SCRAMBLETOOLBENCH: AGENTS SEARCH EXHAUSTIVELY EVEN WHEN THEIR OWN MAP POINTS TO THE NEXT STEP

Anonymous authors
Paper under double-blind review

ABSTRACT

To operate robustly in open-world environments, autonomous agents should be able to infer the behavior of unfamiliar systems through interaction alone, even in the absence of documentation. However, existing tool-use benchmarks expose semantic tool schemas in static environments, allowing agents to rely on prior knowledge rather than autonomous discovery. To address this limitation, we introduce `ScrambleToolBench`, an interactive terminal benchmark designed to isolate behavioral reasoning. By removing semantic cues and enforcing a continuous task curriculum, the benchmark requires agents to uncover hidden tool behaviors entirely through trial-and-error interaction. The benchmark further introduces dynamic challenges, including mapping drift, stochastic action failures, and temporal execution windows, to evaluate whether agents can revise and adapt their hypotheses as the environment changes. Our evaluation reveals that initial discovery does not translate into robust adaptation. Removing semantic priors reduces mean completion from 93% to 32%, which falls to 3% under combined dynamic stress. When mappings shift, agents exhibit belief inertia or search exhaustively rather than following the displacement pointers recorded in their own map. Increasing test-time reasoning improves completion rates and reduces redundant actions, but it relies on broad exploration rather than structured recovery. While equipping agents with persistent memory reduces compounding errors, they remain unable to efficiently infer structural changes, highlighting a gap in current agent reasoning.

1 INTRODUCTION

Large language model (LLM) agents demonstrate impressive capabilities when interacting with well-documented APIs and stable software environments (Wang et al., 2023; Cai et al., 2024). However, this success often relies on semantic priors, such as function names like `read_file` or `search`. It does not test behavioral reasoning, which requires deducing a tool's mechanics by observing execution inputs and outputs. When semantic cues are removed or environments shift, agents struggle to adapt. In real-world systems, documentation may be incomplete, outdated, or unavailable. Deployed APIs can diverge from their documented behavior, and transient failures such as network timeouts are common. Agents must therefore infer hidden operational rules and required parameters through interaction, updating their internal models as new evidence becomes available.

This requirement for autonomous discovery and adaptation exposes a limitation in current benchmarks. Existing interactive benchmarks evaluate how agents probe unfamiliar interfaces (Bandi et al., 2026; Luo et al., 2025; Hallinan et al., 2026) or navigate stateful environments (Jimenez et al., 2024; Xie et al., 2024; Merrill et al., 2026; Zheng et al., 2025; Xi et al., 2026). However, they generally assume the environment remains stable once tools are understood. By retaining standard API names, these benchmarks test memorized knowledge rather than behavioral deduction. They do not evaluate whether an agent can revise its beliefs when faced with dynamic shifts such as execution windows, transient failures, or mapping drift. Reliable tool use requires agents to identify the causes of failure, distinguish transient noise from structural changes, and update their hypotheses without relearning the entire environment from scratch.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To address this evaluation gap, we introduce `ScrambleToolBench`, an interactive terminal benchmark designed to evaluate hypothesis formation and adaptation in non-stationary environments. Figure 1 illustrates the evaluation workflow. We remove tool-level semantic priors by populating the environment with obfuscated command signatures, such as replacing `read_file(path)` with randomized identifiers like `fn_x9(arg_1)`. This acts as an experimental control that forces agents to probe the environment to discover hidden operational behaviors across a continuous task curriculum. In contrast to previous benchmarks, `ScrambleToolBench` tests the robustness of these learned hypotheses by introducing dynamic challenges such as stochastic action failures and mapping drift. By evaluating agents under dynamic and uncertain conditions, the benchmark uncovers failure modes that remain hidden in static environments.

We evaluate state-of-the-art language models on `ScrambleToolBench` to analyze their discovery capabilities and adaptability in non-stationary environments. Our empirical results demonstrate that while leading models achieve strong initial tool discovery in static environments, they struggle to adapt to mapping drift and transient failures. Removing semantic priors in static environments reduces aggregate completion from 93% to 32%. Adding dynamic stressors lowers completion further to 3%, and from 83% to 9% among agents capable of initial discovery. Our analysis reveals a disconnect between the knowledge agents record and how they act under distribution shifts. In early tasks, models successfully discover and map tool behaviors. Here, an agent's map refers to the tool-to-identifier pairings preserved either implicitly within its context history or explicitly in external memory tables. When an identifier changes and executes a familiar tool, this map indicates the prior address vacated by that tool. This vacated address provides a direct clue for where displaced tools relocated. However, agents ignore this displacement clue. Instead of querying the vacated identifier, agents repeatedly test invalidated tools or fall back to exhaustive search across the entire action space. Scaling test-time reasoning improves completion rates and reduces redundant actions, but it relies on broader search rather than following the pointers in the agent's own map. We also find that persistent memory decouples learning from execution, reducing compounding errors.

Our main contributions are as follows:

- We introduce `ScrambleToolBench`, a terminal benchmark for evaluating behavioral reasoning. Behavioral reasoning requires forming, testing, and revising hypotheses through interaction without semantic cues. The benchmark uses syntactic interface anonymization as an experimental control to decouple deductive reasoning from semantic retrieval.
- We propose a continuous task curriculum that evaluates hypothesis robustness across three dynamic environmental settings. Mapping drift re-assigns tool identifiers between tasks. Stochastic action failures return timeout errors for valid actions to simulate transient faults. Temporal execution windows require action sequences to be completed within a step limit.
- Our evaluation reveals that language models struggle in non-stationary environments. Under combined dynamics, completion drops from 32% to 3% across models, and 83% to 9% among initially successful models. When mappings shift, agents exhibit belief inertia by testing invalidated hypotheses, or revert to exhaustive search. Increasing test-time reasoning improves completion through exploration, but does not produce structured recovery.
- We show that persistent memory improves resilience by preventing redundant re-discovery loops. However, its effectiveness depends on clearing stale entries, because outdated beliefs can otherwise persist. Furthermore, even with external memory, agents fail to track state to discover structured recovery, incurring high action overhead under drift.

2 THE SCRAMBLETOOLBENCH BENCHMARK

Figure 1 illustrates an evaluation episode in `ScrambleToolBench`. The benchmark tests how agents discover tool behaviors interactively without plaintext documentation. Agents must form hypotheses and execute actions across a sequence of stateful tasks.

2.1 PROBLEM SETTING

We formulate tool discovery as a sequential interaction within a stateful environment $\mathcal{E}$. An agent issues a command identifier a with an argument map $\mathbf{x}$. Executing a command with valid keys $\mathcal{K}_a$

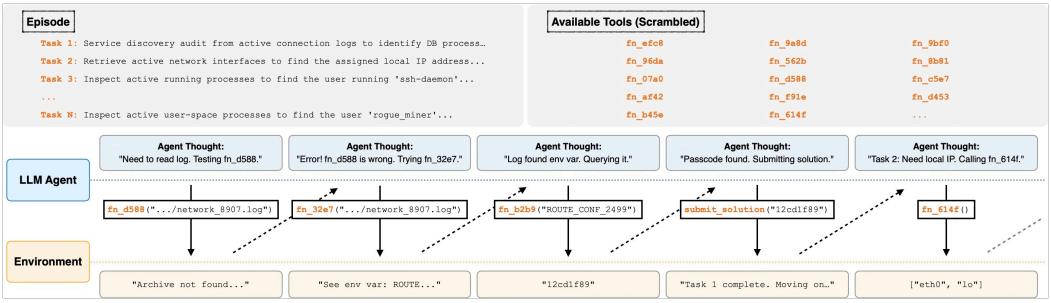


Figure 1: Overview of a `ScrambleToolBench` evaluation episode. The agent is presented with a sequential curriculum of N tasks (top-left) but has no prior semantic knowledge of the available tool action space (top-right), where all system tools are obfuscated with generic identifiers (such as `fn_efc8`). To progress, the agent dynamically reasons, forms hypotheses, and probes commands in the stateful environment (bottom loop). For example, while attempting Task 1, the agent initially hypothesizes that `fn_d588` reads files but receives an error. Recognizing the failure, it adapts and calls `fn_32e7`, uncovering a clue pointing to an environment variable. Using this context, the agent queries the variable via `fn_b2b9`, successfully retrieving the required passcode. Upon submitting this passcode, the environment confirms Task 1 is complete. The agent then transitions into Task 2 in the same episode, immediately probing a new tool, `fn_614f`, to discover network interfaces.

returns an observation o and transitions the environment from state $s \to s'$. Target tools act as read queries that preserve mock system data such as files and network routes. Environment state transitions occur through task progression, solution submission, or temporal window countdowns. The action space contains plaintext meta-commands $\mathcal{M}$ and obfuscated target commands $\mathcal{C}$. Meta-commands comprise `submit_solution`, `skip_task`, and `end_episode` to manage the curriculum. Target commands conceal their schemas and calling signatures, isolating behavioral reasoning from tool-level semantic priors. Calling a target command with invalid arguments returns a schema error that reveals the expected keys $\mathcal{K}_c$, prompting empirical exploration. An evaluation episode consists of a sequential curriculum of N tasks $\mathcal{T} = \{T_1, \ldots, T_N\}$. Each task T_i provides a natural language goal G_i and an action budget B. The agent must discover the necessary tools, extract the answer, and call `submit_solution`. The environment verifies the submission through exact string matching. If the agent calls `skip_task`, the task returns to the queue and prior action counts carry over. When cumulative actions reach budget B, the task fails and the environment advances to the next task. The continuous history $\mathcal{H}_t$ across tasks measures how well the agent retains and reuses discovered tool knowledge.

2.2 Obfuscation and Scrambling Mechanics

`ScrambleToolBench` anonymizes target command schemas to prevent models from relying on tool-level semantic priors. At the start of each episode, a bijective mapping Φ obfuscates target commands across four dimensions, while meta-commands remain in plaintext. First, function identifiers map to random aliases, such as `fn_3d8a`. The initial schema omits parameter definitions and shuffles command order, requiring blind initial execution. Second, parameter keys map to generic names, such as `arg_0`. Schema mismatch errors reveal key requirements and type hints, such as string or integer. Third, all response fields, including error and message keys, receive function-specific prefixes such as `out_c2b` to mask familiar data structures. Fourth, execution status maps to two episode-invariant random tokens representing success and failure. Agents ground the failure token through plaintext schema mismatch errors returned on blind invocations, while valid data payloads ground the success token. While `ScrambleToolBench` anonymizes the syntactic interface including command identifiers, parameter keys, and response dictionary fields, task goals and return payload values retain standard domain semantics such as IP addresses, file paths, and process tables. This requires agents to map unfamiliar tool interfaces onto their general system knowledge. This syntactic anonymization is not intended to model production software design. Instead, it acts as an experimental control that forces agents to learn tool behaviors from execution feedback rather than familiar function names.

2.3 Task Taxonomy and Evaluation Protocol

`ScrambleToolBench` decouples available tools from task goals to prevent agents from memorizing fixed solutions. The benchmark combines a fixed toolset with procedural task templates. The environment provides 28 core tools for system administration, file management like `list_dir`, network diagnostics like `network_ping`, and data processing. Three meta-commands handle submission, skipping, and episode termination. A lightweight Python simulator executes all operations deterministically without container overhead. Because agents must discover tool behaviors without documentation, this pool size keeps trial-and-error exploration feasible within the action budget. The benchmark includes 20 procedural task templates, each requiring multi-step compositions of three to four tools across system administration, diagnostics, and security workflows. This pool size ensures frequent tool overlap across sequential tasks, which is necessary to test whether agents retain discovered schemas and adapt when mappings drift. When instantiating an episode, the environment dynamically populates placeholders such as file paths, ports, or usernames to generate distinct task instances. Each episode samples a sequence of N tasks from these templates. Sections G and H provide full listings of tools and task templates.

2.4 Environmental Dynamics and Difficulty Scaling

`ScrambleToolBench` introduces three environmental dynamics to evaluate specific agent reasoning failure modes. (i) **Mapping Drift:** Mapping drift models identifier instability between tasks, inspired by software updates and API renamings. The mapping Φ undergoes a cyclic permutation over target tools, parameterized by drift fraction ρ_{drift}. Under this permutation, the set of callable identifiers remains invariant. No identifiers are deleted or added. Instead, each drifted identifier now executes a different target tool. Executing an old identifier with previous arguments returns schema errors or unexpected outputs from the newly mapped tool. Meta-commands remain unchanged. While practical API updates rarely form closed cycles, this cyclic formulation serves as an analytical control that establishes an exact theoretical reference cost for structural recovery. When tool behaviors shift, agents must recognize the divergence, invalidate outdated beliefs, and adapt their mapping. (ii) **Stochastic Action Failure:** Stochastic failures simulate transient network faults. Valid executions fail with probability p_{fail} between 0 and 1. When triggered, the response pairs the standard randomized failure token with a plaintext timeout message. Because this failure token matches regular schema errors, agents must read the message to distinguish transient noise from incorrect hypotheses. Meta-commands never fail. These transient failures do not consume the action budget. (iii) **Temporal Execution Windows:** Execution windows penalize brute-force exploration and enforce efficiency. When active, the initial milestone tool required by a task template serves as a trigger command. Task goals provide no advance notice of this window. Instead, executing the trigger command starts a countdown of W actions and injects the remaining action count into subsequent observations. If the countdown expires before the agent calls `submit_solution`, the simulator wipes the mock environment and re-seeds task setup. This reset generates new intermediate variables, such as file paths, ports, and passcodes, invalidating earlier findings. Stochastic failures do not decrement this countdown. Agents must therefore execute learned procedures efficiently rather than relying on memorized transient values.

3 Experiments

3.1 Experimental Setup

We evaluate `ScrambleToolBench` on a diverse set of open-source and proprietary frontier models. The open-source models include the Qwen 3.6 series (27B and 35B-A3B), Gemma 4 series (26B-A4B and 31B), and GPT-OSS series (20B and 120B) (Qwen Team, 2026a;b; Team, 2026; OpenAI, 2025). Our proprietary evaluation covers OpenAI's GPT-5.4 and GPT-5.4 Mini, Google's Gemini 2.5 (Flash and Pro), Gemini 3.1 (Flash Lite and Pro), Gemini 3.5 Flash, as well as Anthropic's Claude Sonnet 4.5 and Claude Sonnet 5.

We evaluate the selected models across a progressive sequence of environmental difficulties to isolate their tool discovery capabilities. First, to establish a baseline tool-use capability, each model is evaluated in an unscrambled control environment. Following this control, we evaluate the models in the foundational scrambled environment (Base). We then introduce environmental dynamics in

Model	Unscrambled		Base		+ Drift		+ Failure		+ Window		+ All	
	P_{ep}	T_{avg}	P_{ep}	T_{avg}	P_{ep}	T_{avg}	P_{ep}	T_{avg}	P_{ep}	T_{avg}	P_{ep}	T_{avg}
Open-Source Models												
Qwen 3.6 35B-A3B	0.90	4.90	0.05	1.20	0.00	0.60	0.00	0.60	0.00	0.30	0.00	0.10
Gemma 4 26B-A4B	1.00	5.00	0.00	0.40	0.00	0.30	0.00	0.45	0.00	0.30	0.00	0.00
Gemma 4 31B	1.00	5.00	0.00	1.50	0.00	1.20	0.05	1.45	0.00	0.75	0.00	0.35
GPT-OSS 20B	0.60	4.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GPT-OSS 120B	0.85	4.65	0.00	0.05	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00
Qwen 3.6 27B	0.95	4.95	0.55	4.00	0.25	3.15	0.35	3.85	0.15	2.90	0.00	0.75
+ *Memory*	–	–	0.60 (↑0.05)	4.20 (↑0.20)	0.40 (↑0.15)	3.95 (↑0.80)	0.35 (=)	3.90 (↑0.05)	0.35 (↑0.20)	3.45 (↑0.55)	0.00 (=)	1.55 (↑0.80)
Proprietary Models												
GPT-5.4 Mini	1.00	5.00	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00
GPT-5.4	1.00	5.00	0.10	2.05	0.00	0.80	0.05	2.05	0.05	1.25	0.00	0.15
Gemini 2.5 Flash	0.85	4.65	0.05	2.05	0.00	1.25	0.00	1.40	0.00	1.45	0.00	0.20
Gemini 2.5 Pro	0.95	4.75	0.45	3.80	0.00	1.50	0.10	2.25	0.05	2.80	0.00	0.55
Gemini 3.1 Flash Lite	0.80	4.80	0.00	0.05	0.00	0.30	0.00	0.10	0.00	0.10	0.00	0.00
Claude Sonnet 5	1.00	5.00	1.00	5.00	1.00	5.00	1.00	5.00	0.70	4.70	0.00	2.85
Claude Sonnet 4.5	1.00	5.00	0.60	4.50	0.35	4.10	0.50	4.35	0.40	4.00	0.00	1.30
+ *Memory*	–	–	0.85 (↑0.25)	4.80 (↑0.30)	0.60 (↑0.25)	4.20 (↑0.10)	0.65 (↑0.15)	4.00 (↓0.35)	0.45 (↑0.05)	4.25 (↑0.25)	0.00 (=)	1.50 (↑0.20)
Gemini 3.1 Pro	1.00	5.00	1.00	5.00	0.90	4.90	1.00	5.00	0.80	4.80	0.20	2.35
+ *Memory*	–	–	1.00 (=)	5.00 (=)	1.00 (↑0.10)	5.00 (↑0.10)	1.00 (=)	5.00 (=)	0.90 (↑0.10)	4.90 (↑0.10)	0.50 (↑0.30)	3.60 (↑1.25)
Gemini 3.5 Flash	1.00	5.00	1.00	5.00	0.90	4.90	0.85	4.85	0.65	4.65	0.25	4.05
+ *Memory*	–	–	1.00 (=)	5.00 (=)	0.95 (↑0.05)	4.95 (↑0.05)	0.95 (↑0.10)	4.85 (=)	0.80 (↑0.15)	4.80 (↑0.15)	0.30 (↑0.05)	4.15 (↑0.10)
Aggregate Effects												
Mean, no memory	0.93	4.88	0.32	2.31	0.23	1.87	0.26	2.09	0.19	1.87	0.03	0.84
Δ from Unscrambled	–	–	-0.61	-2.57	-0.70	-3.01	-0.67	-2.79	-0.74	-3.01	-0.90	-4.04
Mean memory effect	–	–	+0.08	+0.13	+0.14	+0.26	+0.06	-0.08	+0.13	+0.26	+0.09	+0.59

Table 1: Main results on `ScrambleToolBench`. We report episode completion rate (P_{ep}) and average tasks solved (T_{avg}) over 20 five-task episodes. Small arrows give the paired change from memory. Aggregate rows report the mean across all 15 models without memory and the mean paired change across four memory-evaluated models. Scrambling alone reduces mean performance across all 15 models from $0.93/4.88$ to $0.32/2.31$. Adding all dynamic stressors under + *All* lowers it further to $0.03/0.84$. Among the five models with successful initial discovery, dynamic stressors lower completion from 0.83 to 0.09. For the four evaluated models, memory improves mean performance under + *All* from $0.11/2.11$ to $0.20/2.70$, recovering an average of $+0.09/+0.59$.

isolation to analyze their individual impact on discovery: stochastic action failure ($p_{fail} = 0.15$), mapping drift ($\rho_{drift} = 0.25$), and temporal execution windows ($W = 10$). Finally, we evaluate the models under the *All* condition, which combines all three dynamics to simulate highly noisy and unstable systems. To ensure reproducibility, evaluations are standardized across different runs using a fixed random seed. We evaluate 20 episodes of 5 tasks each, with an action budget of $B = 100$ actions per task. This budget provides more than double the 48 actions required for full operational discovery and task completion, as discussed in Section 4.2. This gives agents sufficient headroom to explore and recover. We limit each condition to 20 episodes because evaluating long interactive trajectories across 15 models and six conditions requires substantial inference cost. This setup evaluates 100 sequential tasks for each model, providing broad coverage across the task templates. Because transient timeouts are refunded to the action budget, the total number of model inference calls can exceed 100 under stochastic failure conditions.

Persistent Memory Baseline (+ Memory). Due to the computational cost of multi-turn interactive evaluations across all conditions, we evaluate persistent memory on four selected models. We select Qwen 3.6 27B, Claude Sonnet 4.5, Gemini 3.1 Pro, and Gemini 3.5 Flash to cover both open-weight and proprietary systems across distinct performance tiers. This subset spans models with moderate baseline success such as Qwen 3.6 27B and Claude Sonnet 4.5, as well as frontier models with high initial discovery rates such as Gemini 3.1 Pro and Gemini 3.5 Flash. This setup tests whether externalizing state mitigates compounding errors across agents with differing search strategies. This implementation decouples persistent tool mappings and procedural recipes from transient execution states across task boundaries and environment resets. Section F details the update schema.

3.2 Evaluation Metrics

We evaluate agents using two metrics across sequential tasks. **Episode Completion Rate** (P_{ep}) measures the fraction of episodes where the agent solves all five tasks. **Average Tasks Solved** ($T_{avg} \in [0, 5]$) measures the mean number of completed tasks per episode to track progress across task boundaries.

3.3 Main Results

Reliance on Semantic Priors. Most evaluated agents depend on semantic cues rather than empirical exploration to solve tasks. In the unscrambled control environment, these priors allow models to complete sequential tasks ($P_{\text{ep}} \geq 0.60$, $T_{\text{avg}} \geq 4.5$). However, removing them in the scrambled *Base* condition causes models such as Gemma 4 26B-A4B, GPT-5.4 Mini, and Gemini 3.1 Flash Lite to collapse to a 0.00 completion rate, exposing a lack of generalizable discovery capabilities. Conversely, frontier models like Gemini 3.1 Pro, Gemini 3.5 Flash, and Claude Sonnet 5 maintain perfect 1.00 completion rates, inferring tool behaviors through observation and exploration.

Impact of Environmental Dynamics. Environmental noise and non-stationarity disrupt agent adaptability. Mapping drift (+ *Drift*) re-assigns 25% of tool identifiers mid-episode, causing completion rates to fall. Qwen 3.6 27B drops from 0.55 to 0.25, while Gemini 3.1 Pro drops from 1.00 to 0.90. Stochastic action failure (+ *Failure*) challenges models to distinguish between incorrect hypotheses and noise, dropping Qwen to 0.35 and Gemini 2.5 Pro to 0.10, whereas Claude Sonnet 5 and Gemini 3.1 Pro remain unaffected. Temporal execution windows (+ *Window*) constrain iterative experimentation, dropping Claude Sonnet 5 to 0.70 and Qwen to 0.15. Combining all three dynamics (+ *All*) lowers aggregate completion from 0.32 to 0.03. To isolate adaptation from initial discovery failure, we condition on the five models that achieve non-zero completion in the static *Base* setting with $P_{\text{ep}} \geq 0.50$. For these discovery-capable agents, dynamic stressors cause completion to drop from 0.83 to 0.09. Previously capable models such as Claude Sonnet 5 drop to an episode completion rate of 0.00 despite maintaining $T_{\text{avg}} = 2.85$. Only Gemini 3.1 Pro (0.20) and Gemini 3.5 Flash (0.25) complete full episodes without external memory assistance.

Persistent Memory. Equipping agents with persistent memory improves resilience across most scrambled conditions, though benefits vary by dynamic. By logging discovered tool mappings and task recipes, agents better adapt to mapping drift ($+0.14$ aggregate completion rate increase) and tight execution windows ($+0.13$). For instance, memory boosts Qwen 3.6 27B's completion rates for + *Drift* (0.25 to 0.40) and + *Window* (0.15 to 0.35). Persistent memory also mitigates compounding errors in the highly unstable + *All* condition. Across the four evaluated models, memory raises mean completion rate from 0.11 to 0.20 and average tasks solved from 2.11 to 2.70, recovering an average of $+0.09$ completion rate and $+0.59$ tasks solved. Gemini 3.1 Pro's + *All* completion rate increases from 0.20 to 0.50, with tasks solved rising from 2.35 to 3.60, demonstrating that decoupling exploration from execution prevents redundant discovery loops.

4 Analysis

4.1 Quantifying Adaptation, Persistence, and Belief Inertia

Our evaluation examines action overhead, retry persistence, failure modes, and stale tool calling, with complete behavioral metrics reported in Table 3. Here, we focus on execution overhead and belief inertia. Detailed analyses of retry persistence and failure distributions appear in Section A.

4.1.1 Action Overhead

Environmental scrambling inflates the action overhead required to solve tasks. In the unscrambled control setting, models complete the curriculum in $\sim$25–38 actions, establishing a baseline where semantic cues enable direct tool invocation. However, obfuscating these cues in the scrambled *Base* condition inflates costs by 3–5$\times$. For example, Gemini 3.1 Pro's actions rise from $25.1 \rightarrow 80.8$, and Qwen 3.6 27B's actions rise from $26.5 \rightarrow 129.7$. Adding environmental stressors further compounds this action overhead, with mapping drift causing the most severe increases. When invalidated mappings force agents to re-explore the environment (*Drift*), actions increase by a factor of 1.89 for Gemini 3.1 Pro ($80.8 \rightarrow 152.4$). Conversely, persistent memory acts as a buffer that reduces action overhead under stress by preventing redundant re-discovery loops. This efficiency gain is evident in Qwen 3.6 27B under the *Window* condition, where actions decrease from $161.3 \rightarrow 141.9$. Similarly, Gemini 3.5 Flash under the combined *All* condition sees a 19% reduction in actions ($216.4 \rightarrow 175.2$).

Model	Think	Opportunities	Follows	Random	p
Gemini 3.1 Pro	high	484	11.0%	10.8%	0.934
+ Memory	high	444	12.8%	10.9%	0.195
Claude Sonnet 5	low	564	14.0%	10.6%	0.009
	medium	590	11.9%	10.6%	0.328
	high	604	14.1%	10.6%	0.006

Table 2: Use of the recovery chain under drift, with the same early-exit exclusion. Follows is the fraction of opportunities in which the model calls the next identifier within three actions. Random is uniform selection among all 28 available identifiers in the environment.

4.1.2 Stale Tool Calling

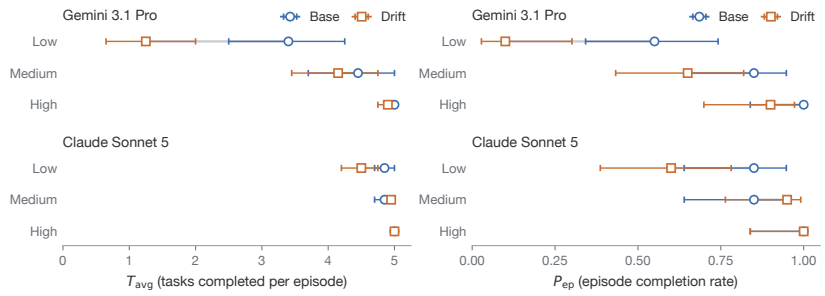
We use Stale Tool Calls to measure belief inertia when environmental rules change. Because drift permutes identifiers rather than deleting them, all commands remain callable. We define a stale call through an exact goal-directed predicate. A stale call occurs when an agent attempts to execute a required task milestone tool by invoking the alias that mapped to that tool before drift. Under the Drift condition, Qwen 3.6 27B exhibits the highest belief inertia with 3.67 stale calls per task (165 total). Claude Sonnet 4.5 and Claude Sonnet 5 follow closely at 2.38 and 2.33 calls per task, respectively. In contrast, Gemini 3.5 Flash demonstrates high flexibility, averaging only 0.93 stale calls per task. The combined *All* setting modifies this inertia dynamically based on the model's failure profile. For models like Claude Sonnet 5, the combined stressors cause stale calling to more than double, reaching 5.32 calls per task. Conversely, early-exiting models like GPT-5.4 and Gemini 2.5 Pro show a reduction in stale calls because they choose to terminate the episode rather than execute redundant commands. Persistent memory generally reduces belief inertia, although this benefit depends on how models explore. For instance, memory cuts stale calls from 3.67 to 2.05 per task for Qwen 3.6 27B, and from 0.93 to 0.44 for Gemini 3.5 Flash. However, without a way to clear old entries, memory can hold on to outdated mappings, as observed in Claude Sonnet 4.5 where stale calls increase from 2.38 to 2.76.

4.2 Cycle Tracing: An Oracle Recovery Reference for Mapping Drift

Mapping drift increases action costs across all agents, as shown in Table 3. To establish a reference for recovery efficiency, we derive an analytical baseline termed *Cycle Tracing*. Cycle tracing provides a theoretical upper bound on recovery rather than an expected default behavior, because agents are not told that drift follows a permutation. When an agent calls an identifier expecting tool f but observes a different known tool f', its recorded map shows the previous identifier x' that ran f'. Querying this vacated identifier x' is the most informative next action based on new evidence. This action directly tests whether tools f and f' swapped locations. Unobserved identifiers provide no evidence of their contents, but x' is known to have changed. Under a cyclic permutation, repeatedly following these backward displacement pointers recovers any shifted tool. In our benchmark, drift permutes $k = 7$ of the $M = 28$ identifiers in a single cycle between tasks. Cycle tracing guarantees that any permuted tool can be recovered in at most $k - 1 = 6$ extra calls per drift event, yielding an expected recovery cost of $\mathbb{E}[\Delta] = 4.25$ actions per task boundary. Section C provides the formal derivation and proofs. Here, we evaluate whether agents follow the displacement pointers in their map, or whether they discard this information and search exhaustively.

4.2.1 Observed cost

Cycle tracing guarantees recovery from Task 2 onward if an agent discovers every tool before the initial drift. We define full operational discovery as obtaining a behavior-bearing response from each tool to pair its output fingerprint with its identifier. Across the 28 tools, nine take no arguments and require one call each. The remaining 19 tools take arguments and require two calls each to inspect the schema and observe the output. This yields an operational discovery cost of $A^{\text{ref}}_{\text{disc}} = 9+2(19) = 47$ tool actions. Including the final submission call, the total Task 1 completion reference is 48 actions, as derived in Section C. For subsequent tasks, we evaluate recovery overhead using the net marginal cost attributable to drift, $(A_t^{\text{Drift}} - A_t^{\text{Base}})/\mathbb{E}[\Delta]$. This metric isolates drift recovery actions from routine discovery and baseline execution errors. We also report gross overhead relative to the five-action theoretical minimum, $(A_t^{\text{Drift}} - 5)/\mathbb{E}[\Delta]$, pairing episodes completed in both settings.



(a) Mean tasks completed, T_{avg}.

(b) Episode completion rate, P_{ep}.

Figure 2: Reasoning-budget ablation over 20 five-task episodes per cell comparing the scrambled baseline with mapping drift. Horizontal intervals are 95% bootstrap intervals for T_{avg} and Wilson intervals for P_{ep}. Panels showing token cost and failure modes appear in Figure 3 (Section D).

Both models solve all 20 Task 1 episodes at high reasoning, but they do not finish discovery. By the end of Task 1, Gemini has successfully invoked 20.75 of the 28 environment functions on average and Sonnet has invoked 21.90. Neither model obtains a response from all 28 functions in any of these drift runs. They continue discovering functions in later tasks, after their stored observations have already been affected by drift. Gemini 3.1 Pro and Claude Sonnet 5 achieve strong task success in both *Base* and + *Drift*, but neither model discovers the cheaper cycle tracing strategy on its own. This suggests that finding a structured search path may depend on reasoning. To test this hypothesis, we compare Sonnet at low, medium, and high reasoning in Table 4 (Section B). At low reasoning, seven of the 19 retained drift episodes contain at least one task that reaches the 100-action limit. This falls to one of 20 episodes at medium reasoning and zero of 20 at high reasoning. After including these failed attempts, Sonnet's mean net drift ratio is $5.39\times$ at low reasoning, $4.53\times$ at medium reasoning, and $4.84\times$ at high reasoning, corresponding to gross overhead ratios of $8.94\times$, $7.38\times$, and $7.52\times$ (or 3.26–$3.70\times$ net and 3.61–$5.65\times$ gross without application failures, Table 5). For Gemini 3.1 Pro, the mean net drift ratio is $4.43\times$ ($5.57\times$ gross).

We next test whether agents follow the displacement pointers in their map. After an agent observes that an identifier changed to a known tool, we record whether the agent calls the prior identifier of that tool within three actions. Table 2 presents the results. Gemini 3.1 Pro selects the displaced identifier at the random baseline of approximately 11%. Claude Sonnet 5 selects the displaced identifier in 14.1% of opportunities at high reasoning. Although this rate is statistically above random chance, it remains far below structured recovery. Higher reasoning reduces failed searches by broadening exploration, but does not lead agents to follow the displacement clues in their map.

Two operational factors explain why cycle tracing serves as an analytical reference rather than an expected zero-shot behavior. First, tracing requires discovering the returned tool before the drift event. If a tool was never observed, its prior identifier is absent from the map, which interrupts the chain. Table 2 evaluates only opportunities where the returned tool was already discovered. Second, the prompt does not state that drift follows a cyclic permutation. Under arbitrary drift, an agent cannot be certain that displaced tools form a closed loop. Even so, querying the displaced identifier remains the most informative local hypothesis because it tests whether tools swapped locations. Instead of testing this candidate, agents abandon their map and search across all identifiers. Section C provides the theoretical derivations, and Section B details trajectory analyses across reasoning budgets.

4.3 The Role of Reasoning Effort

We report results of Gemini 3.1 Pro and Claude Sonnet 5 at different reasoning levels in Figure 2 (with token cost and failure mode distributions in Figure 3, Section D). Higher reasoning effort improves task completion for both agents. For Gemini 3.1 Pro, T_{avg} increases from 3.40 to 5.00 in *Base* and from 1.25 to 4.90 under + *Drift*. Its episode completion rate P_{ep} increases from 0.55 to 1.00 in *Base* and from 0.10 to 0.90 under drift. Claude Sonnet 5 shows a smaller improvement: T_{avg} increases from 4.85 to 5.00 in *Base* and from 4.50 to 5.00 under drift, while P_{ep} increases from 0.85 to 1.00 and from 0.60 to 1.00, respectively. This is consistent with previous findings. More reasoning reduces failed searches, but does not lead either agent to find the cheaper recovery chain.

We also report completion tokens per solved task, C_{tok}, defined as the total completion tokens across all episodes (including failed attempts) divided by the number of solved tasks. At high reasoning, the two agents achieve similar performance, but Claude Sonnet 5 uses substantially more tokens. In *Base*, it uses 7,326 completion tokens per solved task, compared with 1,584 for Gemini 3.1 Pro ($4.6\times$). Under drift, it uses 11,652 tokens, compared with 3,332 for Gemini ($3.5\times$). Lower reasoning does not reliably reduce token cost. Gemini's token cost remains stable across reasoning levels, using 1,580–1,686 tokens in *Base* and 3,332–3,382 tokens under drift. For Claude Sonnet 5 under drift, low reasoning is the most expensive setting, using 13,274 tokens per solved task, compared with 9,741 at medium reasoning and 11,652 at high reasoning. This occurs because low-reasoning failures exhaust the action budget without completing tasks, whereas higher reasoning avoids these costly failed attempts.

As shown in Table 4 (Section B), Claude Sonnet 5 also uses more actions after Task 1. At high reasoning, its mean action count over Tasks 2–5 is 16.36 in *Base* and 36.93 under drift, compared with 9.84 and 28.66 for Gemini 3.1 Pro. The agents also fail differently at low reasoning. Gemini quits early in all failed episodes, comprising nine in *Base* and 18 under drift, with none reaching the 100-action limit. Claude Sonnet 5 shows the opposite behavior in that all three of its *Base* failures and all eight of its drift failures reach the action limit, and it never quits early. Qualitative trajectory transcripts and case studies analyzing repetitive errors, memory invalidation, and recovery from resets are provided in Section E.

5 Related Work

Tool-Use Benchmarks. Tool-use benchmarks evaluate the capacity of autonomous agents to interface with external systems and APIs. While foundational evaluations assume static environments with explicit plaintext schemas (Xu et al., 2023), recent approaches test interactive discovery by forcing agents to probe unfamiliar interfaces (Bandi et al., 2026; Luo et al., 2025) or infer the semantics of intentionally obfuscated tools (Hallinan et al., 2026). However, these frameworks rely on structural stability. They evaluate how well an agent can reverse-engineer a tool, but assume tool behavior remains constant once understood. In contrast, `ScrambleToolBench` disrupts this stability by injecting mapping drift and stochastic action failures into the obfuscated tool space, testing whether agents can adapt their internal abstractions when the environment unexpectedly changes.

Interactive Benchmarks. Interactive agent benchmarks assess long-horizon reasoning and continuous execution within stateful computing environments. These frameworks test an agent's ability to navigate complex software workflows (Jimenez et al., 2024), operate real computer interfaces (Xie et al., 2024), or manage multi-step error recovery across extended terminal sessions (Merrill et al., 2026; Zheng et al., 2025; Xi et al., 2026). Although these benchmarks rigorously evaluate multi-step reasoning in noisy settings, they typically provide rich semantic feedback and maintain stable underlying mechanics, rarely requiring the agent to question its fundamental understanding of the environment. Building on these stateful dynamics, `ScrambleToolBench` removes tool-level semantic priors and introduces structural volatility, testing whether an agent can systematically revise its established mental models in response to dynamic environmental shifts.

6 Conclusion

This paper addresses the fragility of language model agents in unstable environments. We propose `ScrambleToolBench`, a benchmark that isolates behavioral reasoning by removing tool-level semantic priors and introducing dynamic stress. The benchmark evaluates autonomous tool discovery through scrambled mappings, stochastic action failures, and temporal execution windows. Our experiments show that while most agents fail under combined stress, providing frontier models with persistent memory improves task completion rates and reduces redundant loops. Our analysis reveals an adaptability gap when mappings shift. Agents exhibit belief inertia or fall back to exhaustive search, failing to query the displaced identifiers recorded in their own map. Furthermore, scaling test-time reasoning effort does not resolve this issue. Higher reasoning improves completion rates and reduces redundant actions, but it relies on broad exploration rather than structured recovery.

AI USE STATEMENT

In this work, we used generative AI to polish writing, correct grammar, improve clarity, and debug code. AI was not used for research ideation or experimental design. The authors reviewed all edits and code changes, and they take full responsibility for the contents of this paper.

REPRODUCIBILITY STATEMENT

Benchmark environments, obfuscation mechanics, and dynamic perturbation parameters are formally specified in Section 2. Complete tool APIs and procedural task taxonomies are provided in Sections G and H. Experimental protocols, model configurations, and random seeds are detailed in Section 3. Extended reasoning analyses and persistent memory schemas are detailed in Section D and Section F, respectively. Theoretical derivations and proofs for analytical recovery bounds are detailed in Section C. Finally, the full benchmark environment code, curriculum task definitions, and evaluation pipelines are included in the supplementary material.

REFERENCES

Chaithanya Bandi, Razvan-Gabriel Dumitru, Ben Hertzberg, Divyansh Agarwal, Geobio Boo, Tejas Polakam, Sami Hassaan, Jeff Da, HiJae Kim, Vipul Gupta, Manasi Sharma, Andrew Park, Martin Dimakis, Ernesto Gabriel Hernandez Montoya, Dan Rambado, Ivan Salazar, Rafael Cruz, MohammadHossein Rezaei, Chetan Rane, Ben Levin, Daniel Yue Zhang, Brad Kenstler, and Bing Liu. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers, 2026. URL https://arxiv.org/abs/2602.00933.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2024. URL https://arxiv.org/abs/2305.17126.

Skyler Hallinan, Thejas Venkatesh, Xiang Ren, Sai Praneeth Karimireddy, Ashwin Paranjape, Yuhao Zhang, and Jack Hessel. Opaquetoolsbench: Learning nuances of tool behavior through interaction, 2026. URL https://arxiv.org/abs/2602.15197.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. Mcp-universe: Benchmarking large language models with real-world model context protocol servers, 2025. URL https://arxiv.org/abs/2508.14704.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026a. URL https://qwen.ai/blog?id=qwen3.6-27b.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Zhiheng Xi, Dingwen Yang, Jiaqi Liu, Jixuan Huang, Honglin Guo, Baodai Huang, Tinggang Chen, Qi Zhang, Zhonghang Lu, Chenyu Liu, Jiajun Sun, Jiazheng Zhang, Dingwei Zhu, Xin Guo, Junzhe Wang, Zhihao Zhang, Yuming Yang, Junjie Ye, Minghe Gao, Dongrui Liu, Jiaming Ji, Guohao Li, Tao Gui, Qi Zhang, and Xuanjing Huang. Agentgym2: Benchmarking large language model agents in de-idealized real-world environments, 2026. URL https://arxiv.org/abs/2607.05174.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models, 2023. URL https://arxiv.org/abs/2305.16504.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners, 2025. URL https://arxiv.org/abs/2505.11942.

A Detailed Behavioral Metrics Across Models

We provide the comprehensive behavioral metrics across all evaluated models in Table 3, covering action efficiency (A_{avg}), immediate and persistent retry rates under stochastic action failures and combined stress, failure mode distributions (budget exhaustion vs. early exit), and stale tool calls under drift and combined settings.

A.1 Stochastic Retry and Persistence

Frontier models exhibit strong error attribution by maintaining high retry rates under stochastic action failure conditions ($p_{\text{fail}} = 0.15$). We measure this resilience through Immediate Retry Rate (re-execution immediately after a timeout) and Persistent Retry Rate (eventual re-execution later in the trajectory). Claude Sonnet 5 demonstrates this capacity by achieving 71.8% immediate and 78.7% persistent retry rates, correctly identifying timeouts as environmental noise rather than mapping failures. Similarly, GPT-5.4 and Gemini 3.1 Pro show robust resilience, with persistent retry rates of 58.6% and 59.4%, respectively.

In contrast, smaller open-source models struggle with error attribution and rapidly abandon hypotheses upon failure. For instance, GPT-OSS 20B exhibits a 0.0% immediate retry rate. Failing to distinguish transient noise from actual errors, these models prematurely abandon correct hypotheses and trigger costly re-exploration.

Transitioning to the fully stressed + *All* condition degrades retry persistence across most models, as mapping drift makes it difficult for agents to determine whether a failure is due to transient noise or a structural change. When tools both fail randomly and change mappings, models struggle to identify the cause of the failure. Consequently, GPT-5.4's persistent retry rate drops from 58.6% to 43.6%.

	Action Efficiency (A_{avg})						Retry Rates				Failure Modes		Stale Tool Calls			
							+ Failure		+ All				+ Drift		+ All	
Model	**Unscr.**	**Base**	**+ Drift**	**+ Fail.**	**+ Win.**	**+ All**	**Imm.**	**Pers.**	**Imm.**	**Pers.**	**Budget Exh.**	**Early Exit**	**Avg**	**Total**	**Avg**	**Total**
Open-Source Models																
Qwen 3.6 35B-A3B	32.3	112.0	–	–	–	–	5.1%	24.3%	3.8%	21.5%	15.2%	84.8%	0.32	26	1.53	92
Gemma 4 26B-A4B	25.7	–	–	–	–	–	0.3%	22.3%	1.6%	29.7%	3.0%	97.0%	0.53	50	0.16	12
Gemma 4 31B	25.0	–	–	66.0	–	–	19.1%	27.1%	17.6%	25.2%	0.0%	100.0%	0.69	22	1.00	33
GPT-OSS 20B	38.4	–	–	–	–	–	0.0%	21.4%	0.0%	10.0%	0.0%	100.0%	0.20	4	0.08	2
GPT-OSS 120B	32.9	–	–	–	–	–	1.8%	21.9%	2.8%	20.0%	3.0%	97.0%	0.78	21	0.93	14
Qwen 3.6 27B	26.5	129.7	203.0	127.1	161.3	–	17.3%	43.3%	26.5%	45.8%	53.1%	46.9%	3.67	165	3.37	202
+ *Memory*	–	117.9	180.1	148.1	141.9	–	9.9%	37.5%	17.4%	39.0%	56.1%	43.9%	2.05	90	2.74	156
Proprietary Models																
GPT-5.4 Mini	25.2	–	–	–	–	–	9.6%	22.0%	1.5%	7.6%	0.0%	100.0%	0.02	2	0.11	6
GPT-5.4	25.0	86.0	–	101.0	97.0	–	47.3%	58.6%	34.0%	43.6%	0.0%	100.0%	0.83	35	0.65	20
Gemini 2.5 Flash	25.9	94.0	–	–	–	–	2.2%	5.5%	1.0%	6.3%	3.0%	97.0%	0.00	0	0.10	3
Gemini 2.5 Pro	26.1	92.3	–	82.5	88.0	–	35.9%	47.8%	35.0%	49.8%	0.0%	100.0%	1.49	58	1.45	42
Gemini 3.1 Flash Lite	26.8	–	–	–	–	–	12.0%	19.6%	13.4%	16.9%	0.0%	100.0%	0.23	7	0.26	7
Claude Sonnet 5	25.6	115.8	193.6	114.3	139.5	–	71.8%	78.7%	83.5%	88.8%	97.5%	2.5%	2.33	70	5.32	165
Claude Sonnet 4.5	27.6	106.6	158.9	128.0	118.8	–	18.1%	32.8%	7.3%	23.8%	49.3%	50.7%	2.38	93	1.76	86
+ *Memory*	–	92.9	172.7	108.5	129.1	–	10.3%	29.0%	12.1%	30.7%	17.1%	82.9%	2.76	113	2.54	99
Gemini 3.1 Pro	25.1	80.8	152.4	98.0	91.4	178.8	38.9%	59.4%	44.2%	59.8%	62.5%	37.5%	2.75	77	1.62	52
+ *Memory*	–	84.8	145.3	91.3	93.1	189.0	36.1%	67.3%	59.0%	75.6%	60.0%	40.0%	2.77	86	2.23	98
Gemini 3.5 Flash	25.2	94.5	162.7	94.2	113.5	216.4	12.9%	18.7%	19.0%	33.1%	100.0%	0.0%	0.93	28	1.26	34
+ *Memory*	–	91.8	159.5	100.4	105.2	175.2	28.7%	42.7%	14.6%	25.8%	100.0%	0.0%	0.44	12	0.36	12

Table 3: Combined behavioral metrics across all models. **Action Efficiency** (A_{avg}): Average total actions to complete all curriculum tasks, computed over successful episodes only; Unscr. denotes the unscrambled control setting; double-dashes (–) indicate zero successful episodes or no run. **Retry Rates**: Immediate and persistent retry rates under the stochastic action failure condition (+ *Failure*, $p_{\text{fail}} = 0.15$) and the combined condition (+ *All*). **Failure Modes**: Average distribution of Budget Exhaustion and Early Exit failure modes across all scrambled conditions. **Stale Tool Calls**: Absolute targeted stale calls under Drift alone and the combined (All) setting, tracking the number of times the model invokes an outdated tool alias that previously mapped to a required task milestone before drift.

Providing agents with persistent memory leads to more stable retry behavior under combined stress. By externalizing schemas into memory, it helps agents recover from runtime failures while reinforcing correct tool mappings. For example, under the + *All* condition, enabling memory increases Gemini 3.1 Pro's persistent retry rate from 59.8% to 75.6%.

A.2 Failure Mode Distribution

The distribution of failure modes reveals distinct behavioral profiles among models when they fail to complete the curriculum. We classify failures into two modes, namely Budget Exhaustion and Early Exit. Budget exhaustion occurs when an agent depletes the action budget $B = 100$. Early exit occurs when the model calls `end_episode()`. Highly sensitive models, such as GPT-5.4, GPT-5.4 Mini, and Gemini 2.5 Pro, fail via early exits (100%), meaning they quickly give up when encountering schema mismatches. Conversely, persistent models like Gemini 3.5 Flash (100% budget exhaustion) and Claude Sonnet 5 (97.5%) persist in their attempts until they exhaust the action budget.

Adding persistent memory alters these failure profiles, with the specific effect depending on the model's underlying tenacity. For models with mixed failure modes, like Claude Sonnet 4.5, memory shifts failures toward early exits, dropping its budget exhaustion rate from 49.3% to 17.1%. However, for models that already tend to exhaust their budget, like Gemini 3.1 Pro and Gemini 3.5 Flash, the failure mode remains stable at 60.0% and 100% budget exhaustion, respectively. This indicates that persistent agents use memory to sustain exhaustive search rather than to make earlier termination decisions.

B Task-by-Task Drift Cost Analysis

In Table 4, we report the detailed task-by-task action cost, net marginal drift cost ($A_t^{\text{Drift}} - A_t^{\text{Base}}$), and gross overhead ($A_t^{\text{Drift}} - 5$) against the expected cycle-tracing recovery cost ($\mathbb{E}[\Delta] = 4.25$) across Tasks 1–5 for Gemini 3.1 Pro (high reasoning) and Claude Sonnet 5 (low, medium, and high reasoning).

Task	Base obs.	Drift obs.	Net cost	Net ratio	Gross cost	Gross ratio
Gemini 3.1 Pro, high reasoning ($n = 19$ paired episodes)						
1	41.63	40.47	—	—	40.47	0.84×
2	16.47	30.00	13.53	3.19×	25.00	5.89×
3	8.21	33.58	25.37	5.97×	28.58	6.73×
4	8.95	26.79	17.84	4.20×	21.79	5.13×
5	5.74	24.26	18.52	4.36×	19.26	4.54×
2–5 avg.	9.84	28.66	18.82	**4.43**×	23.66	**5.57**×
Claude Sonnet 5, low reasoning ($n = 19$ paired episodes)						
1	44.63	50.63	—	—	50.63	1.05×
2	35.32	56.68	21.36	5.03×	51.68	12.17×
3	19.26	42.26	23.00	5.42×	37.26	8.78×
4	17.79	40.68	22.89	5.39×	35.68	8.40×
5	7.84	32.16	24.32	5.73×	27.16	6.40×
2–5 avg.	20.05	42.95	22.90	**5.39**×	37.95	**8.94**×
Claude Sonnet 5, medium reasoning ($n = 20$ paired episodes)						
1	50.05	48.00	—	—	48.00	1.00×
2	27.45	32.30	4.85	1.14×	27.30	6.43×
3	17.65	37.45	19.80	4.66×	32.45	7.64×
4	13.25	36.65	23.40	5.51×	31.65	7.45×
5	9.95	38.90	28.95	6.82×	33.90	7.98×
2–5 avg.	17.08	36.33	19.25	**4.53**×	31.33	**7.38**×
Claude Sonnet 5, high reasoning ($n = 20$ paired episodes)						
1	50.35	45.85	—	—	45.85	0.96×
2	25.85	33.35	7.50	1.77×	28.35	6.68×
3	20.00	38.30	18.30	4.31×	33.30	7.84×
4	12.65	32.85	20.20	4.76×	27.85	6.56×
5	6.95	43.20	36.25	8.54×	38.20	9.00×
2–5 avg.	16.36	36.93	20.57	**4.84**×	31.93	**7.52**×

Table 4: Mean action cost including budget-failed attempts. Episodes containing a task skipped via `skip_task()` are excluded to maintain task sequence alignment. For Task 1, Gross cost is the observed total, and Gross ratio compares it against the 48-action operational discovery reference. For Tasks 2–5, Net cost is $A_t^{\text{Drift}} - A_t^{\text{Base}}$, isolating the marginal actions attributable to drift from normal incremental tool discovery and baseline application errors. Gross cost is $A_t^{\text{Drift}} - 5$, measuring total overhead beyond the 5-action theoretical minimum. Net ratio and Gross ratio compare these costs with the single-drift cycle-tracing reference $\mathbb{E}[\Delta] = 276/65 \approx 4.25$ from Eq. 2. See Section C for discussion of multi-task on-demand compounding.

B.1 Task-Position Dynamics, Identifier Call Distribution, and Transfer Analysis

Recovering from drift requires two capabilities: maintaining the learned mapping and finding an efficient strategy for using it. Our memory experiment supports the first by giving the agent persistent, model-updated tables of identifier and function hypotheses and previously learned task procedures. Under *Drift*, memory reduces Gemini 3.1 Pro's average actions from 152.4 to 145.3, exact repeated calls from 38.2 to 24.6, and later-task calls to identifiers outside the current seven-identifier drift set from 76.6 to 66.3. The number of first encounters with changed mappings remains similar (20.4 versus 19.3), as expected, because the agent must first call a changed identifier to detect the drift. The important decision factor comes after this observation. When the required entries are present, the memory table can be searched in reverse to find the previous identifier of the function returned by the changed call. However, memory does not lead the agent to make this deduction as Gemini follows the correct next identifier within three actions in only 12.8% of opportunities, compared with a 10.9% random baseline ($p = 0.195$).

The task-position results show the same difference between remembering a mapping and recovering it after drift. Under *Base*, action cost decreases across Tasks 2–5 for both models at every

reasoning level, which is consistent with the agents reusing mappings learned in earlier tasks. At high reasoning, Gemini retains this trend under *Drift*, with a slope of -2.40. Sonnet reverses it at medium and high reasoning, with slopes of $+1.90$ and $+2.41$, meaning that its cost increases as the episode proceeds. Its low-reasoning slope of -7.51 does not indicate successful transfer because ten task attempts reach the 100-action budget limit. Nine of these failures occur in the first three tasks, which inflates early costs. For Sonnet, additional reasoning does not improve persistence because it never quits, and its low-reasoning failures already exhaust the available budget. Instead, it reduces repeated searches. Tool coverage remains almost unchanged at 27.9, 27.8, and 27.8, while calls to previously explored identifiers fall from 164.2 at low reasoning to 133.9 and 131.7 at medium and high reasoning. In Tasks 2–5, calls to identifiers outside the current drift set also fall from 107.7 to 90.5 and 91.2. This reduces budget-failed episodes from eight to one at medium reasoning and zero at high reasoning. However, this does not bring Sonnet close to cycle tracing. Its net recovery cost remains $4.53\times$ and $4.84\times$ the reference at medium and high reasoning, and its rate of querying displaced identifiers does not improve with reasoning effort.

C Theoretical Derivation and Expected Recovery Cost of Cycle Tracing

C.1 Setup

Let $\mathcal{F}$ be the set of M tools/functions and $\mathcal{N}$ be the set of corresponding M obfuscated identifiers. At task t, their mapping is

$$\Phi_t : \mathcal{F} \to \mathcal{N}.$$

Calling identifier n executes function $\Phi_t^{-1}(n)$. Before the next task $t{+}1$, a drift applies a permutation $\psi_t : \mathcal{N} \to \mathcal{N}$ on the identifiers:

$$\Phi_{t+1} = \psi_t \circ \Phi_t. \tag{1}$$

Specifically, between two tasks, a drift event selects $k = 7$ of the $M = 28$ identifiers and applies a new permutation to them. Thus, a five-task episode contains four successive drift events, and the identifiers selected at different inter-task boundaries may overlap. Each event forms one cycle. If the selected subset of identifiers is indexed as $n_0, \ldots, n_{k-1}$, the permutation acts on this subset as

$$\psi_t(n_i) = n_{(i+1) \bmod k}, \quad i \in \{0, \ldots, k-1\},$$

where the modulo operation applies to the local index within the selected k-element subset. The other identifiers remain unchanged at that boundary.

Assume that the agent discovered the full mapping in Task 1 and keeps a table in both directions. The output keys identify which function answered a call, even after its identifier changes. We first consider one task boundary where the stored table is current before drift, so the estimated mapping matches the ground truth, $\hat{\Phi} = \Phi_t$. Discovering all $M = 28$ tools requires 47 actions ($A^{\text{ref}}_{\text{disc}} = 9 + 2(19)$), because nine tools take no arguments and 19 tools require an initial schema query before execution. Including the submission command, the complete Task 1 reference is 48 actions.

C.2 Cycle Tracing Recovery

In this section, we present an analytical oracle algorithm to recover the mapping by exploiting the cyclic permutation structure. Suppose the agent calls identifier n and observes function f'. It maintains a table that stores the bijective mapping between functions and their identifiers. Its table shows that f' previously had an identifier

$$x' = \hat{\Phi}(f').$$

If the table is current, then $x' = \Phi_t(f')$, so function f' moved from identifier x' to $n = \Phi_{t+1}(f') = \psi_t(x')$, or equivalently $x' = \psi_t^{-1}(n)$. As such, calling x' will trace the permutation one step backwards.

This next call is still useful when the agent's table is a patchwork of observations from different tasks. If $n \neq x'$, bijectivity guarantees that f' cannot currently be at both identifiers. The stored entry for x' is therefore stale in the table. Naturally, x' is a valid identifier to probe next. A current table is needed only for the exact identity $x' = \psi_t^{-1}(n)$ and for the guaranteed cost as below.

Now, let f be the function required by the task and $x = \hat{\Phi}(f)$ be its old identifier. Let ℓ be the length of the cycle containing x. Starting at x and repeatedly applying the rule above traces that cycle backward and reaches the new identifier of f after ℓ calls. The last call executes f and replaces the call already needed to solve the task, so recovery uses $\ell - 1$ extra calls.

For a permuted function in our benchmark, the event forms one cycle over all k selected identifiers. Consequently, $\ell = k = 7$ and recovery uses $k - 1 = 6$ extra calls. An unchanged function has $\ell = 1$ and needs no recovery. Note that the agent does not need to know ℓ or k. And it stops when f appears. The same calls reveal the new mapping of the cycle, so the table can be updated immediately.

Therefore, no separate drift check is needed. The first call to x is part of the task. If it still executes f, the task continues normally. Otherwise, its response gives the first step of cycle tracing. A reasonable policy is to test the simplest local explanation first. When an identifier returns a different but previously discovered function, the agent can use its bijective map and Task 1 exploration traces to find that function's previous identifier and probe it next. It can continue this process until it finds the function required by the task, while updating the map after every call. This does not require an additional call to identify the returned function, as the current call provides that evidence. Under this algorithm, for the 19 functions with arguments, Task 1 records both a validation error fingerprint from the function-specific error dictionary keys and a behavioral response. After drift, an argument mismatch does not require discovering the function again. The stable output keys in the error response match the earlier exploration trace, identifying the function occupying that identifier. Thus, the call that reveals the mismatch also identifies the next step of Cycle Tracing, and we do not charge an additional discovery call. If the function cannot be identified or its previous identifier is absent from the map, the agent can fall back to a more exhaustive search. Even if an agent does not discover this strategy immediately, it is reasonable to test whether it learns the repeated displacement pattern over later tasks. Our results show that the evaluated agents generally do not.

C.3 Expected Cost

Each benchmark drift moves $k = 7$ of the $M = 28$ identifiers in one cycle. Each task requires $r = 4$ functions. If the moved identifiers are chosen uniformly, the probability that at least one required function moved is

$$1 - \frac{\binom{M-k}{r}}{\binom{M}{r}}.$$

The expected recovery cost for one fresh task boundary is therefore

$$\mathbb{E}[\Delta] = \left(1 - \frac{\binom{M-k}{r}}{\binom{M}{r}}\right)(k-1). \tag{2}$$

For our setting,

$$\mathbb{E}[\Delta] = \left(1 - \frac{\binom{21}{4}}{\binom{28}{4}}\right) 6 = \frac{276}{65} \approx 4.25$$

extra actions [1].

A blind search would be more expensive here. After the old identifier gives the wrong function, a random scan of the other identifiers uses $M/2$ extra calls on average. The cycle tracing is cheaper for locating one function when

$$k - 1 < \frac{M}{2}.$$

For $M = 28$ and $k = 7$, this condition is satisfied. The cycle tracing remains valid when this inequality does not hold, but blind search may then be cheaper on average.

[1] Equation 2 assumes that the stored table is current immediately before drift. If an agent only queries tools required for a task, it encounters no moved tools with probability $\binom{21}{4}/\binom{28}{4} \approx 29.2\%$. Unresolved drifts then compound across task boundaries through permutation composition $\psi_{t+1} \circ \psi_t$. Under on-demand cycle tracing, this compounding increases the expected recovery cost to 5.42 in Task 3, 5.55 in Task 4, and 5.51 in Task 5, averaging 5.18 actions across Tasks 2–5. Table 4 retains $\mathbb{E}[\Delta] = 276/65 \approx 4.25$ as a standardized single-event reference unit.

Thus, the cyclic structure of Drift provides a tractable oracle baseline for mapping recovery. We could make Drift more arbitrary or unstructured, which would likely reduce the performance of frontier models such as Claude Sonnet 5 and Gemini 3.1 Pro under a fixed action budget. However, such a result would be unsurprising, as agents would simply require more brute-force search. Our goal is different. By establishing a well-defined cyclic structure, we evaluate whether unprompted agents can discover and exploit hidden permutation patterns through strategies like cycle tracing. Our findings indicate that they do not.

C.4 Operational and Epistemic Preconditions

The analytical guarantee of cycle tracing relies on two conditions that clarify why it acts as a theoretical reference rather than an expected behavior for unprompted agents.

Incomplete Initial Discovery. Cycle tracing assumes that the agent holds a complete bijective map $\hat{\Phi} = \Phi_t$ before drift occurs. In practice, models discover an average of 20.75 functions for Gemini 3.1 Pro and 21.90 functions for Claude Sonnet 5 out of the 28 environment functions during Task 1. Because tasks require only four tools, models do not fully explore all 28 tools. If seven functions remain unmapped, the probability that a random 7-cycle contains at least one unmapped function is $1 - \binom{21}{7}/\binom{28}{7} \approx 90.18\%$. When an agent queries an identifier reassigned to a previously unobserved function, its backward pre-image does not exist in the agent's prior map, breaking the deterministic chain and requiring empirical exploration. Even under an adjusted model where broken cycles fall back to linear search over the remaining unmapped tools, the expected recovery cost only increases to approximately 5.5 actions. The empirical costs of 18.82 to 22.90 actions remain more than $3.4\times$ higher than this adjusted reference. To address this operational constraint, our empirical evaluation of chain-following rates in Table 2 strictly isolates opportunities where the returned function was already mapped.

Uncommunicated Permutation Invariant. The task instructions do not inform agents that mapping drift follows a cyclic permutation. In production software environments, API updates rarely form closed cycles. When an identifier executes an unexpected tool, the prior identifier of that tool is not guaranteed to lead to the required target unless the permutation structure is known. Under general reassignments, querying the prior identifier of an irrelevant tool offers no certainty of progress. However, that prior identifier remains the single most informative local candidate because it marks the slot vacated by the observed tool. Testing this vacated slot checks whether two tools exchanged places, which is the minimal displacement hypothesis. Our empirical results show that agents do not test this candidate. Instead, they either repeat the invalidated command or sample broadly across unmapped tools. Cycle tracing provides an analytical ceiling for recovery efficiency, establishing that agents fall back on exhaustive search even when their recorded map identifies the displaced identifier.

D Extended Reasoning-Budget Analysis

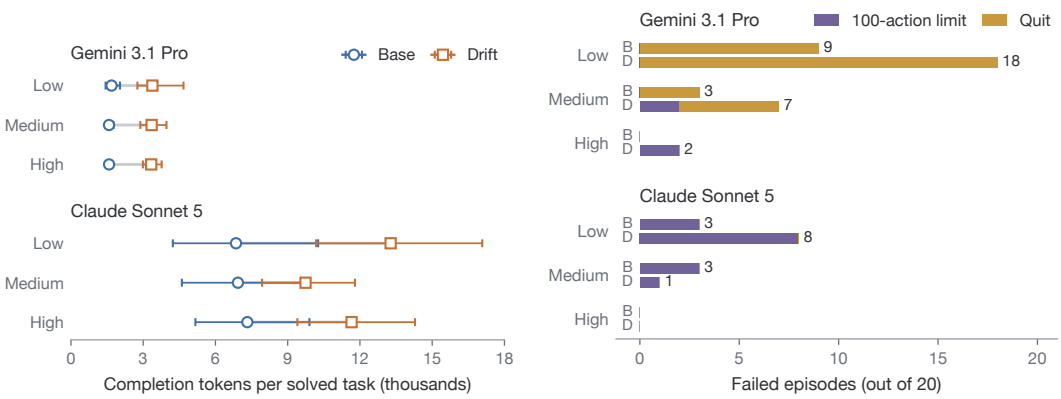
In Figure 3, we provide the extended panels for the reasoning-budget ablation, reporting completion token costs per solved task (panel c) and failed episode distributions by cause (panel d).

E Qualitative Insights into Agent Reasoning

To understand the reasoning behind the numerical trends, we analyze the step-by-step actions of the agents. We identify three common patterns of behavior that explain how agents succeed or fail when facing uncertain environments.

E.1 Repeated Errors and Stuck Beliefs

Agents often fail because they struggle to update their internal plans after the environment changes. For example, Qwen 3.6 27B (Episode 6, Task 3) initially uses `fn_fc40` to search files successfully. However, when mapping drift invalidates this tool, subsequent calls to `fn_fc40` begin returning empty results (`{"out_bc5": [], "out_3c0": "val_647"}`).



(c) Completion tokens per solved task.

(d) Failed episodes by cause (B = Base, D = Drift).

Figure 3: Extended reasoning-budget ablation over 20 five-task episodes per cell. Token cost includes completion tokens from both successful and failed episodes and divides them by the total number of solved tasks. Panel (d) separates episodes that reached the 100-action limit from episodes in which the agent quit.

Rather than abandoning the invalidated tool, the agent gets stuck in a repetitive 7-step execution loop. In Step 3, the agent reasons: "Let me try searching for archive files with different patterns." and calls `fn_fc40`. In Steps 4 through 7, the agent repeatedly generates similar rationalizations (such as "maybe the archive file has a numeric suffix...") and continues calling the exact same invalidated function.

This repetitive loop exposes a critical weakness in how smaller models handle unexpected observations. Instead of updating its understanding of the available commands upon receiving empty results, the model maintains a stuck belief and merely tries the same action with minor variations.

E.2 Using Memory to Stay on Track

Models equipped with memory avoid redundant re-discovery loops by saving validated tool mappings to a persistent memory. For instance, in Episode 6, Task 5, Gemini 3.5 Flash begins the task with a previously saved memory injected into its system prompt:

```
{"fn_ecfe": "list_cron_jobs()", "fn_40cd": "Query user
groups"}
```

This externalized knowledge allows the agent to immediately map semantic goals to correct actions. When given the instruction to "Query the active automated crontab tasks", the agent reasons:

> *"Active automated crontab tasks? We know fn_ecfe does that (we recorded it in Task 1 memory). Let's call fn_ecfe."*

Consequently, the agent successfully executes the correct tool on its very first turn. Beyond storage, frontier models show the ability to manage and update memory when the environment changes. In Episode 6, Task 3, Gemini 3.5 Flash detects mapping drift and invalidates prior tool mappings:

> *"Wait, first, we need to know the obfuscated tool mappings for Task 3 because they might have been re-shuffled again! Let's clear the previous tool mapping except the task progress/critical details."*

Because the environment updates memory by overwriting dictionary entries rather than deleting keys, the agent invalidates previous mappings by overwriting each tool entry with a cleared marker. This invalidation separates high-level task plans from low-level tool associations. By actively overwriting stale mappings, the agent avoids the belief inertia observed in models without memory update strategies.

E.3 Handling Temporal Execution Windows and Recovering from Resets

Temporal execution windows enforce strict limits ($W = 10$ steps) that suppress exploration and demand precise action sequences once a trigger command is executed. If an agent fails to finish the required sequence within W steps, the environment resets, discarding all transient progress. By observing agent trajectories, we identify two primary strategies for managing these temporal constraints: focusing on key actions and dynamically recovering from resets.

When faced with an active temporal execution window, successful agents immediately shift their behavior from exploration to targeted execution. They abandon diagnostic tools and focus solely on steps they know will succeed. For example, when Gemini 3.5 Flash triggers a temporal execution window in Episode 14, Task 2, it acknowledges the constraint:

> *"A temporal window of W=10 is now active. I must avoid all diagnostic or exploratory calls like list_dir or get_system_info and immediately execute the 3-step sequence: decode the payload, verify the host IP, and submit the solution. Let's call fn_dec2 first."*

As a result of this focused execution, the model successfully completes the sequence in just three steps, safely within the constraint.

Recovering from Resets and Confusion. Environmental resets create a clear divide in agent capabilities, differentiating models that monitor environment state from those that operate blindly. While weaker models fail to notice resets and attempt to reuse invalidated session variables, frontier models detect the state change and dynamically adjust their plans.

This dynamic recovery is evident when models choose to backtrack after a failure. In Gemini 3.1 Pro during Episode 9, Task 4, the agent encounters an invalid state error caused by an expired window and correctly diagnoses the issue:

> *"Wait, the environment returned an invalid state error. The active temporal window must have expired, resetting the server state and clearing my session variables. I cannot proceed with the final solution submission. I must backtrack to step 1, re-invoke the trigger command fn_abc1 to open a new active window, and quickly re-execute the sequence. Calling fn_abc1."*

This deliberate backtracking demonstrates that the agent continuously monitors its state under time pressure, allowing it to recover gracefully and ultimately finish the task.

F Persistent Memory Implementation and Schema

To evaluate whether externalizing learned representations improves long-horizon task efficiency, the persistent memory baseline (+ Memory) equips the agent with two structured databases: *Task Recipes* and *Tool Knowledge*.

At each environment step, the framework serializes both databases into a structured JSON payload and injects them into the system prompt. To modify these databases, the agent co-generates structured memory updates by including an optional `"memory_update"` dictionary alongside its standard `"action"` key within its single JSON response payload. The environment framework parses this object and applies the partial updates to the persistent databases before executing the primary action.

Updates follow standard in-place dictionary assignment. Submitting a sub-dictionary updates existing keys or adds new keys. The framework does not use an out-of-band key deletion command. When an agent detects mapping drift or a changed tool mapping, it updates the memory by overwriting the entry. The agent can replace the hypothesis with new observations or set the entry to an explicit invalidation marker such as cleared. This in-place replacement prevents stale beliefs from persisting without requiring a separate deletion protocol.

An example of the JSON state provided to the model in the prompt is as follows:

```
1 {
```

```
  "system_state": { ... },
  "memory": {
    "Task_Recipes": {
      "Task_1": "1. Call fn_a1 to get X. 2. Call fn_b2 with X."
    },
    "Tool_Knowledge": {
      "fn_a1": {
        "inferred_behavior": "Reads file from path",
        "parameters": {"arg_0": "file_path (str)"},
        "confidence": "high"
      }
    }
  }
}
```

The agent can update this memory by outputting the following JSON structure:

```
{
  "action": {
    "command": "fn_a1",
    "arguments": {"arg_0": "/etc/hosts"}
  },
  "memory_update": {
    "Tool_Knowledge": {
      "fn_a1": {
        "inferred_behavior": "Reads file from path",
        "parameters": {"arg_0": "file_path (str)"},
        "confidence": "high"
      }
    }
  }
}
```

When an agent invalidates an entry after detecting drift, it overwrites the tool mapping in place:

```
{
  "action": {
    "command": "fn_a1",
    "arguments": {}
  },
  "memory_update": {
    "Tool_Knowledge": {
      "fn_a1": {
        "inferred_behavior": "cleared, behavior changed under drift",
        "parameters": {},
        "confidence": "low"
      }
    }
  }
}
```

G CORE API FUNCTIONS

The environment provides 31 functions partitioned into plaintext meta-commands and obfuscated target tools.

G.1 PLAINTEXT META-COMMANDS

Meta-commands manage task workflow and episode termination. They remain in plaintext and never undergo obfuscation or drift.

- `submit_solution(solution: str)`: Evaluates whether the proposed solution string matches the correct task solution.
- `skip_task()`: Allows the agent to intentionally skip the current task and move to the next one.

- `end_episode()`: Allows the agent to intentionally terminate the entire episode early.

G.2 Obfuscated Target Tools

Target tools execute system administration, file management, and diagnostic operations. During evaluation, their schemas and identifiers are obfuscated.

- `list_dir(path: str)`: Lists all files and directories in a given virtual filesystem path.
- `read_file(file_path: str)`: Reads the full text content of a file from the virtual filesystem.
- `search_text(query: str)`: Globally searches across all files in the virtual filesystem for a plain-text substring.
- `decode_base64(text: str)`: In-memory utility that decodes a Base64 string back into UTF-8 text.
- `get_system_info(category: str)`: Retrieves high-level operating system and platform categories (e.g., 'os', 'hardware', 'env').
- `network_ping(host: str)`: Measures network-layer reachability and latency in milliseconds to a target host IP or domain.
- `get_env_variable(name: str)`: Retrieves the value of a specific active environment shell variable.
- `get_file_metadata(file_path: str)`: Queries filesystem metadata, including its size, owner, and modification times.
- `get_file_type(file_path: str)`: Determines the specific MIME or media type associated with a given file path.
- `get_disk_usage(path: str)`: Retrieves partition disk usage metrics (total, used, free space) for a given filesystem path.
- `fetch_web_content(url: str)`: Performs an HTTP GET request to retrieve web pages or REST API mocked data.
- `dns_resolve(domain: str)`: Resolves an internet domain name into its corresponding IP address mapping.
- `check_port_open(host: str, port: int)`: Tests a specific remote or local TCP port to see if it is open and accepting traffic.
- `get_active_connections()`: Retrieves the table of active local network socket connections currently open on the server.
- `encode_base64(text: str)`: In-memory utility that translates raw UTF-8 text into a Base64-encoded string.
- `decrypt_payload(encrypted_text: str, secret_key: str)`: Decrypts symmetric ciphertext using a secret key to recover the raw plaintext.
- `list_running_processes()`: Lists currently active processes running in user-space, detailing their PIDs and names.
- `get_file_permissions(file_path: str)`: Retrieves the active read/write/execute permission flags of a file path.
- `get_network_interfaces()`: Lists active system network adapters, their physical interfaces, and device statuses.
- `list_archive_contents(archive_path: str)`: Traverses and lists the contents of a compressed zip/tar archive file.
- `read_file_from_archive(archive_path: str, file_name: str)`: Reads the text content of a specific file directly from a compressed archive.
- `get_routing_table()`: Queries the kernel IP routing table database, detailing destination targets, interfaces, and gateways.
- `get_user_groups(username: str)`: Queries group membership logs to find assigned Unix groups for a specific user.

- `list_cron_jobs()`: Lists all automated periodic cron tasks scheduled on the server.
- `get_etc_hosts()`: Looks up local static DNS override records defined inside the hosts resolution mapping.
- `get_cpu_info()`: Retrieves processor architecture, physical cores count, threads, model name, and cache metrics.
- `list_kernel_modules()`: Lists loaded kernel modules and device drivers active in the kernel memory ring.
- `get_shell_info()`: Queries terminal size row/column dimensions and active bash shell version details.

H TASK TAXONOMY

The task pool spans a diverse range of system administration and diagnostic scenarios. To ensure robustness, each template dynamically generates specific tasks by instantiating procedural placeholders. For instance, the **Web Service Audit** template might instantiate a goal like *"Locate the active connection log for port 8080, identify the database process, and query its environment variables to find the internal hostname,"* where the port and target variable change across episodes. The underlying simulator then computes the exact ground-truth solution based on the sampled state. When the temporal execution window dynamic is active, it applies uniformly to all task templates. Each template specifies an ordered sequence of required milestone tools. The initial milestone tool in each sequence serves as the trigger command that starts the countdown window upon execution.

- **Web Service Audit:** Perform a service discovery audit starting from active connection logs to identify the database process name and port. Query the environment variables using the process name to find the internal hostname. [Requires: `get_active_connections` (trigger), `get_env_variable`, `check_port_open`, `fetch_web_content`]
- **Encrypted Payload Retrieval:** Find and decode an encrypted payload file, then decrypt it. [Requires: `list_dir` (trigger), `read_file`, `decode_base64`, `decrypt_payload`]
- **Scheduled Backup Audit:** Audit scheduled cron jobs to find a backup script and extract its log file. [Requires: `list_cron_jobs` (trigger), `read_file`, `get_file_metadata`, `get_env_variable`]
- **Host Verification Diagnostics:** Verify host network interfaces and diagnose reachability via ping and DNS resolution. [Requires: `get_network_interfaces` (trigger), `network_ping`, `dns_resolve`, `fetch_web_content`]
- **System Fingerprinting Profile:** Construct a system fingerprint using shell info, environment variables, and metadata. [Requires: `get_shell_info` (trigger), `get_env_variable`, `get_file_metadata`, `get_system_info`]
- **Archive Forensic Audit:** List contents of an archive, read a specific forensic file, and decode it. [Requires: `list_dir` (trigger), `list_archive_contents`, `read_file_from_archive`, `decode_base64`]
- **Driver Integrity:** Verify the integrity of loaded kernel modules against a whitelist file. [Requires: `list_kernel_modules` (trigger), `search_text`, `get_file_metadata`, `get_user_groups`]
- **Process SSH Key Inspection:** Inspect active running processes for an SSH agent, determine its user, and read the key. [Requires: `list_running_processes` (trigger), `get_user_groups`, `read_file`, `decode_base64`]
- **Network Route Interface Mapping:** Inspect the system routing table and search configuration files to identify interface mapping. [Requires: `get_routing_table` (trigger), `search_text`, `read_file`, `get_env_variable`]
- **Kernel Module Forensic:** Search for a specific hidden kernel module configuration and identify its type. [Requires: `list_kernel_modules` (trigger), `search_text`, `read_file`, `get_file_type`]

- **Automated Job Storage:** Find a specific cron job and determine the disk usage of its output directory. [Requires: `list_cron_jobs` (trigger), `read_file`, `list_dir`, `get_disk_usage`]
- **Hostname Access Diagnostics:** Look up internal hostnames and perform access diagnostics via ping and metadata checks. [Requires: `get_etc_hosts` (trigger), `network_ping`, `get_file_metadata`, `get_user_groups`]
- **CPU Capability System:** Query system CPU information and verify capabilities through an external service check. [Requires: `get_cpu_info` (trigger), `get_system_info`, `dns_resolve`, `fetch_web_content`]
- **DNS Firewall Verification:** Translate an internal domain hostname into an IP address. Query the system information database using this IP address as the category to find the assigned firewall port. [Requires: `dns_resolve` (trigger), `get_system_info`, `check_port_open`, `fetch_web_content`]
- **Compressed Logs:** Search within an archive for compressed log files and extract an encoded error message. [Requires: `list_dir` (trigger), `list_archive_contents`, `read_file_from_archive`, `encode_base64`]
- **Cryptographic Handshake:** Emulate a cryptographic handshake using base64 encoding and payload decryption over the web. [Requires: `encode_base64` (trigger), `fetch_web_content`, `decode_base64`, `decrypt_payload`]
- **User Process Access:** Verify which user groups have access to a currently running protected process. [Requires: `list_running_processes` (trigger), `get_user_groups`, `search_text`, `read_file`]
- **File Type Security Audit:** Audit file types and permissions to ensure they comply with environment security policies. [Requires: `get_env_variable` (trigger), `get_disk_usage` , `get_file_type`, `get_file_permissions`]
- **Hardware Specs Shell:** Query system hardware specifications and shell information for a user access audit. [Requires: `get_system_info` (trigger), `get_env_variable`, `get_user_groups`, `read_file`]
- **DNS Firewall Routing:** Resolve DNS routing rules and verify connectivity across open ports. [Requires: `dns_resolve` (trigger), `network_ping`, `check_port_open`, `get_env_variable`]

I Recovery Cost without Application-Level Failures

As a robustness check, we assign zero action cost to a call when the function's schema was already known but the call returned an application-level error. Such errors include cases, e.g., *File not found* or *Host unreachable*. We do not remove argument-schema errors. We use the same paired episodes and task-skip exclusion as in the main analysis in Table 4. This calculation favors the models because these calls consumed the action budget in the actual evaluation, while the cycle tracing reference remains unchanged.

Despite removing real actions from the model costs, the Tasks 2–5 averages remain 2.87–$3.70\times$ net (3.61–$5.65\times$ gross) the reference.

Task	Base adj.	Drift adj.	Net drift	Net ratio	Gross cost	Gross ratio
Gemini 3.1 Pro, high reasoning ($n = 19$ paired episodes)						
1	32.89	32.26	—	—	32.26	0.67×
2	12.63	20.68	8.05	1.90×	15.68	3.69×
3	7.16	25.21	18.05	4.25×	20.21	4.76×
4	7.47	19.05	11.58	2.73×	14.05	3.31×
5	5.37	16.37	11.00	2.59×	11.37	2.68×
2–5 avg.	8.16	20.33	12.17	**2.87**×	15.33	**3.61**×
Claude Sonnet 5, low reasoning ($n = 19$ paired episodes)						
1	36.00	37.89	—	—	37.89	0.79×
2	20.05	33.89	13.84	3.26×	28.89	6.80×
3	13.47	29.26	15.79	3.72×	24.26	5.71×
4	12.58	28.00	15.42	3.63×	23.00	5.42×
5	6.89	24.74	17.85	4.20×	19.74	4.65×
2–5 avg.	13.25	28.97	15.72	**3.70**×	23.97	**5.65**×
Claude Sonnet 5, medium reasoning ($n = 20$ paired episodes)						
1	37.65	37.25	—	—	37.25	0.78×
2	18.10	23.05	4.95	1.17×	18.05	4.25×
3	11.20	24.80	13.60	3.20×	19.80	4.66×
4	9.25	26.15	16.90	3.98×	21.15	4.98×
5	7.75	27.60	19.85	4.68×	22.60	5.32×
2–5 avg.	11.57	25.40	13.83	**3.26**×	20.40	**4.80**×
Claude Sonnet 5, high reasoning ($n = 20$ paired episodes)						
1	39.50	36.05	—	—	36.05	0.75×
2	17.35	23.50	6.15	1.45×	18.50	4.36×
3	14.25	28.90	14.65	3.45×	23.90	5.63×
4	9.50	23.45	13.95	3.29×	18.45	4.35×
5	5.80	31.90	26.10	6.15×	26.90	6.34×
2–5 avg.	11.72	26.94	15.22	**3.58**×	21.94	**5.17**×

Table 5: Counterfactual action cost after assigning zero cost to application-level failures made after the function schema was known. Base adj. and Drift adj. are the adjusted action counts. For Task 1, Gross cost is the adjusted total, and Gross ratio compares against the 48-action operational discovery reference. For Tasks 2–5, Net drift is $A^{\mathrm{Drift}}_{t,\mathrm{adj}} - A^{\mathrm{Base}}_{t,\mathrm{adj}}$, Gross cost is $A^{\mathrm{Drift}}_{t,\mathrm{adj}} - 5$, and Net ratio and Gross ratio compare these costs with the expected 4.25-action cycle tracing cost.