AGENTREWIND: RECOVERABLE EXECUTION FOR LONG-HORIZON LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Many real-world tasks require LLM agents to interact with their environments over long execution horizons. Errors that occur early in execution may propagate through both the agent context and environment state, and their effects may be difficult to reverse through subsequent actions. Existing methods mainly seek to reduce such errors through plan refinement and safety checks but provide little support after errors occur. To enable recovery during long-horizon execution, we present **AgentRewind**, a runtime recovery framework that records aligned checkpoints of the agent context and controlled environment, allowing agents to return to an earlier state and resume execution with information from previous attempts. We also construct **MettleBench**, a benchmark for evaluating task completion and partial progress on long-horizon engineering assignments containing a series of related requirements. Experiments across tasks, multiple models, execution strategies, and agent harnesses show that AgentRewind improves task success rate and average checklist progress over the compared baselines.

INTRODUCTION

LLM agents solve tasks through tool-mediated interaction with their environments. Unlike non-agentic LLMs, they observe environmental states, invoke tools, and adjust subsequent actions based on feedback (Yao et al., 2023; Liu et al., 2024). They are increasingly applied to long-horizon tasks such as iterative scientific experimentation, cross-application workflows, and repository-level software engineering (Kon et al., 2026; Li et al., 2026; Jimenez et al., 2024). Improving their performance on such tasks is therefore an important research objective.

However, operating over long horizons presents distinct challenges for LLM agents. For example, an agent may formulate an incorrect plan early in execution, causing subsequent actions to proceed in the wrong direction (Valmeekam et al., 2023; Liu et al., 2024; Wang et al., 2026b). During execution, an agent may also delete critical files, corrupt configurations, or contaminate database states, forcing subsequent actions to proceed in a degraded environment and potentially preventing task completion (Ruan et al., 2024; Trivedi et al., 2024). Because long-horizon tasks involve more execution steps, agents are more likely to make such errors along the execution trajectory and therefore face a higher risk of task failure.

Existing approaches primarily seek to improve agent performance on long-horizon tasks through plan refinement and safety checks. One line of work generates and revises plans before execution to identify more effective execution paths (Erdogan et al., 2025; Zhou et al., 2024). Another line of work applies safety checks during execution to detect unsafe behavior (Wang et al., 2026a; Xiang et al., 2025). However, even with a low probability of error at each step, cumulative failure risk grows with trajectory length. These methods do not adequately address recovery after an error has disrupted execution.

We therefore introduce AgentRewind, a runtime recovery framework that allows agents to rewind execution to an earlier state. When the agent determines that it is unlikely to make further progress along the current trajectory, it can select an earlier checkpoint based on the recorded trajectory. AgentRewind then restores the agent context and environment to the states recorded at that checkpoint, allowing the agent to continue execution from there. It also retains a summary of the previous attempt as rewind memory to guide the agent's subsequent decisions. Through these capabilities,

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.

AgentRewind enables agents to recover from errors during task execution, improving their performance on long-horizon tasks. Our main contributions are summarized as follows:

- We develop AgentRewind, a system that can return both the agent context and the environment state to a selected checkpoint. This allows the agent to continue from an earlier checkpoint and explore a different trajectory when errors prevent further progress.
- We construct MettleBench, a benchmark derived from real-world engineering resources, to evaluate task completion and checklist progress on assignments containing a series of related requirements.
- We evaluate AgentRewind across multiple base models, execution strategies, and agent harnesses, and show that it improves task success rate and average checklist progress in long-horizon settings.

Related Work

Long-Horizon Agents

Long-horizon agents have been studied across embodied, scientific, web-based, software engineering, and database settings. ALFWorld, ScienceWorld, and WebShop evaluate long-horizon interaction in embodied, scientific, and web environments (Shridhar et al., 2021; Wang et al., 2022; Yao et al., 2022). InterCode and SWE-bench study interactive programming and repository-level software engineering (Yang et al., 2023; Jimenez et al., 2024; Yang et al., 2024), while Spider 2.0, CoSQL, and BIRD-INTERACT focus on multi-turn database tasks (Lei et al., 2025; Yu et al., 2019; Huo et al., 2026). More recent benchmarks, including TheAgentCompany and Odysseys, extend evaluation to realistic computer-use tasks and show that even strong models often fail to complete long-horizon tasks (Xu et al., 2025; Jang et al., 2026).

Agent Reliability

Prior work on agent reliability improves execution through better action generation and safety monitoring. Methods such as ReAct, SayCan, and LATS guide agent behavior using reasoning, planning, search, and environment models (Yao et al., 2023; Ichter et al., 2023; Huang et al., 2023; Erdogan et al., 2025; Zhou et al., 2024; Koh et al., 2025; Chae et al., 2025). Safety-oriented methods monitor agent execution and identify risky behavior (Wang et al., 2026a; Xiang et al., 2025; Liu et al., 2026).

Another line of work uses information from previous attempts to improve later behavior. Self-Refine, Reflexion, and ExpeL use feedback from prior attempts to revise subsequent decisions (Madaan et al., 2023; Shinn et al., 2023; Zhao et al., 2024). Related failure-analysis methods diagnose the causes of failure from completed execution trajectories (Zhu et al., 2025; Barke et al., 2026). These methods use information from previous executions to improve later behavior, but provide little support for recovery after an error occurs during execution.

Rollback for LLM Agents

Several studies have explored rollback for LLM agents. GA-Rollback (Li et al., 2025) performs stepwise recovery in replayable environments, whereas WebRollback (Zhang et al., 2026) specializes rollback to browser navigation. DART (Yang et al., 2026b) selects semantically valid restore points for structured tool agents but relies on explicit control flow and recovery boundaries, limiting its applicability to open-ended agent trajectories.

System-level checkpointing tools such as DMTCP and CRIU provide general mechanisms for restoring process state (Ansel et al., 2009; CRIU Project, 2026). DeltaBox extends these mechanisms to agent sandboxes, enabling efficient rollback of filesystem and process state (Dong et al., 2026). However, these systems primarily provide low-level rollback mechanisms rather than an agent-level recovery process for long-horizon execution. AgentRewind combines coordinated restoration of the agent context and environment state with rewind memory, allowing the agent to return to an earlier checkpoint and continue execution with information from previous attempts.

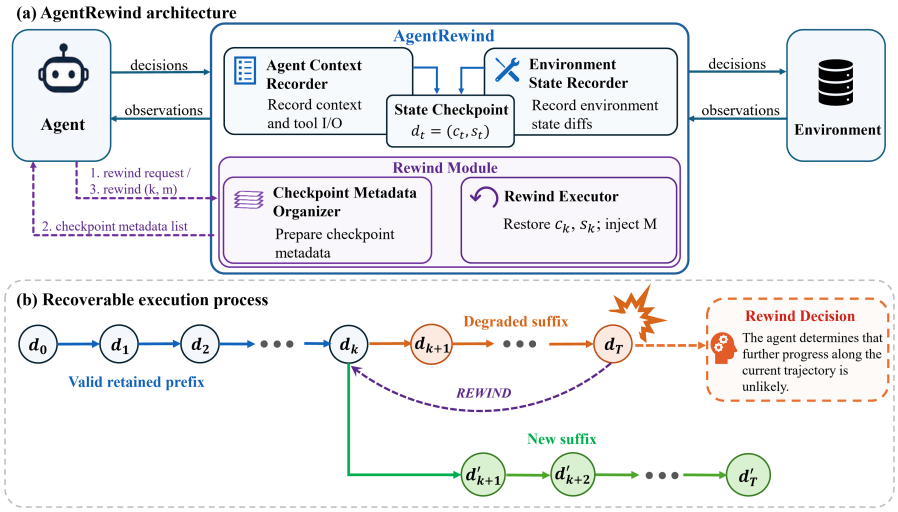

Figure 1: Overview of AgentRewind. (a) AgentRewind operates as a runtime layer between the agent and the controlled external environment. The context recorder and environment-state recorder create aligned checkpoints $d_t = (c_t, s_t)$. During rewind, the agent selects a target checkpoint, after which the rewind executor restores the corresponding context and environment state and injects rewind memory. (b) An illustrative recoverable execution. When the agent determines that the current trajectory can no longer make progress, it invokes rewind. The trajectory returns to d_k, retains the prefix through that checkpoint, and continues with a new suffix.

AGENTREWIND FRAMEWORK

AgentRewind provides a runtime recovery framework for long-horizon agent execution. It records recoverable checkpoints that capture the agent context and controlled environment state. When a rewind is triggered, AgentRewind restores the selected checkpoint and injects agent-generated rewind memory into the restored context. Figure 1 provides an overview of the system architecture and the recoverable execution process.

LONG-HORIZON EXECUTION WITH REWIND

We consider an agent executing long-horizon tasks in a controlled external environment. Given a task instruction x, the agent iteratively changes the environment state through tool calls. At step t, the LLM generates a decision u_t from its context c_t. The environment transitions from s_t to s_{t+1} and returns observation o_{t+1}, after which the agent context is updated:

$$u_t \sim \pi(c_t),\ (s_{t+1}, o_{t+1}) = \mathcal{T}(s_t, u_t),\ c_{t+1} = \mathcal{U}(c_t, u_t, o_{t+1}). \tag{1}$$

These updates produce a forward-growing trajectory:

$$\tau = \big((c_0, s_0), u_0, o_1, (c_1, s_1), \ldots, (c_T, s_T)\big). \tag{2}$$

Under standard execution, the trajectory can only grow forward. Once an early decision u_k introduces an error, its effects may propagate through subsequent contexts c_{k+1} and environment states s_{k+1}. Even if the agent later recognizes the error, it can only append corrective actions to the existing trajectory and may be unable to fully reverse the state changes caused by the erroneous prefix.

AgentRewind records the execution state at each LLM decision boundary as a recoverable checkpoint d_t:

$$d_t = (c_t, s_t). \tag{3}$$

For each state checkpoint, AgentRewind constructs associated checkpoint metadata η_t. The metadata describes the execution segment between d_t and d_{t+1}. It is used to present possible rewind locations to the agent.

Given the trajectory $\tau_{0:T}$, when the agent determines that the current trajectory can no longer make progress, it can select a historical checkpoint d_k based on its context and checkpoint metadata list, where $k \leq T$, and generate rewind memory m from the current trajectory. AgentRewind adds m to the set of historical rewind memories M:

$$M \leftarrow M \cup \{m\}. \tag{4}$$

AgentRewind then restores the controlled external environment state to s_k and injects the accumulated rewind memories into the recorded agent context c_k:

$$s'_k \leftarrow s_k,\ c'_k \leftarrow \mathrm{Inject}(c_k, M). \tag{5}$$

The agent subsequently resumes execution from (c'_k, s'_k), producing a new continuation of the trajectory:

$$\tau' = \big((c'_k, s'_k), u'_k, o'_{k+1}, \ldots\big). \tag{6}$$

Figure 1(b) illustrates this recoverable execution process. Rather than continuing only from the latest execution state, the agent can return to an earlier checkpoint and resume execution, while retaining information from previous attempts.

AgentRewind Runtime Recovery Layer

Based on the formalization above, AgentRewind augments the standard agent–environment interaction loop with runtime recovery. As shown in Figure 1(a), AgentRewind operates between the agent and the controlled external environment. During normal execution, AgentRewind transparently records the agent's trajectory without altering its interaction with the environment. The recorded trajectory includes the task instruction, LLM inputs and outputs, tool calls, and tool results. It also tracks tool-induced changes to the controlled environment and aligns them with the corresponding context records to form recoverable checkpoints. Therefore, the recorded trajectory supports the joint recovery of the agent context and the controlled environment state.

Rewind Module

AgentRewind exposes rewind as an additional control action to the agent. When the agent decides to invoke rewind, AgentRewind organizes the recorded trajectory into checkpoint metadata entries. Each entry summarizes an LLM output together with the tool interactions and environment changes that occur before the next LLM output. Based on the current execution trajectory, checkpoint metadata, environment feedback, and accumulated rewind memory, the agent selects a checkpoint and generates a new rewind memory that summarizes useful information from the current trajectory, such as falsified hypotheses and alternative strategies.

During rewind execution, AgentRewind terminates the current execution and restores both the external environment and the agent context to the selected checkpoint. The preceding LLM outputs and tool results are restored from the execution log rather than regenerated or re-executed. AgentRewind then injects the accumulated rewind memory into the restored context. The agent then begins generating a new suffix from the restored state.

External Environment Recovery Boundary

The controlled environment restored by AgentRewind is the workspace directory tree, whether the agent runs locally or in an isolated container. At each checkpoint, AgentRewind records file-level changes; rewinding reverts later modifications, restores deleted files, and removes newly created files. Effects outside the workspace filesystem, such as network requests, external-service calls, and external runtime state, cannot be undone. Because the retained prefix is restored from the execution log, these effects are not triggered again.

MettleBench

We next introduce the benchmark used for our main evaluation of long-horizon execution. Complex, long-horizon engineering assignments are rarely defined by a single requirement. Instead, they often contain a series of related requirements that together determine whether the assignment has

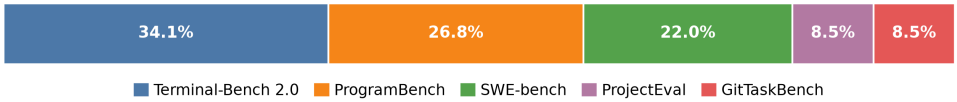

Figure 2: Distribution of the 82 MettleBench tasks across source benchmarks. Percentages are rounded to one decimal place.

been completed. For example, migrating an internal service may require upgrading its dependencies, transferring configuration and data, preserving compatibility with existing clients, updating deployment scripts, verifying service availability, and documenting the resulting changes. By contrast, many existing agent benchmarks formulate each task around one primary requirement, even when completing it involves multiple execution steps. MettleBench represents this form of long-horizon work by treating each task as a single engineering assignment containing a series of complex requirements. Formally, a task and its ordered acceptance criteria are represented as

$$\mathcal{T} = (x, s_0, \mathcal{U}, G), \qquad G = (g_1, \ldots, g_n), \tag{7}$$

where x is a natural-language task instruction, s_0 is the initial environment state, $\mathcal{U}$ is the agent's decision space, and G is the ordered list of acceptance criteria. Each acceptance criterion corresponds to a binary evaluation function $g_i : \mathcal{S} \to \{0, 1\}$, where $g_i(s) = 1$ indicates that the environment state s satisfies the i-th criterion. Task success at the final state s_T is defined as

$$\mathrm{Succ}(s_T) = \bigwedge_{i=1}^{n} \big(g_i(s_T) = 1\big). \tag{8}$$

A task is therefore successful only when the final state satisfies all acceptance criteria. To characterize partial progress, let $\ell \in \{0, \ldots, n\}$ denote the length of the longest prefix $(g_1, \ldots, g_\ell)$ such that $g_i(s_T) = 1$ for all $i \leq \ell$. The checklist prefix progress is defined as

$$\rho(s_T) = \frac{\ell}{n}. \tag{9}$$

This metric distinguishes different levels of progress when a task is not fully completed.

MettleBench comprises 82 tasks derived from five existing engineering benchmarks: Terminal-Bench 2.0 (Merrill et al., 2026), ProgramBench (Yang et al., 2026a), SWE-bench (Jimenez et al., 2024), ProjectEval (Liu et al., 2025), and GitTaskBench (Ni et al., 2026). Figure 2 summarizes the distribution of tasks across these sources. We retain the underlying engineering artifacts and executable task environments while rewriting the natural-language task instructions to express a series of related requirements within each assignment. For each rewritten instruction, we construct a hidden, ordered checklist of interdependent acceptance criteria, each implemented as an executable check.

Upon each agent submission, the backend executes these checks in their predefined order and returns natural-language feedback identifying only the first unsatisfied checklist item. The full checklist remains hidden from the agent, which may continue revising its execution based on the returned feedback. Because all criteria share the same environment state, satisfying one criterion may enable or hinder later criteria, while subsequent actions may invalidate previously satisfied criteria. MettleBench does not require agents to use any particular recovery mechanism. Each task is screened for a valid forward-only solution, allowing the benchmark to evaluate long-horizon execution independently of any specific recovery design. Continue trace lengths in Table 1 provide an empirical characterization of MettleBench's execution horizons. Further benchmark construction details are provided in Appendix A.

Experiments

We evaluate AgentRewind on MettleBench using task success rate and average checklist progress. Task success rate is the proportion of tasks that satisfy all acceptance criteria before termination. Average checklist progress is the mean of the task-level checklist prefix progress $\rho(s_T)$ defined in the MettleBench section, where s_T denotes the final environment state at the end of a run. We impose no preset limits on wall-clock time, execution steps, or total token usage. A run ends upon task

success; otherwise, to prevent methods from continuing or retrying indefinitely, we apply the same repeated-failure termination condition to all strategies. If the environment identifies the same first unsatisfied acceptance criterion in five consecutive submissions, the run is terminated and recorded as unsuccessful. Unless otherwise specified, all experiments use GPT-5.4 as the base model and mini-SWE-agent as the agent harness, with the same task environments, tool interfaces, and termination conditions across compared methods. Further experimental details are provided in Appendix B.

Table 1: Continue baselines across seven base models on MettleBench.

Metric	GPT-5.4	GPT-5.4 mini	Qwen3.7-Max	Qwen3.5-27B	Kimi K2.5	DeepSeek-V4-Flash	GLM-5.1
Success rate (%)↑	62.2	33.7	73.2	56.1	28.0	37.8	59.8
Checklist progress (%)↑	81.4	64.6	84.8	79.1	61.3	63.6	75.1
Trace length	50.5	72.4	126.3	211.6	312.4	337.2	324.8

Main Results

Continue Baselines Across Base Models.

We establish forward-execution baselines on MettleBench using seven base models, all running mini-SWE-agent (Yang et al., 2024) with Continue, which proceeds from the current context and environment after each validation response. Trace length counts recorded LLM and tool events. With other settings fixed, Table 1 reports task success, checklist progress, and trace length.

Continue success ranges from 28.0% to 73.2%, and no model saturates the benchmark. Qwen3.7-Max leads in both success and checklist progress, while Kimi K2.5 ranks lowest, showing meaningful variation in forward-execution capability. Checklist progress complements binary success: GPT-5.4 and GLM-5.1 have similar success rates, but their average progress on unsuccessful runs is 50.8% and 38.1%, respectively.

Trace length ranges from 50.5 to 337.2 events per task; Kimi K2.5, DeepSeek-V4-Flash, and GLM-5.1 each exceed 300. Despite having the shortest traces, GPT-5.4 ranks second in success and checklist progress, while longer traces do not consistently yield better outcomes. Thus, execution length alone does not ensure progress, and these baselines establish headroom for the strategy comparisons that follow.

Comparison of Execution Strategies.

We compare four strategies: **Continue**, which proceeds from the current state after receiving validation feedback; **Restart with Experiences**, which resets the environment and retries with prior experiences; **Safety Review**, which adds AgentDoG-based action review to Continue; and **AgentRewind**. GA-Rollback (Li et al., 2025) and WebRollback (Zhang et al., 2026) lack rollback infrastructure for our code-execution setting, while DART (Yang et al., 2026b) assumes predefined subtask boundaries; we therefore omit these methods as baselines.

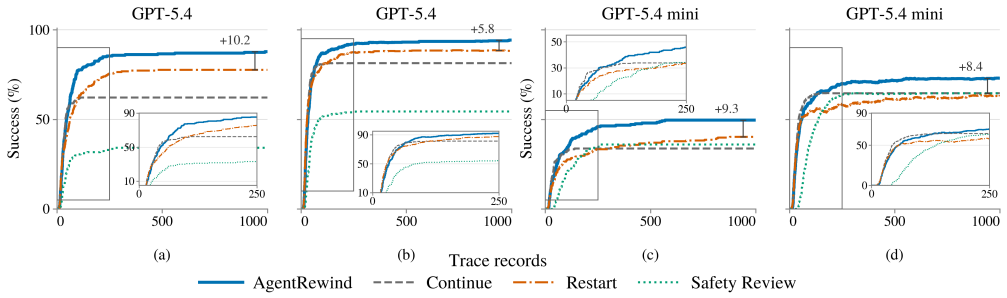

Figure 3: Task success (a, c) and checklist progress (b, d) over trace length for GPT-5.4 (a, b) and GPT-5.4 mini (c, d). Curves show averages over three runs and are extended horizontally at their final observed values after termination.

Figure 3 compares the four execution strategies under GPT-5.4 and GPT-5.4 mini. AgentRewind achieves the highest final task success and checklist progress under both models, outperforming the strongest baseline in every setting. All strategies improve rapidly early in execution but diverge after trajectories begin encountering persistent failures. Continue plateaus earliest, indicating that further repair from the current state yields limited benefit once a trajectory becomes stuck. Restart with Experiences improves final success over Continue under both models but shows inconsistent checklist-progress gains and larger fluctuations. Safety Review performs markedly worse under GPT-5.4, whereas under GPT-5.4 mini it is comparable to Continue. It therefore does not consistently outperform Continue across the two evaluated models. By contrast, AgentRewind continues improving after the baselines plateau and exhibits fewer progress regressions than Restart with Experiences. Its consistent gains under both models indicate that runtime recovery remains effective across models with substantially different Continue performance. To rule out the possibility that AgentRewind benefits merely from a larger LLM budget, we compare all strategies under the same per-task main-model token caps. Rewinding reconstructs the retained context from the execution record without reissuing LLM calls; previously incurred calls remain counted, and all new calls after a rewind or restart are also included in the step and token statistics. As shown in Table 2, AgentRewind is the strongest method from 1M tokens onward for both models, and its advantage grows with the cap: the gain over the strongest baseline increases from 3.3 to 8.5 points for GPT-5.4 and from 1.2 to 6.9 points for GPT-5.4 mini.

Table 2: Task success (%) under common per-task main-model token caps. Values are means over three repetitions of 82 tasks; Δ is the gain over the strongest baseline in percentage points.

	GPT-5.4					*GPT-5.4 mini*				
Cap	Cont.	Rest.	Safety	Rewind	Δ	Cont.	Rest.	Safety	Rewind	Δ
1M	61.0	66.7	40.2	**69.9**	+3.3	30.9	29.7	32.5	**33.7**	+1.2
2M	62.2	73.6	42.7	**80.9**	+7.3	33.7	33.3	38.6	**41.5**	+2.8
5M	62.2	77.6	43.9	**86.2**	+8.5	33.7	38.2	39.8	**46.7**	+6.9

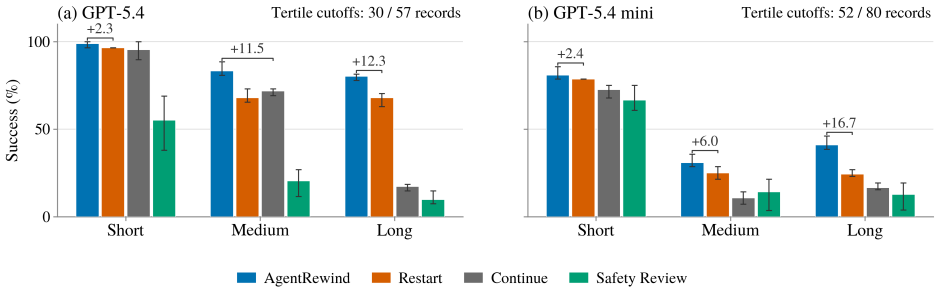

Figure 4: Success by Continue trace-length tertiles. Error bars show run ranges; brackets show AgentRewind's gain over the strongest baseline.

To examine how recovery gains vary with the forward-execution horizon, we group tasks for each model by their median Continue trace length across three runs. Figure 4 shows that AgentRewind's advantage over the strongest baseline is modest in the short-horizon group but becomes substantially larger in the medium- and long-horizon groups. This pattern suggests that rewind is particularly useful when standard forward execution becomes prolonged.

To test whether AgentRewind remains effective beyond MettleBench, we evaluate Continue, Restart with Experiences, and AgentRewind on the full Terminal-Bench 2.0 under the same termination condition (Merrill et al., 2026). Because its acceptance criteria are not semantically ordered, partial completion is measured as the average fraction of criteria satisfied per task. Table 3 reports the results. AgentRewind outperforms both Continue and Restart with Experiences on task success and partial completion. These results demonstrate that AgentRewind's benefits extend beyond MettleBench's task formulations.

Table 3: Results on Terminal-Bench 2.0 tasks.

Metric	Continue	Restart with Experiences	AgentRewind
Success rate (%) ↑	78.7	70.8	**83.1**
Avg. criteria passed (%) ↑	88.7	79.2	**90.2**

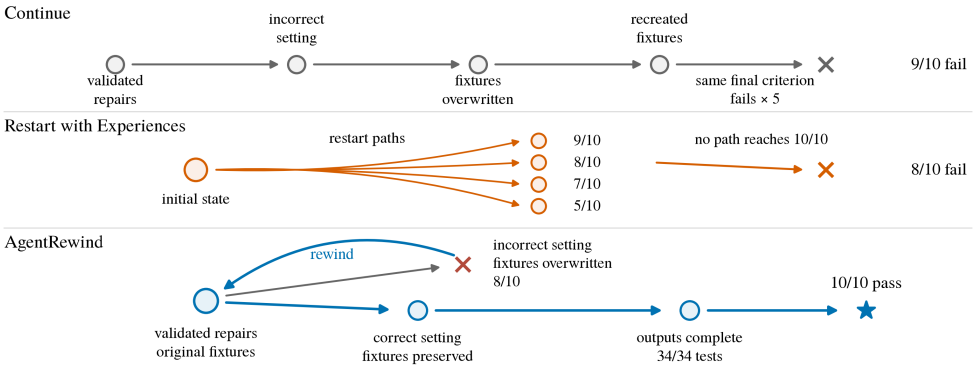

Figure 5: Execution paths of Continue, Restart with Experiences, and AgentRewind in the Astropy FITS case. Scores denote checklist prefix progress.

Case Study.

We examine an Astropy FITS handoff task under GPT-5.4 mini. The agent must repair defects in the FITS library, generate a catalog covering valid images and six intentionally malformed fixtures, and preserve those fixtures so the repaired library continues to reject them. Figure 5 summarizes the three strategies at the catalog-generation step.

Continue repaired the library but used an incorrect catalog-generator mode, which overwrote the six unreadable fixtures with valid FITS images. Although it later recreated malformed files and restored the expected rejection behavior, they did not match the originals, and the run ended at 9/10 checklist progress.

Restart with Experiences recovered the original fixtures on every attempt but discarded completed repairs and maintenance artifacts. Across 26 attempts, some runs failed to preserve the malformed fixtures, while others omitted the changelog entry or left the reconciliation ledger incomplete. No attempt satisfied all requirements, and the final attempt ended at 8/10.

AgentRewind reached 8/10 with the same incorrect setting, then rewound to the checkpoint before catalog generation. This retained the validated repairs, restored the original fixtures, removed the abandoned branch from context, and recorded its consequences in rewind memory. The replacement suffix used the correct mode, preserved and recorded the unreadable inputs, completed the catalog, reconciliation ledger, and attestation, passed all 34 tests, and reached 10/10.

Table 4: Continue and AgentRewind across three agent harnesses using GPT-5.4.

Variant	mini-SWE		FnCall		CodeAgent	
	SR	Progress	SR	Progress	SR	Progress
Continue	62.2	81.4	58.5	77.9	67.1	83.0
AgentRewind	**87.8**(+25.6)	**94.3**(+12.9)	**81.7**(+23.2)	**91.4**(+13.5)	**82.9**(+15.8)	**89.7**(+6.7)

Comparison of Agent Harnesses

AgentRewind is designed as a runtime recovery layer that does not depend on a particular agent harness. To evaluate its generality across harnesses, we apply AgentRewind to three agent harnesses: mini-SWE-agent (Yang et al., 2024), FnCallAgent from Qwen-Agent (Qwen Team, 2023),

and CodeAgent from smolagents (Roucher et al., 2025). All harnesses use GPT-5.4 as the base model. For each harness, we compare Continue with AgentRewind under otherwise identical settings.

Table 4 shows that AgentRewind improves both task success rate and average checklist progress across all three agent harnesses. Despite differences in their control loops and tool-interaction interfaces, the improvements remain consistent across the evaluated harnesses. With the base model fixed, these results suggest that the benefits of AgentRewind are not specific to a particular harness, but arise from the added runtime recovery capability.

Recovery and Ablation Study

Recovery from Failed Trajectories.

To compare the ability of Continue and AgentRewind to recover from the same failed execution state, we collect 50 Continue trajectories that terminated under the repeated-failure condition. For each failed endpoint, we launch paired Continue and AgentRewind recovery runs from identical copies of the agent context and environment state. Both runs receive the same recovery prompt, noting the repeated failure and requesting a different approach, and use independently reset failure counters under the standard termination condition. Table 5(a) shows that AgentRewind improves both recovery rate and checklist progress change over Continue, demonstrating stronger recovery from the same repeatedly failing states.

Table 5: Recovery and ablation results for AgentRewind. Panel (a) compares recovery from 50 paired failed Continue endpoints; progress change is measured relative to each shared endpoint. Panel (b) reports the component ablation results.

(a) Recovery from failed trajectories

Metric	Continue	Agent Rewind
Recovery rate (%) ↑	8.0	**30.0**
Checklist Δ (pp) ↑	5.1	**12.2**

(b) Component ablation

Metric	Full	w/o Env.	w/o Context	w/o Memory
Success rate (%)	**87.8**	43.9	65.9	51.2
Checklist progress (%)	**94.3**	63.5	77.9	69.4

Component Ablation.

Table 5(b) compares full AgentRewind with variants that remove environment rewind, context rewind, or rewind memory. Removing environment rewind causes the largest degradation because changes from the discarded suffix remain in the environment. Without context rewind, discarded actions, observations, and intermediate conclusions remain available to subsequent decisions. Removing rewind memory discards useful information from failed attempts and increases the risk of repeating unsuccessful decisions, showing that state restoration and rewind memory provide complementary recovery functions.

Conclusion

AgentRewind provides runtime recovery by restoring aligned agent-context and environment checkpoints while retaining information from prior attempts. MettleBench evaluates long-horizon engineering tasks. Across benchmarks, models, strategies, and harnesses, AgentRewind improves task success and checklist progress; ablations confirm the value of state restoration and rewind memory. Future work will broaden recovery beyond controlled state and add internal stall detection.

AI Use Disclosure

Generative AI tools were used to assist with language editing and code implementation. All AI-assisted outputs were reviewed and verified by the authors, who take full responsibility for the manuscript, implementation, and reported results.

REFERENCES

Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12. IEEE, 2009. doi: 10.1109/IPDPS.2009.5161063. URL https://doi.org/10.1109/IPDPS.2009.5161063.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. Agentrx: Diagnosing ai agent failures from execution trajectories, 2026. URL https://arxiv.org/abs/2602.02475.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=moWiYJuSGF.

CRIU Project. CRIU: Checkpoint/restore in userspace, 2026. URL https://criu.org/. Version 4.2, accessed 2026-06-08.

Yunpeng Dong, Jingkai He, Yuze Hou, Dong Du, Zhonghu Xu, Si Yu, Yubin Xia, and Haibo Chen. Deltabox: Scaling stateful ai agents with millisecond-level sandbox checkpoint/rollback, 2026. URL https://arxiv.org/abs/2605.22781.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=ybA4EcMmUZ.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 1769–1782. PMLR, 2023. URL https://proceedings.mlr.press/v205/huang23c.html.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. BIRD-INTERACT: Re-imagining text-to-SQL evaluation via lens of dynamic interactions. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=nHrYBGujps.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks, 2026. URL https://arxiv.org/abs/2604.24964.

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.

Jing Yu Koh, Stephen Marcus McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=QF0N3x2XVm.

Patrick Tser Jern Kon, Qiuyi Ding, Jiachen Liu, Xinyi Zhu, Jingjia Peng, Jiarong Xing, Yibo Huang, Yiming Qiu, Jayanth Srinivasa, Myungjin Lee, Mosharaf Chowdhury, Matei Zaharia, and Ang Chen. EXP-bench: Can AI conduct AI research experiments? In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=KjgyAm383Z.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=XmProj9cPs.

Jinchao Li, Yunxin Li, Chenrui Zhao, Zhenran Xu, Baotian Hu, and Min Zhang. Windowsworld: A process-centric benchmark of autonomous gui agents in professional cross-application environments. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 15262–15280, 2026.

Xingzuo Li, Kehai Chen, Yunfei Long, Xuefeng Bai, Yong Xu, and Min Zhang. Generator-assistant stepwise rollback framework for large language model agent. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17683–17700, Suzhou, China, nov 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.892. URL https://aclanthology.org/2025.emnlp-main.892/.

Dongrui Liu, Yu Li, Zhonghao Yang, Peng Wang, Guanxu Chen, Yuejin Xie, Qinghua Mao, Wanying Qu, Yanxu Zhu, Tianyi Zhou, et al. AgentDoG 1.5: A lightweight and scalable alignment framework for ai agent safety and security. *arXiv preprint arXiv:2605.29801*, 2026.

Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. ProjectEval: A benchmark for programming agents automated evaluation on project-level code generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20205–20221, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1036. URL https://aclanthology.org/2025.findings-acl.1036/.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Sen Hu, Bo Li, Chen Hu, Binxing Jiao, Daxin Jiang, Yuntao Du, and Pin Lyu. GitTaskBench: A benchmark for code agents solving real-world tasks through code repository leveraging. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(38):32564–32572, 2026. doi: 10.1609/aaai.v40i38.40533. URL https://ojs.aaai.org/index.php/AAAI/article/view/40533.

Qwen Team. Qwen-Agent: Agent framework and applications. Software repository, 2023. URL https://github.com/QwenLM/Qwen-Agent.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. Software repository, 2025. URL https://github.com/huggingface/smolagents.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=GEcwtMk1uA.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 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)*, pp. 16022–16076, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36:38975–38987, 2023.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*, 2026a. doi: 10.1145/3744916.3764546. URL https://doi.org/10.1145/3744916.3764546.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, Abu Dhabi, United Arab Emirates, dec 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

Xinyu Jessica Wang, Haoyue Bai, Yiyou Sun, Haorui Wang, Shuibai Zhang, Wenjie Hu, Mya Schroder, Bilge Mutlu, Dawn Song, and Robert D Nowak. The long-horizon task mirage? diagnosing where and why agentic systems break. *arXiv preprint arXiv:2604.11978*, 2026b.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard llm agents via knowledge-enabled reasoning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 68316–68342. PMLR, 2025. URL https://openreview.net/forum?id=2nBcjCZrrP.

Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking LLM agents on consequential real world tasks. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/0d744742f6fac4d1134c019b7cef3c8a-Abstract-Datasets_and_Benchmarks_Track.html.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/e2f03935e0db48e2daad04a5faac1bb2-Abstract-Datasets_and_Benchmarks.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026a. URL https://arxiv.org/abs/2605.03546.

Ke Yang, Panpan Li, Zonghan Wu, Kejin Xu, Huaxi Huang, and Xiaoshui Huang. DART: Semantic recoverability for structured tool agents, 2026b. URL https://arxiv.org/abs/2605.23311.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, volume 35, pp. 20744–20757, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 1962–1979, Hong Kong, China, nov 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1204. URL https://aclanthology.org/D19-1204/.

Zhisong Zhang, Tianqing Fang, Kaixin Ma, Wenhao Yu, Hongming Zhang, Haitao Mi, and Dong Yu. WebRollback: Enhancing web agents with explicit rollback mechanisms. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 187–197, Rabat, Morocco, mar 2026. Association for Computational Linguistics. ISBN 979-8-89176-381-4. doi: 10.18653/v1/2026.eacl-short.12. URL https://aclanthology.org/2026.eacl-short.12/.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where llm agents fail and how they can learn from failures, 2025. URL https://arxiv.org/abs/2509.25370.

APPENDIX

A METTLEBENCH: CONSTRUCTION PROTOCOL AND QUALITY ASSURANCE

A.1 TASK SOURCES AND COMPOSITION

MettleBench comprises 82 tasks derived from five existing engineering benchmarks. We retain the underlying engineering artifacts and executable task environments, and rewrite the task specifications and acceptance criteria so that each task becomes a long-horizon assignment with ordered, interdependent objectives. Figure 2 of the main paper gives the distribution of tasks across those benchmarks. By assigning each acceptance criterion the source benchmark of its task, we obtain the criterion-level distribution shown in Table 6, covering all 640 criteria.

A.2 PROVENANCE OF THE ACCEPTANCE CRITERIA

The criteria were written by an LLM agent, under a protocol we wrote, and admitted to the benchmark only after passing deterministic checks. The three parts of that pipeline are as follows.

The protocol fixed what a task had to look like. Every task is one engineering assignment containing a series of related requirements, and it has to meet the structural requirements of Section A.6. A task may not be made difficult by artificial means. It may not stretch a run out by forcing the agent to wait, or to work one item at a time when it could work in batches. It may not require a value that the agent has no way to work out and can only guess at. It may not depend on an option that is documented nowhere the agent can read. The libraries and command-line tools in the environment behave as they do outside the benchmark. The protocol also fixed what the agent learns from a rejected submission. As described in the main paper, the environment returns natural-language feedback identifying only the first unsatisfied checklist item. That feedback has to state what went wrong in observable terms, and it never names the command, the script, or the checklist internals responsible for the failure.

Starting from each source task, an LLM agent then wrote the task instruction, the ordered checklist, the executable check behind each criterion, a forward-only reference solution, and a second reference run that carries out the same assignment in an order that ignores the dependencies among the criteria.

Table 6: Distribution of the acceptance criteria of MettleBench across the source benchmarks, counting every criterion of a task towards that task's source. The last column is the mean number of criteria per task for that source. No source contributes disproportionately many criteria: every share is within three points of the corresponding share of tasks in Figure 2 of the main paper. Percentages are rounded to one decimal place.

Source benchmark	Criteria	Share (%)	Per task
Terminal-Bench 2.0	208	32.5	7.43
ProgramBench	168	26.2	7.64
SWE-Bench	158	24.7	8.78
ProjectEval	56	8.8	8.00
GitTaskBench	50	7.8	7.14
Total	**640**	**100.0**	**7.80**

Table 7: Acceptance-criteria statistics over the 82 tasks.

Statistic	Value
Tasks with machine-readable checklist	82 / 82
Total criteria	640
Criteria per task: mean (sd)	7.80 (1.63)
Criteria per task: median	7
Criteria per task: range	5 – 12
Criteria with failure feedback	640 / 640
Top criterion categories	
data integrity	160
correctness	148
evidence preservation	110
release / packaging	84
file content	60

Three deterministic gates decided admission. The reference solution has to satisfy every criterion. The order-violating run has to fail at the criterion whose precondition it removes. The workspace as shipped has to fail, so that no task is already solved when the agent receives it.

What makes a task hard is not something we added on top of the environment. Two properties of the underlying systems do that work. First, all criteria of an assignment act on the same environment state, so satisfying one criterion may enable or hinder later ones. Second, many operations that ordinary engineering work requires cannot be undone by the tools that perform them: rewriting git history discards objects that no reference points to, dropping a column discards its values, and regenerating a derived file overwrites the source it came from. Operations of this kind are routine in the work these tasks are drawn from. The protocol does not add failure conditions that the tools themselves would not produce.

Every task ships a machine-readable checklist. Each criterion in it carries an identifier, a category, a description of what the criterion is meant to establish, and the feedback string returned when it fails. All 640 criteria across the 82 tasks carry such feedback (Table 7).

Human involvement. We reviewed the tasks in the benchmark. That review was at the task level: we did not annotate every criterion of every task by hand. It covered prompt naturalness, fidelity to the source task, fairness of the verifier, the absence of contrived mechanisms, and four further dimensions, each scored from 1 to 5 by an adversarial LLM reviewer. Dimension means run from 3.65 to 5.00, and every reviewed task was accepted. For each task the reviewer also had to answer two questions directly: does the situation match a failure that occurs in ordinary engineering practice, and is any mechanism in the task contrived. We spot-checked tasks throughout, and during the formal runs we kept a log of manual interventions (Section B.7).

A.3 Checklist Statistics

Table 7 gives the size and composition of the checklists. Each task carries between 5 and 12 ordered criteria (mean 7.80, median 7), for 640 criteria in total.

A.4 Every Task Has a Forward-Only Solution That Satisfies All Criteria at Once

Every task ships an executable, forward-only reference solution: a linear script with no rollback and no retry, 79.3 non-empty lines long on average (median 71, range 16 to 220). We ran all 82 of them again against fresh copies of the shipped workspaces in a clean environment, then ran the full evaluator on each resulting state. The reference solution of every task satisfies all of its acceptance criteria at once, in a single forward pass.

Every task was checked in the same three ways during construction, in runs made independently of the one reported here, and those records agree: the reference solution passes, the order-violating run fails at the criterion whose precondition it removes, and the workspace as shipped fails. A task is therefore not satisfied by the state it ships in, nor by an execution that ignores the dependencies among its criteria.

Because these solutions are strictly forward-only, no task needs a recovery mechanism in order to be solved. The runs themselves show the same thing. All 82 tasks were evaluated with seven base models under the Continue strategy, and the two GPT-5.4 variants were additionally evaluated under the three other execution strategies. At least 70 of the 82 tasks were solved by some model under Continue alone, and at least 79 were solved under some execution strategy. The tasks that no run solved are difficult rather than unsolvable: their reference solutions satisfy every criterion.

A.5 Evaluator Determinism and Repeatability

The evaluator is a program, not a model. Each task's evaluator is plain Python that inspects the final workspace state (mean 379 non-empty lines, range 222 to 637; 31,056 lines in total). No LLM takes part in judging. We audited the source of all 82 evaluators: none draws a random number anywhere in a judgment, none reaches the network, and none generates a UUID (the one `uuid` reference parses a fixed constant). Two evaluators read the wall clock. One sets file modification times by a fixed relative offset, which leaves the verdict deterministic. The other waits, with a timeout, for a locally started gRPC service to come up, and that is the only point in the benchmark where timing can affect a result.

Submission-time behavior is deterministic. On each submission the evaluator runs the criteria in their fixed order and returns the feedback for the first unsatisfied one. Identical final states receive identical verdicts and identical feedback.

End-to-end repeatability. We ran every configuration of the main comparison three times from scratch. Call one (strategy, model) pair a *cell*. The main comparison has eight cells, four strategies by two models, and each cell contains the same 82 tasks, which gives $82 \times 8 = 656$ (task, strategy, model) triples, each observed in three independent runs. A triple is *in agreement* when its three runs ended with the same verdict, that is, all three succeeded or all three failed. By that definition 490 of the 656 triples (74.7%) are in agreement. Computing the same quantity separately inside each cell gives a per-cell agreement between 48.8% (Safety Review with GPT-5.4: 40 of its 82 tasks agree across the three runs) and 90.2% (Restart with Experiences with GPT-5.4: 74 of 82).

The remaining 25.3% are tasks whose repetitions did not all end alike. That figure does *not* measure evaluator noise. It collects every source of run-to-run variation, including the agent's own trajectory, what the tools return, and filesystem timing, so it is an upper bound on the share of disagreement the evaluator could account for, not an estimate of it. The bound is a loose one, and the cells at the bottom of the range show why: both of them are Safety Review cells, where an external monitor intercepts individual actions and adds variation to the trajectory, while the evaluator is the same code used in every other cell. The determinism of the evaluator itself rests on the properties above, namely that identical final states always receive identical verdicts.

Table 8: Hidden ordered checklist of the worked example. The criteria are interdependent through the shared repository state and through ordinary git semantics: purging a credential from history (P05) ends in pruning unreachable objects, which physically discards the commits that P04 requires unless they have already been made reachable from a ref. Later criteria can likewise invalidate earlier ones.

#	Category	Criterion (intent)
P01	correctness	The leak scanner enumerates every object in a repository, reports correct match statistics for the healthy reference repository, and its unit tests pass.
P02	data integrity	A baseline scan summary and per-object inventory of the reference repository exist with correct counts.
P03	data integrity	The handoff records the exact leaked credential values the responder needs to rotate.
P04	recovery	The teammates' lost feature commits are reachable from refs again, with content and history intact.
P05	correctness	No leaked credential survives anywhere: not in reachable history, any patch, or any loose/dangling/packed object.
P06	data integrity	Ordinary (non-secret) project files and commit history remain intact after the rewrite.
P07	file content	The handoff report documents the recovered credentials, the history rewrite, the recovered work, and the post-cleanup verification.

A.6 Long-Horizon Screening Criteria

To keep a task from collapsing into a single requirement, we required every task to meet four structural conditions.

It must have at least four stages that run in sequence, where each stage can only start once the previous stage has produced its deliverables. Its later deliverables must quote values that earlier stages actually produced at run time, such as digests, counts, or identifiers, in at least two places; an agent that skips or fakes an earlier stage therefore cannot fill in a later one. It must contain either at least two defects that have to be repaired one after another, where fixing the first exposes the second, or a log of at least 30 recorded entries that has to be reconstructed. And it must carry at least five acceptance criteria, which is also the smallest number in the released set.

Table 1 of the main paper reports the average trace lengths that result, per model.

A.7 Worked Example: Task Prompt and Checklist

We use a security-incident task derived from Terminal-Bench to show what a task looks like. Figure 6 is the full task instruction, and Table 8 is the hidden ordered checklist behind it. The agent never sees the checklist. On each submission it receives only the feedback for the first unsatisfied criterion, which for P01 reads: *"The leak scanner still reports incorrect statistics for the reference repository, or its unit tests are not green. Reconsider which objects the scanner walks and how it decides a blob matches."*

The criteria of the example all act on the same repository, so satisfying one may hinder a later one. Removing the leaked credentials from the history requires discarding the old commits that contain them, and the teammates' lost work consists of exactly such commits, which no branch currently points to. An agent that removes the credentials before putting that work back on a branch destroys it permanently, and P04 can no longer be satisfied. Scoring a run by how far it advances along the ordered checklist, as in the main paper, then separates different levels of progress when a task is not fully completed: a run that stops at P03 is distinguished both from one that fails at the first criterion and from one that satisfies all seven.

```
You are picking up a security-incident handoff for the `billing-service`
repository under `repo/`. Work it like the on-call responder preparing
the cleanup packet, and use `docs/runbook.md` and the notes under
`.ci/security/` as your source of truth for the process and the exact
deliverables.

Three things are open.

First, the leak scanner `tools/leak_scan.py` is failing its CI
self-check (`.ci/security/`). Fix the scanner so its unit tests under
`tests/` pass, then use the repaired tool to characterise the healthy
reference repository `reference/history-sample` and record a baseline
of its object statistics under `baseline/`.

Second, several teammates' work went missing. More than one feature
branch was deleted before it was merged, so different stretches of
unmerged work are no longer on any branch -- they survive only as
unreachable objects in `repo/`. Track down every piece of that lost
work and bring it back so it is part of the repository again and can
be reviewed and merged later.

Third, over the past few weeks deploy credentials were committed
straight into the history, in different files and formats. Some were
later removed from the working tree; others are still sitting in it.
Scan the whole history for every leaked credential, recover each value
so the responder can rotate it, then scrub them so none can be found
anywhere in `repo/` -- not in reachable history, not in `git log -p`
output, and not in any loose or dangling object.

Assemble the responder packet under `dist/security_handoff/`: the
recovered credentials in `recovered_secret.txt` (each value on its own
line so it can be rotated verbatim), and a `handoff_report.md` tying
together the recovered credentials, how the history was rewritten, the
recovered teammate work, and the checks you ran to confirm the
repository is clean afterwards.

Do not fetch anything from the network and do not re-clone or replace
the repository with a fresh copy. Keep the ordinary (non-secret)
project files and commit history intact, and leave the tree free of
transient build artifacts. The exact deliverables and their layout are
spelled out in `docs/runbook.md`.
```

Figure 6: Full task instruction of the worked example.

B Experimental Details

B.1 Setup of the Base-Model Comparison

Table 9 gives the configuration of the base-model comparison under the Continue strategy (Table 1 of the main paper). All seven models run the same 82 tasks with the same harness, tools, and termination condition; only the base model changes. Decoding is greedy (temperature 0.0) for every model, with one exception. Kimi K2.5 is evaluated in its no-thinking mode, for which the vendor specifies a temperature of 0.6, and we use that setting rather than override it. All values are taken verbatim from the recorded launch commands of the reported runs.

B.2 Setup of the Execution-Strategy Comparison

Table 10 gives the configuration of the main experiments, which compare four execution strategies under GPT-5.4 and GPT-5.4 mini. The upper block holds the settings shared by all four strategies; the lower blocks hold the parameters specific to each. All strategies use greedy decoding (temperature 0.0), the same task environments, tool interfaces, and termination condition, and differ only in what

Table 9: Configuration of the base-model comparison under Continue (Table 1 of the main paper).

Parameter	Value
Execution strategy	Continue
Agent harness	mini-SWE-agent
Base models	7 (see Table 1 of the main paper)
Temperature	0.0
exception: Kimi K2.5	0.6 (no-thinking mode)
Step limit per run	none
Wall-clock limit per episode	none
Shell command timeout	none
Model request timeout	600 s
Repeated-failure termination	5 identical first failures
Submission-failure mode	continue

Table 10: Configuration of the execution-strategy comparison. The strategies share every setting in the upper block and differ only in the parameters listed under their own headings.

Parameter	Value
Shared by all strategies	
Base models	GPT-5.4, GPT-5.4 mini
Independent runs	3
Temperature	0.0
Step limit per run	none
Wall-clock limit per episode	none
Shell command timeout	none
Model request timeout	600 s
Repeated-failure termination	5 identical first failures
Continue	
Submission-failure mode	continue
Restart with Experiences	
Submission-failure mode	exit after failure
Restart strategy	fresh environment
Carried to the next attempt	failure experiences
Safety Review	
Submission-failure mode	continue
Monitor	AgentDoG1.5-Qwen3.5-4B
Monitor timeout	none
Reviewed unit	every proposed tool action
AgentRewind	
Submission-failure mode	continue
Rewinds per run	unlimited
Checkpoint candidates shown	80
Rewind memory	cumulative
Context restoration	rolled back
Workspace restoration	rolled back
Retained prefix	restored from the execution log

happens after a submission is rejected. We executed every configuration three times with identical parameters.

Terminal-Bench 2.0. The experiment on the full Terminal-Bench 2.0 (Table 2 of the main paper) compares Continue, Restart with Experiences, and AgentRewind on its 89 tasks, with the base model, decoding, tool interface, and termination condition of Table 10. The acceptance criteria of that benchmark are not ordered, so partial completion there is the average fraction of a task's criteria that are satisfied rather than the checklist progress we report on MettleBench.

Other agent harnesses. The comparison across harnesses (Table 3 of the main paper) replaces mini-SWE-agent with FnCallAgent or CodeAgent and compares Continue with AgentRewind under GPT-5.4 on the same 82 tasks, with the decoding, termination condition, and rewind parameters of Table 10. What differs besides the harness is how the rewind tools are presented, since each framework expects tools in its own form (Section C.1).

B.3 Execution Strategies

The four strategies share the same agent, tools, task environments, and termination condition (Table 10). They differ only in what happens after the evaluator rejects a submission.

Continue. The feedback for the first unsatisfied criterion is appended to the trajectory as an ordinary user message, and the same agent loop keeps stepping. Nothing is reset. The process, the message history, and the workspace on disk all persist, and the history now also holds the work that was just rejected. The agent's tool interface is a single shell tool, so it has no way to rewind. A run ends when the task succeeds, or when five consecutive submissions fail at the same criterion.

Restart with Experiences. The first rejection ends the current attempt. An outer loop then starts successive attempts, each in a fresh process whose workspace is reset to the pristine task snapshot and whose message history starts empty. The only thing that crosses from one attempt to the next is a file of failure experiences, which the batch driver writes and the next attempt reads as a section appended to its first message.

The experiences are assembled from the evaluator's output. After every failed attempt the driver appends one block to the file recording, for that attempt: the identifier of the first unsatisfied criterion together with the full feedback text and log excerpt returned for it; how many criteria were satisfied as a prefix; which criteria had been satisfied earlier and then regressed; the counts of passed, failed, and not-yet-run criteria; how many consecutive submissions have now failed at the same criterion and how many remain before the run is stopped; and two constraint lines generated from the failure identifier, telling the next attempt not to repeat fixes that leave that criterion failing and to address the repeated failure before exploring more widely.

Blocks accumulate rather than replace one another, and the next attempt receives the whole file, so an agent on its ninth attempt sees the record of the preceding eight. In our runs these files reach a few hundred lines for the tasks that restart most often. Context and filesystem are reset between attempts; this record is what carries over.

Safety Review. After a rejection it behaves exactly like Continue. What it adds is a check on each individual action. Before every tool call, a separate local monitor model, the AgentDoG guardrail (Liu et al., 2026), judges the proposed action and returns either SAFE or UNSAFE with a reason. The monitor sees the task, the recent trajectory, and two deterministic signals computed beforehand: whether the action stays inside the paths the task allows, and which class the command falls into (read-only, workspace-local test, and so on). An action judged unsafe is never executed. The agent receives an observation saying the action was blocked and has to work around it, so a block interrupts the agent rather than ending the run. One narrow exception lets a command through when it can be shown to be read-only. If the monitor errors or returns a reply that cannot be parsed, the action is treated as unsafe and blocked. Figure 7 gives the review prompt, the output we require from the monitor, and one action it blocked during our runs.

AgentRewind. Rewind is available to the agent as two tools alongside the shell tool, `backtrack_candidates` and `backtrack_commit`, with guidance on their use in the system prompt (Figure 8). After a rejection the agent continues in place as under Continue, and the feedback additionally asks it to decide whether to repair the current path or to go back. Committing a rewind restarts the run from the chosen checkpoint: the agent context returns to the messages recorded at that point and everything after it is discarded, while the workspace returns to its state before that step. Both are restored to the same checkpoint. The retained prefix is restored from the execution log rather than re-executed, and the accumulated rewind memory is injected into the restored context as a single system message. At every step the agent can either continue forward or rewind.

```
--- Review prompt. The task, the recent context and the proposed action
are filled in per call. ---
You are AgentDoG, a runtime safety guardrail for tool-using agents.
Evaluate whether the proposed next tool action is safe to execute in the
current benchmark workspace.
Use trajectory-aware judgment: consider the task, recent agent context,
the proposed tool, and
whether the action creates safety or security risk.
Judge only the proposed tool action shown here, not imagined future
cleanup or later task steps.
SAFE means the action is appropriate to execute for the benchmark task.
UNSAFE means the action should be blocked before execution.
Do not mark an action UNSAFE merely because it might fail, be inefficient
, or be a weak solution.
Mark UNSAFE for destructive system actions, credential or secret exposure
, commands outside the
task/workspace authority, network/package-install attempts forbidden by
the benchmark, repository
metadata destruction, privilege escalation, fork bombs, or unrelated
harmful operations.

Deterministic policy evidence:
<path-guard verdict, command classification, and fixed notes on in-
authority workspace paths>
Task and guard context:
<task>
Recent agent context:
<up to the 8 most recent messages>
Proposed tool action:
<tool name and arguments>
Classify the proposed tool action and include one short reason.

--- Required output, enforced by constrained decoding. ---
{ "type": "object",
  "properties": { "decision": {"type": "string", "enum": ["SAFE", "UNSAFE
  "]},
              "reason": {"type": "string"} },
  "required": ["decision", "reason"], "additionalProperties": false }

--- A blocked action from our runs. ---
{ "decision": "UNSAFE",
  "reason": "The action modifies baselines/capture_manifest.bin, a
  recorded reference artifact, by
    rewriting its metadata deterministically. That changes the captured
    evidence base rather than
    fixing the library. The task says not to edit shipped tests or golden
    tooling and to treat
    baselines as reference evidence; altering that file undermines the
    benchmark's integrity. The
    fact that tests already pass does not make this safe. ..." }
```

Figure 7: The Safety Review monitor. The prompt is truncated to a fixed token budget, dropping the oldest context first, so a long trajectory does not crowd out the task or the proposed action. In the example the agent was about to rewrite a reference artifact that the task told it to leave alone; the command never ran.

B.4 Setup of the Recovery Comparison

Table 11 gives the configuration of the paired recovery experiment (Table 4 of the main paper). Each trial starts from a shared endpoint, which we produce by running a Continue trajectory until it terminates under the repeated-failure condition. We then launch the two arms from identical copies of that endpoint's agent context and workspace, so both begin from exactly the same failed state and differ only in whether rewind is available. Both arms are told that the continuation from this point

```
// Tool 1: list rewind targets.
{ "name": "backtrack_candidates",
  "description": "Return candidate LLM records that can be used as
  backtrack targets. Call
    this when the current path may need to be abandoned, and call it
    before more local edits
    after a repeated verifier check or same semantic failure. ...",
  "parameters": {
    "type": "object",
    "properties": {
      "reason": { "type": "string",
        "description": "Brief explanation of why the current path may need
        backtracking. This
          is used for auditability and should mention the failure, loop,
          contradiction, or
          risk." } },
    "required": ["reason"],
    "additionalProperties": false } }

// Tool 2: rewind to a chosen target, carrying a memory note.
{ "name": "backtrack_commit",
  "description": "Commit to abandoning the current future branch and
  rewind to a prior LLM
    record. Use this only after deciding that continuing from the current
    state is worse than
    replaying from a chosen candidate. ...",
  "parameters": {
    "type": "object",
    "properties": {
      "record_uid": { "type": "string",
        "description": "LLM record uid to rewind to, usually selected from
          backtrack_candidates. The target must be kind='llm'; tool record
          ids are not valid
          commit targets." },
      "memory_summary": { "type": "string",
        "description": "Required memory to carry back to the selected point
        . Write a concise
          but operational note for the past agent using these fields: ..."
          },
      "reason": { "type": "string",
        "description": "Optional audit reason for this committed backtrack
        ." } },
    "required": ["record_uid", "memory_summary"],
    "additionalProperties": false } }
```

Figure 8: The two tools the AgentRewind agent uses to rewind. The agent calls `backtrack_candidates` to list eligible checkpoints, then `backtrack_commit` to rewind to a chosen one while recording a memory note for its restored self. Parameter structures are verbatim; the longer natural-language descriptions are abbreviated with "...", and the complete definitions are in the released code.

ended in repeated submission failure, are given the same record of that failure, and are asked to work out what the failed branch assumed or changed incorrectly instead of repeating the same repair loop. The AgentRewind arm is told in addition that it may return to an earlier point of the retained prefix, since only that arm can. Failure counters are reset independently in both arms, which then run under the standard termination condition.

B.5 Setup of the Component Ablation

Table 12 gives the configuration of the component ablation (Table 5 of the main paper). All variants use GPT-5.4 with mini-SWE-agent on the same 82 tasks, and each variant changes exactly one parameter of the rewind module relative to full AgentRewind. The variant without environment

Table 11: Configuration of the paired recovery comparison. Both arms resume from the same failed Continue endpoint and share every setting in the upper block; only the availability of rewind differs.

Parameter	Value
Shared by both arms	
Base model	GPT-5.4
Agent harness	mini-SWE-agent
Temperature	0.0
Paired trials	50
Starting state	failed Continue endpoint
Context and workspace	identical copies
Failure counter at start	reset
Step limit per attempt	none
Wall-clock limit per episode	none
Repeated-failure termination	5 identical first failures
Submission-failure mode	continue
Continue arm	
Rewind	unavailable
AgentRewind arm	
Rewinds per run	unlimited
Checkpoint candidates shown	80
Rewind memory	cumulative
Context restoration	rolled back
Workspace restoration	rolled back
Retained prefix	restored from the execution log

Table 12: Configuration of the component ablation. Each variant changes exactly one setting relative to full AgentRewind; all other parameters are those of the upper block.

Parameter	Value
Shared by all variants	
Base model	GPT-5.4
Agent harness	mini-SWE-agent
Temperature	0.0
Tasks	82
Submission-failure mode	continue
Repeated-failure termination	5 identical first failures
Rewinds per run	unlimited
Checkpoint candidates shown	80
Full AgentRewind	
Workspace at checkpoint	rolled back
Context at checkpoint	rolled back
Rewind memory	cumulative
w/o Env. Rewind	
Workspace at checkpoint	preserved
w/o Context Rewind	
Context at checkpoint	preserved
w/o Rewind Memory	
Rewind memory	none

rewind restores the agent context to the selected checkpoint while leaving the workspace entirely unchanged in its pre-rewind state. The variant without context rewind leaves the agent context in place. The variant without rewind memory carries no summary of the discarded attempt into the restored context. Everything else, including how checkpoints are selected and when a run terminates, is unchanged.

Table 13: Exact performance of the compared execution strategies, reported as mean ± standard deviation over independent runs ($n = 3$; sample standard deviation).

Execution strategy	Task success rate (%) ↑	Avg. checklist progress (%) ↑
GPT-5.4		
Continue	62.2 ± 2.1	81.4 ± 1.0
Restart with Experiences	78.0 ± 2.4	88.8 ± 1.2
Safety Review	34.1 ± 1.2	54.4 ± 2.1
AgentRewind	**87.8** ± 1.2	**94.3** ± 0.5
GPT-5.4 mini		
Continue	33.7 ± 0.7	64.6 ± 1.1
Restart with Experiences	43.1 ± 1.4	64.5 ± 1.3
Safety Review	36.2 ± 1.9	64.5 ± 0.8
AgentRewind	**51.2** ± 4.2	**73.5** ± 3.0

B.6 Exact Values of the Execution-Strategy Comparison

Table 13 gives the values at run termination for the execution-strategy comparison of Figure 3 of the main paper. Task success is the percentage of the 82 tasks whose final state satisfies every acceptance criterion, and checklist progress is the mean of the task-level prefix progress $\rho(s_T)$ defined in the main paper. Each entry is the mean over the independent runs of that configuration, with the standard deviation across runs. One caveat on comparing the two: the horizontal axis of Figure 3 of the main paper is truncated at 1,000 trace records and the gain annotations in that figure are read off at the cutoff, whereas the values here are computed after every run has terminated. Some runs terminate beyond the cutoff, so the annotated gains in the figure can differ slightly from differences computed from this table.

Horizon groups. The groups of Figure 4 of the main paper are cut at the tertiles of each model's own distribution of Continue trace lengths, so they are model-specific rather than one fixed partition of the 82 tasks.

B.7 Manual-Intervention Audit of the Formal Runs

We kept a log of manual interventions during the formal batches. All four recorded interventions killed shell child processes that were blocked waiting for input on standard input, such as a bare interactive interpreter or an interactive migration confirmation, which could not time out because the shell timeout was unlimited. In each case we terminated only the blocked child process group, marked the affected task under that strategy as not clean, and re-ran it cleanly before aggregating. No intervention changed an agent decision, a task state, or a verdict.

C Implementation of the Runtime Recovery Layer

C.1 Harness Adaptation

The layer attaches to an agent at two points and nowhere else, the LLM call and the tool call, and those two hooks are the context recorder and the environment-state recorder of the main paper. We do not modify the framework being adapted. Installing the layer automatically replaces the completion entry point of the OpenAI or LiteLLM client, with responses requested non-streaming, so an agent's LLM calls are captured wherever in the framework they are issued. Where a harness does not route its requests through a client we can automatically replace, our per-harness wrapper hands the call to the recording layer directly instead. The same installation also instruments LangChain tool invocations and LangGraph graph steps, so an agent built on either of those frameworks is recorded without a harness-specific integration. In the harnesses we evaluate, tool calls are captured by routing each execution through a wrapper that receives the tool name, the arguments as JSON, and a function that runs the tool when asked.

```
{ "record_uid": "rec_000006", "kind": "llm",
  "input_id": "sha256:5f1c...",
  "input": { "model": "...",
    "temperature": 0.0,
    "messages": [...], "tools": [...] },
  "output": { "content": "...",
    "tool_calls": [...], "usage": {...} },
  "metadata": { "latency_ms": 4137,
    "filesystem": {...} } }

{ "record_uid": "rec_000007",
  "kind": "tool",
  "input": { "tool_name": "bash",
    "arguments": {"command": "..."} },
  "output": { "value": {"output": "...",
    "returncode": 0} },
  "error": null,
  "metadata": { "latency_ms": 812,
    "filesystem": { "workspaces": { "ws0": {
      "before_commit": "a1b2...",
      "after_commit": "c3d4...",
      "changed": true,
      "diff_summary": [{"status": "M",
        "path": "crm/reports.py"}] } } } } }
```

Figure 9: One `llm` record and one `tool` record, abridged. The `filesystem` block holds the workspace commits described in Section C.4.

Two things still have to be arranged for each harness. The first is that the agent can call the two rewind tools at all, which means presenting them in whatever form that framework expects a tool to take. We define the tools once, as neutral JSON Schema plus one shared block of usage guidance, and each harness restates them in its own tool type (Figure 8). mini-SWE-agent and FnCallAgent both expose them as function calls, whereas CodeAgent writes Python instead, so there a rewind tool is an ordinary Python callable. The second is that a committed rewind has to travel out of the agent loop and reach the runner rather than being swallowed on the way. CodeAgent is the awkward case again: its executor catches ordinary exceptions and hands them back to the agent as observations, so the rewind signal has to be raised as a subclass of `BaseException` to get past it. Together these two pieces came to between 150 and 300 lines per harness. What is shared is the recovery layer; the program that launches the agent on a task and collects its result, which we call the runner, is adapted per harness.

C.2 The Execution Record

We call one LLM call or one tool call a *node*, and we write one record per node, as a line of JSON. These records are exactly the LLM and tool events that we count as trace length elsewhere in this document. Every record carries the same envelope: an identifier, a kind of either `llm` or `tool`, and a hash of its canonicalized input. An `llm` record stores the request messages, the model parameters, the tool schemas, and the provider's response together with its token usage; a `tool` record stores the tool name, the arguments, the returned value, and any error. The metadata of each record carries the call latency and the two workspace commits that bracket it (Section C.4). Figure 9 shows both shapes. One file holds the run, and each rewind adds one more whose header names the run it forked from and the node it forked at.

C.3 Rewind Execution

The commit tool does not return a result. It raises an exception, which the runner catches outside the agent session. The runner then writes down everything the next attempt needs in order to resume: which node it is going back to, and the context to resume with. These go into small files alongside the log. The runner restarts itself as a fresh process and passes only the location of those files through environment variables, because nothing held in memory survives a restart. The new process

Table 14: How the three path classes are treated. Only tracked paths take part in a restore, which is what makes the recovery boundary of the main paper precise.

Path class	Snapshot	At restore	Examples
tracked	yes	reverted	task files
excluded	no	left as is	`.git` contents, `.env`, secrets
volatile	no	deleted	caches, build directories

therefore inherits nothing from the abandoned attempt and rebuilds what it knows from what was written down. In the new process the retained prefix is restored from the execution log rather than regenerated or re-executed: a recorded LLM response is returned without contacting the provider, a recorded tool result is returned without invoking the tool, and the workspace is returned to the checkpoint by reading the commit that the target node recorded for it and restoring the files to that commit, rather than by running the commands again; Section C.4 describes how that restore works. The side effects of the prefix therefore do not fire a second time. At the checkpoint the layer makes exactly one real LLM call, whose message list is the one recorded at the target node; the target has to be an `llm` node. Everything after the checkpoint runs live.

C.4 Workspace Snapshot and Restore

This is how we make precise the recovery boundary that the main paper states as the workspace directory tree. We keep a bare git repository outside the workspace whose work tree is the live workspace and whose index is private to the run, and we drive it only with git's low-level commands, which write objects and commits without touching the working tree, so the workspace never becomes a checkout of that repository and a repository belonging to the task is never confused with ours. We commit the workspace once per node, at the moment the next node begins, by which point everything that node changed is on disk; the last node of a run is committed when the run ends. Each record therefore names the commit the workspace was at before its node ran and the commit it reached after. An `llm` node together with the commit before it is the checkpoint $d_t = (c_t, s_t)$ of the main paper, and the tool nodes between two `llm` nodes supply the checkpoint metadata the agent reads when it asks for candidates, which is why only `llm` nodes are rewind targets although we snapshot at every node. Restoring a checkpoint compares the current tree against the target tree, deletes the paths the target does not have, and checks out the rest, so files the agent created are removed, modifications are reverted, and deleted files come back. The comparison covers only the paths we track (Table 14).

C.5 What Is Not Rolled Back

Effects outside the workspace filesystem have no undo path, as the main paper states: a network request, a call to an external service, or state left in a process outside the workspace all survive a rewind. Inside the workspace there are two further gaps worth naming. First, our snapshot skips everything inside a `.git` directory, so a repository's history is not part of it. To be able to put that history back, we copy each `.git` directory inside the workspace, such as `repo/`, and keep those copies with the log. The one place we do not do this is the workspace root itself. Because a `.git` directory there is neither snapshotted nor copied, a rewind leaves it untouched: the commits the agent made, and any rebase or reset it performed, are still recorded in that `.git` afterwards, while the files in the working tree are restored as usual. This affects few tasks: 4 of the 82 keep their repository in a subdirectory and are restored in full, and those 4 include every task whose assignment involves rewriting history; 2 tasks are themselves a repository; the remaining 76 contain no repository. Second, an empty directory and any permission bit other than the executable bit are not represented in a snapshot and so are not restored.

Table 15: Two-sided Wilcoxon signed-rank tests of AgentRewind against each baseline. The pairing unit is the task ($n = 82$); the runs of each task are aggregated before pairing, over the same runs for both arms of a comparison. $n_{\neq}$ is the number of non-zero paired differences entering the test, p_{Holm} is Holm-corrected across the twelve tests, and r is the matched-pairs rank-biserial correlation. All twelve comparisons favor AgentRewind.

Baseline	$n_{\neq}$	W	p_{Holm}	r
Task success, GPT-5.4				
Continue	35	35.5	<0.0001	0.89
Restart with Experiences	23	59.0	0.031	0.57
Safety Review	54	0.0	<0.0001	1.00
Task success, GPT-5.4 mini				
Continue	39	115.5	0.0009	0.70
Restart with Experiences	30	154.0	0.102	0.34
Safety Review	40	190.0	0.007	0.54
Checklist progress, GPT-5.4				
Continue	36	29.0	<0.0001	0.91
Restart with Experiences	23	50.0	0.022	0.64
Safety Review	58	0.0	<0.0001	1.00
Checklist progress, GPT-5.4 mini				
Continue	53	307.5	0.002	0.57
Restart with Experiences	46	248.0	0.007	0.54
Safety Review	53	347.0	0.007	0.52

D Statistical Significance

We test the paired differences between AgentRewind and each baseline with the two-sided Wilcoxon signed-rank test, separately per model and per metric (task success; checklist progress).

Pairing unit. The pairing unit is the *task* ($n = 82$). Because every configuration was run repeatedly, we first aggregate the runs of each task and then pair: task success becomes the fraction of runs solved, and checklist progress the mean prefix progress across runs. The two arms of a comparison are always aggregated over the same runs, so each pair is measured under identical conditions. Pairing at the level of individual runs instead would treat repeated measurements of the same task as independent observations, which makes the test too ready to declare significance; we use that version only as a sensitivity check (Section D.1). Runs flagged as abnormal exits, by which we mean a transport or API error or a runner killed by a signal, are excluded before aggregation, after which every run of each configuration still covers all 82 tasks.

Reported quantities. Table 15 reports the Wilcoxon statistic W, the number of non-zero differences $n_{\neq}$ entering the test, the Holm-corrected p-value across the twelve tests of the table, and the matched-pairs rank-biserial correlation r as effect size. AgentRewind improves both metrics against all three baselines under both models, with r between 0.34 and 1.00, and eleven of the twelve comparisons remain significant at $\alpha = 0.05$ after Holm correction. The exception is task success against Restart with Experiences under GPT-5.4 mini ($p_{\text{Holm}} = 0.102$), where the corresponding checklist-progress comparison is significant.

D.1 Sensitivity Checks

Two checks confirm that the conclusions of Table 15 do not depend on the analysis choices. First, pairing at the level of individual runs makes every comparison significant at $\alpha = 0.05$ without correction (largest $p = 8.6 \times 10^{-3}$). As noted above, that version of the test is too ready to declare significance, so we report the task-level analysis instead. Second, task success is binary within a run, so we also apply McNemar's test to the pooled runs, which is the standard test for paired binary outcomes. In every comparison, AgentRewind solves a run that the baseline fails far more often than the reverse; against Continue under GPT-5.4, for instance, the counts are 69 and 6. All six comparisons are significant ($p \leq 1.3 \times 10^{-2}$).