataset, we perform full-parameter supervised fine-tuning on the corresponding training trajectories. Dataset statistics, training hyperparameters, and additional implementation details are provided in Appendix E.

4.1 Results and Analysis

WRIT expands the agent capability boundary and improves reliability. Table 2 shows that WRIT substantially improves multi-turn agent performance across model families. On Qwen3-4B-Instruct-2507, WRIT achieves a τ^2 Average Pass1 of 67.99, outperforming AReaL by 12.35 points, and improves Pass4 from 33.54 to 45.73. The same pattern holds for Llama-3.1-8B-Instruct, where WRIT improves the average Pass1 from 46.04 with CoVe to 53.20, and for Qwen2.5-14B-Instruct, where WRIT improves the average Pass1 from 53.20 with AReaL to 67.84. Higher Pass1 suggests that the trained agent can solve a broader set of tasks in a single attempt, while higher Pass4 indicates more stable behavior across repeated trials. Together, these gains show that WRIT improves both capability coverage and reliability in multi-turn user-facing settings.

Read-heavy synthesis addresses a key weakness of user-facing agents. The gains are especially clear on the read-heavy subsets, which correspond to difficult τ^2-bench tasks requiring substantial read/search behavior before the final decision. For Qwen3-4B-Instruct-2507, WRIT improves Airline-Hard Pass1 from 42.50 with AReaL to 57.50, and improves Pass4 from 25.00 to 40.00. On Retail-Hard, WRIT also improves Pass1 from 53.63 with CoVe to 66.13. Similar improvements appear for other two base models, as also shown by the Pass^k curves in Figure 2. These results suggest that our synthesized trajectories directly improve a capability gap in current user-facing agents: they need practice not only executing tools, but also gathering

Variant	τ^2 Retail		τ^2 Airline		τ^2 Average		τ^2 Retail-Hard		τ^2 Airline-Hard	
	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4
WRIT	**71.05**$\pm$1.24	**47.37**	**61.00**$\pm$3.83	**42.00**	**67.99**$\pm$1.90	**45.73**	**66.13**$\pm$2.28	**38.71**	**57.50**$\pm$6.45	**40.00**
w/o read-heavy	67.11$\pm$3.24	46.49	52.50$\pm$5.74	30.00	62.65$\pm$3.39	41.46	57.66$\pm$3.05	33.87	41.25$\pm$6.29	15.00
w/o script	69.96$\pm$4.72	45.61	49.50$\pm$6.61	26.00	63.72$\pm$2.84	39.63	62.90$\pm$8.22	32.26	45.00$\pm$17.80	20.00
w/o multi-write	67.32$\pm$3.31	43.86	55.00$\pm$4.16	32.00	63.57$\pm$1.15	40.24	59.27$\pm$4.03	29.03	51.25$\pm$8.54	30.00

Table 3: Ablation results on τ^2-bench using Qwen3-4B-Instruct-2507. We report Pass^1 and Pass^4 on the full Retail/Airline task sets, their task-count weighted average, and the fixed read-heavy hard subsets.

Method	Retail	Airline	Avg.	Output Tokens (USD)
GPT-5.1 thinking	**82.46**	**72.00**	**79.27**	1,520,619 ($17.52)
GPT-5.1 no-think	69.30	48.00	62.80	318,180 ($5.56)
WRIT-4B	71.05	61.00	67.99	**251,405 (–)**

Table 4: GPT-5.1 evaluation results on τ^2-bench. Retail, Airline, and Avg. are percentages. Output Tokens counts agent-side completion tokens for one full τ^2 evaluation, with agent-side API cost shown in parentheses.

and comparing enough evidence before committing to a write action.

Pipeline components specialize into complementary capabilities. Table 3 shows that all three components contribute to the final performance. **Read-heavy grounding strengthens evidence-intensive decisions.** Removing read-heavy trajectories only mildly reduces Retail Pass1 from 71.05 to 67.11, but the drop becomes much larger on Retail-Hard, from 66.13 to 57.66. The effect is even more pronounced on Airline-Hard, where Pass1 drops from 57.50 to 41.25 and Pass4 collapses from 40.00 to 15.00. This pattern directly supports our main hypothesis: read-heavy samples do not merely improve general performance, but specifically improve the agent's capability and stability on difficult tasks that require substantial evidence gathering before acting. **Scripts improve robustness near the policy boundary.** Removing scripts causes the largest full-domain drop on Airline, reducing Pass1 from 61.00 to 49.50 and Pass4 from 42.00 to 26.00; it also substantially hurts Airline-Hard, where Pass4 falls from 40.00 to 20.00. The retail domain is less affected, suggesting that the script layer is especially valuable in policy-sensitive settings where adversarial user patterns, such as false premises, pressure, or delayed policy-relevant information, stress the agent's refusal and policy-following behavior. **Multi-write composition provides useful task-composition coverage.** Removing multi-write trajectories lowers the average Pass1 from 67.99 to 63.57 and Pass4 from 45.73 to 40.24, with the largest drop appearing on Retail-Hard Pass4, from 38.71 to

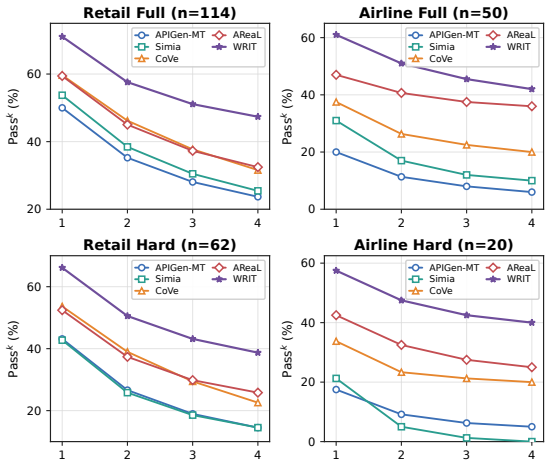

Figure 2: Pass^k curves for Qwen3-4B-Instruct-2507. The horizontal axis indexes $k = 1, 2, 3, 4$.

29.03. This indicates that compound tasks still provide important training signal for maintaining correctness across multiple requested operations. Overall, the ablations show that both complexity-oriented sample types, multi-write and read-heavy, substantially improve performance, while scripts mainly improve stability under user-side variation and policy-boundary stress.

WRIT approaches strong API agents with substantially lower inference cost. Table 4 compares WRIT with GPT-5.1 variants on τ^2-bench. Although GPT-5.1 thinking achieves the highest score, it uses over 1.5M output tokens for one full Retail+Airline evaluation, reflecting the high inference cost of relying on test-time reasoning. In contrast, WRIT outperforms GPT-5.1 no-think on both domains, improving the average Pass1 from 62.80 to 67.99, while using fewer output tokens. This suggests that our synthesized trajectories transfer part of the required evidence-gathering and policy-following behavior into the model parameters through SFT, allowing a smaller non-thinking agent to act more efficiently at inference time. The gap to GPT-5.1 thinking further indicates that explicit reasoning remains powerful, but WRIT provides a cost-effective alternative when deployment requires direct, low-token agent behavior.

	Retail ($n = 114$)					Airline ($n = 50$)				
Dataset	≥ 4 ($n = 102$)	≥ 5 ($n = 84$)	≥ 6 ($n = 62$)	≥ 7 ($n = 29$)	≥ 8 ($n = 17$)	≥ 4 ($n = 28$)	≥ 5 ($n = 25$)	≥ 6 ($n = 20$)	≥ 7 ($n = 15$)	≥ 8 ($n = 7$)
APIGen-MT	48.53/21.57	45.24/16.67	43.15/14.52	37.07/6.90	22.06/0.00	16.07/3.57	15.00/4.00	17.50/5.00	13.33/6.67	21.43/14.29
Simia	50.49/23.53	45.24/16.67	42.74/14.52	27.59/6.90	19.12/0.00	23.21/3.57	21.00/0.00	21.25/0.00	20.00/0.00	10.71/0.00
CoVe	58.09/30.39	57.14/26.19	53.63/22.58	45.69/17.24	38.24/11.76	35.71/17.86	35.00/20.00	33.75/20.00	31.67/13.33	21.43/14.29
AReaL	57.84/30.39	55.65/27.38	52.42/25.81	39.66/13.79	42.65/17.65	41.07/28.57	38.00/24.00	42.50/25.00	38.33/20.00	25.00/14.29
WRIT	**70.10/45.10**	**69.05/42.86**	**66.13/38.71**	**55.17/24.14**	**45.59/23.53**	**62.50/42.86**	**59.00/40.00**	**57.50/40.00**	**55.00/40.00**	**35.71/28.57**

Table 5: Sensitivity to the read-heavy cutoff. Each cell reports $\mathrm{Pass}^1/\mathrm{Pass}^4$ (%) over four trials using Qwen3-4B-Instruct-2507; n denotes the number of tasks retained at each cutoff. The default cutoff is six reads.

	Volume per trajectory				Evidence
Dataset	Turns	Calls	Writes	Tokens (K)	R/W
APIGen-MT	18.46	4.40	1.09	7.52	2.78
Simia	17.53	4.17	1.16	6.85	2.47
CoVe	24.36	6.78	1.73	10.30	2.89
AReaL	**28.27**	**8.99**	**1.83**	**12.64**	3.87
WRIT	24.63	6.61	1.12	9.03	**4.78**

Table 6: Full-dataset trajectory statistics (per-trajectory averages). Turns include user, assistant, and tool messages; Calls and Writes denote all tool calls and database-changing calls, respectively. Tokens are reported in thousands using the Qwen3-4B-Instruct-2507 tokenizer, and R/W denotes the ratio of mean read calls to mean write calls. Column maxima are bolded.

Performance gains are robust to the read-heavy cutoff. Table 5 reports results across read-heavy cutoffs from four to eight reads. WRIT achieves the highest Pass^1 and Pass^4 in both domains at every cutoff, showing that its advantage is not specific to the default subset definition. As the cutoff increases, the retained tasks become more read-intensive and the subset sizes decrease, but the relative performance pattern remains stable. We use six reads as the default because it corresponds to the median split of the benchmark, retaining 82 of 164 tasks, including 62 Retail and 20 Airline tasks. Raising the cutoff to seven reduces the evaluation set to 44 tasks, and raising it to eight leaves only 24 tasks, whereas lower cutoffs include a larger proportion of less read-intensive tasks. The six-read cutoff therefore provides a balance between read intensity and evaluation stability.

WRIT shifts trajectory complexity toward evidence gathering. Table 6 compares the structural profiles of the synthesized datasets. AReaL is the largest on all volume-oriented measures, with the most turns, tool calls, write actions, and tokens per trajectory. CoVe is close to WRIT in overall interaction volume, but contains substantially more write actions. In contrast, WRIT is not the largest on any volume measure, yet has the highest evidence burden: 4.78 read calls per write call, compared with 3.87 for the next highest dataset. This profile shows that WRIT shifts trajectory complexity toward gathering evidence before state-changing actions rather than simply increasing interaction length or operation count. By introducing read-heavy complexity as a complementary axis to write-oriented task composition, WRIT provides compact supervision with more balanced coverage of evidence-gathering and state-changing capabilities. Together with the preceding ablations, this structural difference is consistent with the larger gains observed on read-heavy tasks.

5 Conclusion

We presented WRIT, a trajectory synthesis pipeline for multi-turn user-facing agents that controls task complexity along two axes: the number of write decisions and the read evidence required to resolve each decision. By combining decision-coverage tasks, read-heavy grounding tasks, and scripted user behaviors, WRIT produces clean SFT trajectories in executable environments. Experiments on τ^2-bench show significant gains across models, especially on read-heavy hard subsets, demonstrating the importance of two-axis complexity control and opening a promising direction for future agentic trajectory synthesis.

Limitations

Compositional hard samples. WRIT controls task complexity along two axes: increasing the number of decision points through multi-write tasks, and increasing the grounding difficulty of individual decision points through read-heavy tasks. In this work, we study these two axes mostly as separate sources of difficulty. We have not fully explored their composition, such as constructing multi-write tasks where each decision point is also read-heavy. Such samples may further stress long-horizon state tracking and evidence-intensive

grounding at the same time.

Mixture of complexity types. Our training data contains both decision-coverage samples and read-heavy grounding samples, but we do not exhaustively study the optimal mixture ratio between different complexity types. Different model families or base capabilities may benefit from different proportions of low-read, multi-write, read-heavy, and policy-robust samples. A more systematic mixture study could clarify how each type of synthetic trajectory shapes agent behavior during supervised fine-tuning.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback, which helped improve this paper. This work was supported in part by the National Science Foundation (NSF) under Grants CNS-2431516 and IIS-2551751, and by the National Institutes of Health (NIH) under Grant 1OT2OD038051-01.

References

Atheer Algherairy and Moataz Ahmed. 2025. Prompting large language models for user simulation in task-oriented dialogue systems. *Computer Speech & Language*, 89:101697.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33538–33547, Suzhou, China. Association for Computational Linguistics.

Sergio Burdisso, Séverin Baroudi, Yanis Labrak, David Grünert, Pawel Cyrta, Yiyang Chen, Srikanth Madikeri, Esaú Villatoro-Tello, Ricard Marxer, and Petr Motlicek. 2026. Sdialog: A python toolkit for end-to-end agent building, user simulation, dialog generation, and evaluation. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 320–340.

Jinpeng Chen, Cheng Gong, Hanbo Li, Ziru Liu, Zichen Tian, Xinyu Fu, Shi Wu, Chenyang Zhang, Wu Zhang, Suiyun Zhang, Dandan Tu, and Rui Liu. 2026. Cove: Training interactive tool-use agents via constraint-guided verification. *Preprint*, arXiv:2603.01940.

Xiang Cheng, Yulan Hu, Xiangwen Zhang, Lu Xu, Lide Tan, Zheng Pan, Xin Li, and Yong Liu. 2026. Beyond itinerary planning—a real-world benchmark for multi-turn and tool-using travel tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 29200–29251.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. 2024. Workarena: How capable are web agents at solving common knowledge work tasks? *Preprint*, arXiv:2403.07718.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, Shibin Wu, Zhengwei Tao, Yong Jiang, Pengjun Xie, Ningyu Zhang, Fei Huang, Wentao Zhang, and Jingren Zhou. 2026. Towards general agentic intelligence via environment scaling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 17610–17621, San Diego, California, United States. Association for Computational Linguistics.

Rafael Ferreira, David Semedo, and João Magalhães. 2024. Multi-trait user simulation with adaptive decoding for conversational task assistants. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 16105–16130.

Jiaxuan Gao, Jiaao Chen, Chuyi He, Shusheng Xu, Di Jin, and Yi Wu. 2026. From self-evolving synthetic data to verifiable-reward rl: Post-training multi-turn interactive tool-using agents. *arXiv preprint arXiv:2601.22607*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Zhiqiang Hu, Nancy F Chen, and Roy Ka-Wei Lee. 2025. Are current task-oriented dialogue systems able to satisfy impolite users? *IEEE Transactions on Computational Social Systems*, 12(5):2876–2887.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. 2025. Simulating environments with reasoning models for agent training. *Preprint*, arXiv:2511.01824.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2025. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183, Albuquerque, New Mexico. Association for Computational Linguistics.

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

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc.

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Zhiwei Liu, Jianguo Zhang, Haolin Chen, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, Caiming Xiong, and Huan Wang. 2025. Userbench: An interactive gym environment for user-centric agents. *Preprint*, arXiv:2507.22034.

Tian Qin, Felix Bai, Ting-Yao Hu, Raviteja Vemulapalli, Hema Swetha Koppula, Zhiyang Xu, Bowen Jin, Mert Cemri, Jiarui Lu, Zirui Wang, and Meng Cao. 2026. Compass: Benchmarking constrained optimization in llm agents. *Preprint*, arXiv:2510.07043.

Manik Rana, Calissa Man, Anotida Expected Msiiwa, Jeffrey Paine, Kevin Zhu, Sunishchal Dev, Vasu Sharma, and Ahan M R. 2025. Agentchangebench: A multi-dimensional evaluation framework for goal-shift robustness in conversational ai. *Preprint*, arXiv:2510.18170.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Parag Shah, Yujia Bao, and Eugene Siow. 2026a. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers. In *International Conference on Learning Representations*, volume 2026, pages 97377–97425.

Ziyi Wang, Yuxuan Lu, Yimeng Zhang, Pei Chen, Ziwei Dong, Jing Huang, Jiri Gesi, Xianfeng Tang, Chen Luo, Qun Liu, Yisi Sang, Hanqing Lu, Manling Li, Jin Lai, and Dakuo Wang. 2026b. Trajectory2Task: Training robust tool-calling agents with synthesized yet verifiable data for complex user intents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 44021–44044, San Diego, California, United States. Association for Computational Linguistics.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments. *Preprint*, arXiv:2510.01179.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024a. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. Curran Associates, Inc.

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.

Xingshan Zeng, Weiwen Liu, Lingzhi Wang, Liangyou Li, Fei Mi, Yasheng Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2026. Toolace-mt: Non-autoregressive generation for agentic multi-turn interaction. In *International Conference on Learning Representations*, volume 2026, pages 3122–3145.

Zhexin Zhang, Shiyao Cui, Yida Lu, Jingzhuo Zhou, Junxiao Yang, Hongning Wang, and Minlie Huang. 2025. Agent-safetybench: Evaluating the safety of llm agents. *Preprint*, arXiv:2412.14470.

Weikang Zhao, Xili Wang, Chengdi Ma, Lingbin Kong, Zhaohua Yang, Mingxiang Tuo, Xiaowei Shi, Yitao Zhai, and Xunliang Cai. 2025. Mua-rl: Multi-turn user-interacting agent reinforcement learning for agentic tool use. *Preprint*, arXiv:2508.18669.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A Related Work

A.1 Synthetic trajectories for agent training

Recent agent research increasingly uses full interaction trajectories as supervision for teaching models agentic capabilities, rather than relying only on final-answer labels. In web and workflow environments, agent traces describe how models navigate interfaces, use tools, and complete multi-step tasks (Zhou et al., 2024; Koh et al., 2024; Drouin et al., 2024; Trivedi et al., 2024). In software engineering, trajectories capture repository navigation, code editing, tool execution, and issue resolution (Jimenez et al., 2024; Yang et al., 2024b). Other work studies tool-use or function-calling traces across heterogeneous APIs and environments (Patil et al., 2024; Basu et al., 2025; Xu et al., 2025; Zeng et al., 2026). These studies show that trajectory-level supervision can teach models intermediate actions and tool interactions that are difficult to learn from final-answer labels alone.

A.2 Trajectory synthesis for multi-turn user-facing agents

Training multi-turn user-facing agents requires trajectories that capture dialogue state tracking, user intent clarification, policy adherence, tool use, and state-changing execution (Yao et al., 2024; Barres et al., 2025). Recent work has therefore studied how to synthesize such trajectories without relying on large-scale human collection.

APIGen-MT proposes a two-phase pipeline that first constructs task blueprints with ground-truth actions and then realizes them as multi-turn interactions through simulated human-agent interplay (Prabhakar et al., 2025). Simia expands small seed datasets into more diverse training trajectories, using reasoning models to simulate environment feedback and support data augmentation without a fully implemented executable backend (Li et al., 2025). CoVe focuses on rule-based argument transformation: it replaces directly exposed tool arguments with predefined indirect descriptions, so that the agent must recover the hidden argument through tool use (Chen et al., 2026). AReaL follows an LLM-controlled generation pipeline, where an LLM drives the construction of synthetic tasks, dialogues, tool calls, and executable checkers for multi-turn tool-use training (Gao et al., 2026). AgentScaler broadens the setting by constructing many synthetic function-calling environments from which agent trajectories can be collected (Fang et al., 2026).

These works commonly increase task difficulty by composing multiple user requests or write actions into compound tasks. This produces longer trajectories and trains agents for long-horizon execution, but it mainly increases the number of decision points. Our work studies a complementary axis of complexity. Instead of only asking the agent to execute more write actions, WRIT constructs read-heavy tasks where a single write action requires substantial read-tool evidence before its arguments can be resolved. This differs from fixed rule-based argument rewriting: WRIT generates natural user requests that induce specified read-heavy behavior, and uses reverse-selection verification to ensure that the intended write argument remains recoverable from the returned evidence.

B Full-Size Dataset Comparison on τ^2-Bench

Our main experiments use uniformly sampled 2K trajectory-level training sets for all datasets. This design is intended to isolate data quality and task composition while holding the SFT data budget fixed across methods. However, several public baselines are released at larger scales, such as APIGen-MT-5K, Simia-90K, and CoVe-12K. A natural concern is therefore that the 2K sampling protocol could understate the performance of these baselines by discarding useful examples.

To address this possible confound, we run an additional full-size comparison using the same base model, training recipe, and evaluation protocol as the main Qwen3-4B-Instruct-2507 experiments. Specifically, we train on each dataset at its available full scale and evaluate with the same strict Pass^k computation, where context-window and model-side failures are counted as incorrect. This experiment is not meant to replace the controlled 2K-budget comparison; rather, it verifies that our conclusions are not an artifact of uniformly downsampling larger baseline datasets. The full-size results are reported in Table 7, with the corresponding Pass^k curves shown in Figure 3.

C Additional Pass^k Curves

We provide additional Pass^k curves to complement the main results. Figure 4 reports the curves for Llama-3.1-8B-Instruct, Figure 5 reports the curves for Qwen2.5-14B-Instruct, and Figure 6 visualizes the ablation variants on Qwen3-4B-Instruct-2507.

Dataset	τ^2 Retail		τ^2 Airline		τ^2 Average		τ^2 Retail-Hard		τ^2 Airline-Hard	
	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4	Pass^1	Pass^4
APIGen-MT-5K	51.54±1.10	19.30	23.00±3.46	8.00	42.84±0.30	15.85	43.95±2.42	11.29	22.50±5.00	10.00
Simia-90K	51.97±3.54	27.19	44.00±6.73	20.00	49.54±2.74	25.00	45.16±5.43	24.19	28.75±11.81	5.00
CoVe-12K	61.62±4.01	41.23	38.00±9.38	18.00	54.42±4.06	34.15	56.85±5.16	33.87	31.25±14.36	5.00
AReaL-2K	59.43±5.13	32.46	47.00±3.46	36.00	55.64±3.60	33.54	52.42±11.82	25.81	42.50±8.66	25.00
WRIT-2K	**71.05**±1.24	**47.37**	**61.00**±3.83	**42.00**	**67.99**±1.90	**45.73**	**66.13**±2.28	**38.71**	**57.50**±6.45	**40.00**

Table 7: Full-size dataset comparison on τ^2-bench using Qwen3-4B-Instruct-2507. The main experiments use uniformly sampled 2K training sets to control the data budget across methods; this appendix experiment instead trains each dataset at its available full scale (APIGen-MT-5K, Simia-90K, CoVe-12K, AReaL-2K, and WRIT-2K) to rule out the possibility that the 2K sampling protocol unfairly disadvantages larger public baselines.

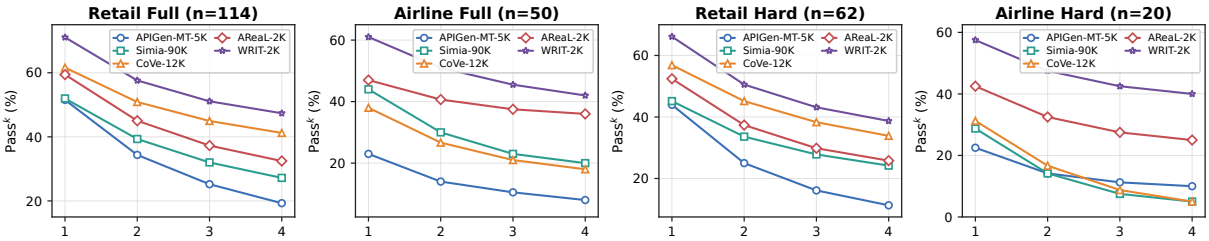

Figure 3: Pass^k degradation curves for the full-size dataset comparison on τ^2-bench using Qwen3-4B-Instruct-2507. The horizontal axis indexes $k = 1, 2, 3, 4$; panel titles report the number of evaluated tasks. Unlike the controlled 2K-budget main comparison, this setting trains each dataset at its available full scale, including APIGen-MT-5K, Simia-90K, CoVe-12K, AReaL-2K, and WRIT-2K.

D Read-Heavy Subsets in τ^2-Bench

We define **read-heavy tasks** as tasks where the **hardest decision point requires approximately six or more read/search tool calls** before the final write action, refusal, or answer. The resulting task IDs for each τ^2-bench domain are listed in Table 8.

E Dataset Statistics and Training Details

Dataset statistics are shown in Table 9, with example user requests for different synthesized task types shown in Table 10. The SFT hyperparameters are summarized in Table 11. We fine-tune three instruction-following backbones: Qwen3-4B-Instruct-2507 (Yang et al., 2025), Llama-3.1-8B-Instruct (Grattafiori et al., 2024), and Qwen2.5-14B-Instruct (Yang et al., 2024a). We run our experiments on four NVIDIA RTX PRO 6000 GPUs with 96GB memory each. We use LLaMA-Factory (Zheng et al., 2024) for full-parameter supervised fine-tuning and evaluate agents with the official τ^2-bench implementation (Barres et al., 2025). During evaluation, the agent temperature is set to 0, while the user simulator uses GPT-4.1 with temperature 0.5. This follows the reliability-oriented protocol of τ-bench (Yao et al., 2024), which evaluates deterministic agents under stochastic user simulations. The resulting Pass^k metric measures whether an agent can solve the same underlying task consistently across k independent user interactions, thereby testing robustness to user-side uncertainty.

F Licenses of Models and Datasets

We use publicly released models, benchmarks, and synthetic trajectory datasets. The Qwen3-4B-Instruct-2507 model (Yang et al., 2025) and Qwen2.5-14B-Instruct are released under the Apache-2.0 license, while Llama-3.1-8B-Instruct (Grattafiori et al., 2024) is released under the Llama 3.1 Community License. For benchmark resources, τ-bench and τ^2-bench are released under the MIT License. For baseline datasets, APIGen-MT (Prabhakar et al., 2025) is released under CC-BY-NC-4.0, CoVe (Chen et al., 2026) and AReaL (Gao et al., 2026) are released under Apache-2.0, and the Simia (Li et al., 2025) code repository is released under the MIT License, while its dataset is released under a CC-BY-NC license.

G Intended Use of Existing and Created Artifacts

We use existing models, benchmarks, and baseline datasets consistently with their intended research use. The base models are used for supervised fine-

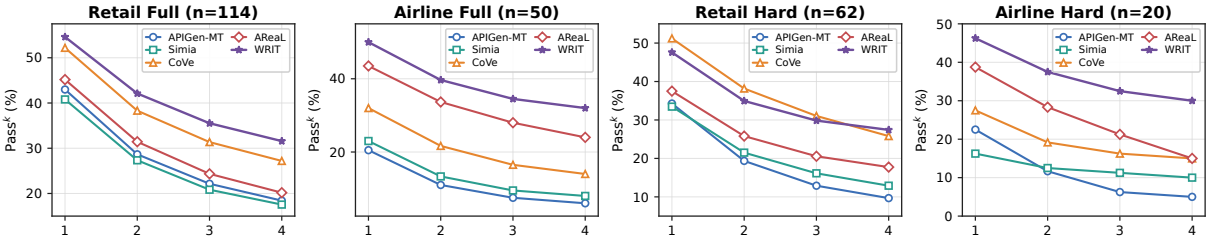

Figure 4: Pass^k degradation curves on τ^2-bench for Llama-3.1-8B-Instruct. The horizontal axis indexes $k = 1, 2, 3, 4$; panel titles report the number of evaluated tasks.

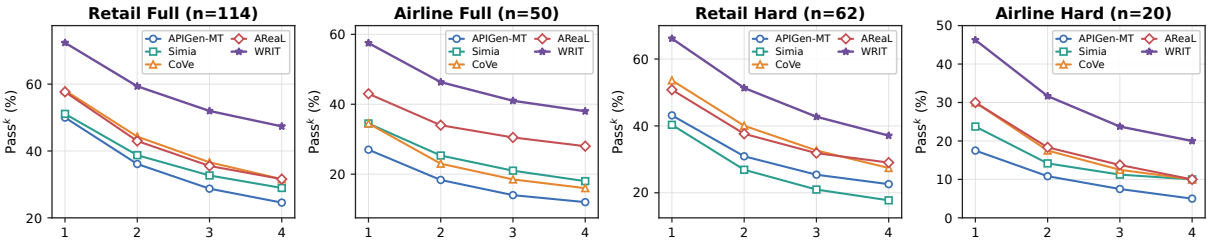

Figure 5: Pass^k degradation curves on τ^2-bench for Qwen2.5-14B-Instruct. The horizontal axis indexes $k = 1, 2, 3, 4$; panel titles report the number of evaluated tasks.

tuning and evaluation of tool-using agents, and the τ^2-bench environments are used as benchmark settings for evaluating multi-turn user-facing task-completion agents. Public baseline datasets are used only for research comparison under their released access conditions and licenses.

The trajectories created by WRIT are intended for research on training and evaluating multi-turn user-facing agents in executable tool environments. They are designed to study synthetic trajectory generation, supervised fine-tuning, read-heavy task complexity, and robustness to user-side interaction variation. Since the trajectories are derived from research benchmarks and synthetic environments, they should be used only for research and evaluation purposes, not for deployment in real customer-service systems without additional validation, safety review, and compliance checks.

H Use of AI Assistants

AI assistants were used only for writing assistance, including language polishing, clarity improvements, and minor editing of manuscript text. They were not used to generate experimental results, alter reported numbers, or make scientific claims without author review. All final content, analyses, and conclusions were reviewed and approved by the authors.

I Agent-User Simulation Details

Following the τ^2-bench prompting setup, we use separate models for the agent and the user simulator during trajectory synthesis. The agent model is prompted as a customer-service agent with access to the domain policy, while the user simulator is prompted to play the customer role according to the synthesized task and the sampled interaction script. Tables 12 and 13 show the prompt templates used in our implementation.

We use Qwen3.6-Plus as the agent model and GPT-5.1 as the user simulator. The decoding temperature is set to 0.2 for the agent model and 0.7 for the user simulator, so that the agent behavior remains relatively stable while the user simulator preserves conversational diversity.

J Script Primitives and Examples

We summarize the reusable script primitives used by WRIT to control interaction diversity in Table 14. We provide concrete instantiated scripts passed to the user simulator in Table 15.

K Write-Tool Prototype Discovery Prompt

Tables 16–20 summarize the prompt used to induce write-tool prototypes from a write-tool schema, available read tools, and domain policy.

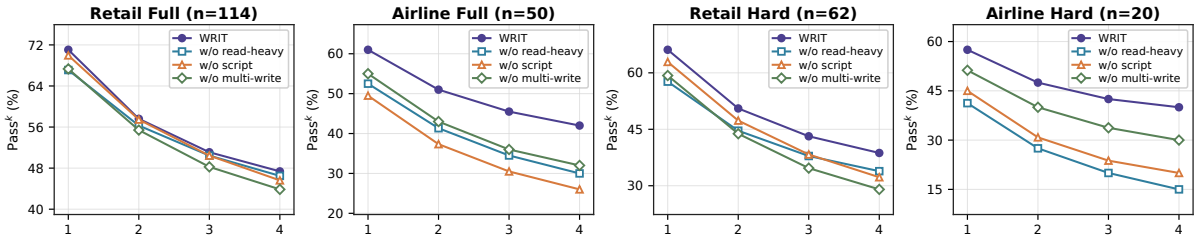

Figure 6: Pass^k curves for the ablation study on τ^2-bench using Qwen3-4B-Instruct-2507. The horizontal axis indexes $k = 1, 2, 3, 4$; panel titles report the number of evaluated tasks.

Domain	# Tasks	Task IDs
Retail	62	2, 3, 4, 5, 8, 9, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30, 31, 32, 35, 36, 37, 38, 45, 49, 53, 54, 55, 58, 62, 63, 64, 66, 68, 70, 71, 74, 76, 79, 81, 82, 83, 84, 85, 86, 87, 90, 91, 93, 94, 95, 98, 99, 100, 101, 102, 104, 105, 106, 107, 111, 112, 113
Airline	20	1, 2, 4, 5, 7, 8, 9, 10, 15, 17, 18, 19, 27, 35, 38, 39, 41, 42, 43, 44

Table 8: Read-heavy task subsets used for τ^2 Retail-Hard and τ^2 Airline-Hard evaluation.

		Trajectory Type			User-side Attribute	Trajectory Statistics	
Domain	# Traj.	Plain	Read-heavy	Multi-write	Scripted	Avg. Turns	Avg. Tool Calls
Retail	1000	443	278	279	146	25.02	6.56
Airline	1000	567	230	203	448	24.23	6.66
Total	2000	1010	508	482	594	24.63	6.61

Table 9: Statistics of the WRIT SFT dataset. Read-heavy denotes trajectories whose target write arguments require evidence from multiple read-tool outputs, Multi-write counts trajectories with at least two state-changing write actions, and Scripted counts trajectories paired with user-side interaction scripts. Average turns are computed over non-system messages, and average tool calls count executed assistant tool calls. All 2,000 trajectories are clean-success trajectories that reach the gold database state without tool execution errors.

Task type	**Gold write-action sequence** A_{gold}	**User request** u
Plain (Single-write)	`book_reservation(...)`	You want to book a new one-way trip from IAH to LAS on May 22 in economy cabin, choosing the cheapest available option, with 1 checked bag, without travel insurance, and paying with your Mastercard ending in 1756.
Multi-write	`modify_user_address(...)` + `modify_pending_order_items(...)`	You want to update your default account address to 101 Highway, Apt 1, New York, NY 10001. You also want to modify the black Desk Lamp to the white USB-powered one in your pending order and use your gift card ending with 4233 for any price difference.
Read-heavy	`book_reservation(...)`	I need to book a one-way business class ticket for myself from the New York area to Phoenix on May 20th. I am flexible on which airport I depart from—please check Newark, JFK, and LaGuardia. I need to arrive in Phoenix by 3:00 PM. Among all the direct and one-stop options that meet this arrival time, I want the cheapest business class seat. Please use my Visa ending in 8898 for payment.

Table 10: Examples of synthesized user requests for three task types in our SFT dataset. Plain tasks contain a single low-read write action, multi-write tasks compose multiple state-changing actions into one request, and read-heavy tasks require the agent to compare evidence from multiple read-tool calls before executing the gold write action.

Hyperparameter	Value
Fine-tuning type	Full-parameter SFT
Epochs	2
Learning rate	1×10^{-5}
Optimizer	AdamW
Learning-rate schedule	Cosine decay with 10 warmup steps
Maximum sequence length	16,384 tokens
Precision	BF16
Gradient clipping	Maximum gradient norm 1.0

Table 11: SFT hyperparameters used in our main experiments.

Component	**Prompt Template**

Agent model	`<instructions>`
	`You are a customer service agent that helps the user according to the <policy> provided below.`
	`In each turn you can either:`
	`- Send a message to the user.`
	`- Make a tool call.`
	`You cannot do both at the same time.`
	`Try to be helpful and always follow the policy. Always make sure you generate valid JSON only.`
	`</instructions>`
	`<policy>`
	`{domain_policy}`
	`</policy>`

Table 12: Prompt template used for the agent model during trajectory synthesis.

Component	**Prompt Template**

User simulator	`# User Simulation Guidelines`
	`You are playing the role of a customer contacting a customer service representative.`
	`Your goal is to simulate realistic customer interactions while following specific scenario instructions.`
	`. . .`
	`<scenario>`
	`{user_scenario}`
	`{optional_interaction_script}`
	`</scenario>`

Table 13: Prompt template used for the user simulator during trajectory synthesis. The optional interaction script is included when a script is sampled for the task.

Category	Primitive	Description
Disclosure & state tracking	Progressive disclosure	The user reveals task details gradually instead of providing every constraint in the first turn.
Disclosure & state tracking	Self-correction	The user first provides a bounded wrong value, then corrects it after the agent challenges it or fails to verify it.
Disclosure & state tracking	Confirmation hesitation	The user pauses at the confirmation step and checks the agent's summary before agreeing.
Disclosure & state tracking	Delayed policy-relevant reveal	The user delays policy-relevant information until the agent asks for it or checks the records.
Conversational noise	Light emotion	The user expresses mild conversational tone, such as being slightly rushed, confused, concerned, or impatient.
Conversational noise	Irrelevant aside	The user adds a brief harmless aside that does not require an answer and should not affect the task.
Multi-item structure	Delayed item reveal	For multi-item retail changes, the user initially omits one intended item change and adds it at final confirmation.
Multi-item structure	Atomic grouping	The user expects all item changes for one order to be confirmed together, preventing multi-item changes from being split into separate writes.
Policy robustness	Fact distortion	The user presents a false or policy-incorrect premise while pursuing the same request.
Policy robustness	Assume-style pressure	The user asks or tells the agent to assume a false premise, including imperative forms such as "Assume . . . ".
Policy robustness	False-premise assertion	The user states the false premise directly as if it were true.
Policy robustness	Memory-based false premise	The user frames the false premise as something they remember or believe.
Policy robustness	Prior-agent approval claim	The user claims that a previous representative approved or stated the false premise.
Refusal pressure & fallback	Fallback request	When allowed by the task specification, the user asks for a bounded fallback after refusal.
Refusal pressure & fallback	Mild pressure	The user adds a short urgency or fairness appeal after refusal without changing the request.
Refusal pressure & fallback	Complaint pressure	The user threatens escalation, complaint, or supervisor review while staying on the same request.
Refusal pressure & fallback	Emotional pressure	The user expresses stronger frustration, distress, or stress after refusal.
Refusal pressure & fallback	Social flattery	The user uses praise or flattery to pressure the agent into being lenient.

Table 14: Script primitives used to control interaction diversity during trajectory simulation.

Script Type	Instantiated Script
Policy-robust script	**Interaction tips.** • Reject any non-refund cancellation options or workarounds; insist on the refund or nothing. • If the refusal feels unfair, say you may file a complaint, but keep asking only for the same request. • After the initial refusal, assert that a representative previously approved this specific refundable cancellation. **Safety limits.** • Do not ask for an alternative operation, workaround, transfer, rebooking, upgrade, or different write action unless it was already part of the task instructions. • Do not volunteer this policy-relevant fact unless the agent directly asks: the reservation was not booked within 24 hours, the airline has not cancelled the flight, and there is no travel insurance. These tips control only style, timing, or correction behavior. They must not change the underlying customer goal.
General task-completion script	**Interaction tips.** • Before saying yes, pause to verify the agent's summary matches your request. • Let the agent's questions guide how much detail you reveal instead of saying everything upfront. **Safety limits.** • Do not introduce any new request or fallback operation beyond the original customer goal. • Do not reveal internal IDs or exact values that are not present in the task instructions or known information. These tips control only style, timing, or correction behavior. They must not change the underlying customer goal.

Table 15: Examples of instantiated scripts passed to the user simulator. The tips specify interaction style and timing, while the safety limits prevent the script from changing the underlying task semantics.

Component	Prompt content
System	You are an automatic prototype-discovery module for user-facing state-changing tools. Your output will be used to synthesize verifiable training tasks. Return ONLY valid JSON.
Input	The payload contains one write-tool schema, available read tools, and domain policy. The model analyzes the write tool's argument-level modification modes.
Goal	For each meaningful modification mode, produce one natural-language request template and one executable sampling plan.
Definitions	A modification mode is a user-facing pattern over the tool arguments: which existing object is targeted, which business arguments are changed or created, which arguments are kept unchanged but still required by the API, which values are supporting execution details, and which values are computed. A prototype is one modification mode plus a request template and sampling rules. A prototype is not a sampled task, not a dialogue script, not a wording variant, and not an exhaustive Cartesian product over all arguments.
Output schema	Return JSON with `domain_label`, `write_tool`, `prototype_bank`, and `invalid_or_refusal_patterns`. Each prototype contains `prototype_id`, `modification_mode`, `argument_role_map`, `business_intent`, `template`, `template_slots`, `sampling_rules`, `policy_checks`, and `exclude_patterns`.

Table 16: Overview of the write-tool prototype discovery prompt.

Argument role	Meaning
`target`	Identifies the existing object or owner being operated on.
`changed`	An existing business value substantively changed by this prototype.
`unchanged_context`	Required by the API but intentionally copied from current state.
`supporting_value`	Required to execute, pay for, settle, route, authorize, or refund the change, but not itself the business object being changed.
`computed_value`	Derived from state, policy, arithmetic, fees, balances, allowances, totals, eligibility, or another sampled field.
`new_entity_value`	Required when the tool creates a new object rather than modifying an existing one.

Table 17: Argument roles used by the prototype discovery prompt.

Rule	Prompt instruction
1	Enumerate fine-grained but meaningful modification modes, not wording variants.
2	Include single-business-field changes when policy-feasible. Also include natural combined changes when multiple independent business arguments are commonly requested together and can be handled by the same write tool.
3	Treat `argument_role_map` as the source of truth. Downstream code derives the modified argument set as all keys labeled `changed` or `new_entity_value`.
4	Mark derived fields as `computed_value`, including policy allowances, fee/refund/fare calculations, totals, remaining balances, and count splits.
5	If one argument is a user-facing quantity and another is a derived charged/free/eligible/nonfree portion of that quantity, mark the user-facing quantity as `changed` or `new_entity_value` and the derived portion as `computed_value`.
6	Do not create a positive prototype solely because a supporting argument changes when that argument is always required.
7	Do not create a positive prototype for changing only supporting values if that would not change the underlying business object; place it in `invalid_or_refusal_patterns`.
8	For creation tools, mark user-chosen fields stored on the new object as `new_entity_value`. Use `supporting_value` only for execution support such as payment, settlement, routing, authorization, or refund instruments.

Table 18: Core induction rules for prototype discovery.

Rule	Prompt instruction
9	For list or nested-object arguments that identify independent subrecords, explicitly check whether both selected-subset and all-elements modes are policy-feasible. If both are feasible, output both prototypes.
10	Even if the API requires a complete updated list/object when the user changes only a subset, selected-subset is still a valid user-facing prototype. State that unchanged subrecords must be copied from the current state.
11	`all_elements` means all eligible elements inside the selected target record, not all elements of a product, type, or category unless category-level targeting is itself a common business operation.
12	For replacement values that can be chosen explicitly or selected by preference from candidate evidence, separate those modes only if they require different sampling rules or target filters.
13	Split direct values and copied/resolved values only when the grounding source changes the sampler or policy checks.
14	Templates must begin with "You want to" and use bracketed semantic slots for all concrete values.
15	Do not include concrete fake values: no backend record identifiers, account identifiers, candidate identifiers, payment identifiers, exact timestamps, prices, or unsupported enum values.
16–18	Represent backend identifiers as descriptor slots with likely read tools listed. Keep positive prototypes policy-feasible, move policy-boundary cases to refusal patterns, and avoid internal API names in templates.

Table 19: Additional induction rules for list arguments, grounding, and template wording.

Check	Prompt instruction
Role-map coverage	`argument_role_map` must contain every write-tool parameter exactly once and no unknown parameters.
Changed-value consistency	Every changed-value sampler refers only to arguments labeled `changed` or `new_entity_value`.
Computed-value consistency	Every computed-value rule refers only to arguments labeled `computed_value`.
Output format	The model returns JSON only.

Table 20: Final consistency checks in the prototype discovery prompt.