REMEMBERING IS NOT RECOVERING: BENCHMARKING AGENT MEMORY FOR TASK RECOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

In real-world deployments, *large language model* (LLM) agents may encounter unexpected interruptions, such as network outages or machine failures, making reliable task recovery essential. However, existing agent memory benchmarks focus on using past experience to improve future tasks, rather than recovering the state of an interrupted task and resuming it without repeating completed work or skipping necessary steps. To address this gap, we introduce *RecoverMemBench*, which evaluates recovery rates and additional costs relative to uninterrupted runs under two settings: *task resumption*, which preserves progress and tests task state reconstruction, and *task rollback*, which restores an earlier environment state and tests whether memory identifies lost progress while retaining reusable information. Across 11 existing memory systems, average recovery rates remain below 50% in both settings, with over 150% additional steps in the rollback setting, highlighting substantial reliability and efficiency gaps. Our analysis reveals two key limitations: (1) semantic retrieval struggles to identify the information needed for recovery among similar execution records of the same task; and (2) inadequate memory state management causes agents to mistake work undone by rollback for completed work. Motivated by these, we develop a memory system that improves recovery rates by over 35% in both settings while reducing average additional cost by more than half. Overall, our work provides a foundation for evaluating and improving agent memory for reliable and efficient task recovery.

1 INTRODUCTION

Large language model (LLM) agents increasingly tackle tasks that require hundreds of sequential actions and interactions with environments (Anthropic, 2025; OpenAI, 2026). Such extended operation inevitably can be interrupted by unexpected issues (e.g., infrastructure failures), resulting in unfinished tasks (Balakrishnan et al., 2026). Recovering these interrupted tasks requires agents to reconstruct their task states; otherwise, repeating completed work or skipping necessary steps could incur huge additional costs or even task failure. A direct way is to input the full operation history into agents, but their limited context windows and noise in that history make them struggle to reconstruct the exact task states for recovering interrupted tasks. Therefore, reliable task recovery requires the memory system to provide reusable task states consistent with the external environment.

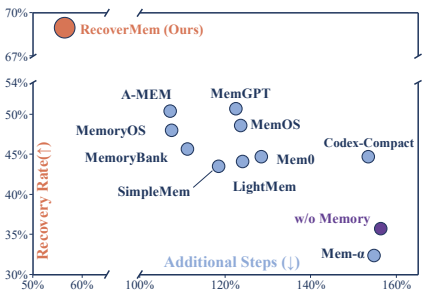

Figure 1: Average performance of existing memory methods on RecoverMemBench. Existing methods perform unreliable task recovery and require notably more steps than uninterrupted runs.

However, current agent memory benchmarks primarily focus on how well memory systems retain knowledge from past interactions to benefit future tasks. For instance, LoCoMo (Maharana et al., 2024) and LongMemEval (Wu et al., 2025) evaluate memory using question-answering tasks based on conversation history, while MemoryArena (He et al., 2026) assesses memory retention for subsequent, interdependent subtasks during agent interaction. While these benchmarks effectively measure how accumulated knowledge improves new tasks, they

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.

overlook a critical real-world scenario: interrupted tasks. Specifically, they fail to evaluate the ability of memory systems to preserve the exact task state for recovery when ongoing tasks are interrupted.

To address this gap, we propose a new benchmark, *RecoverMemBench*, which introduces interruptions during agent execution to evaluate the ability of current memory systems in task recovery. Specifically, RecoverMemBench includes two settings that require different recovery requirements: (1) *task resumption* interrupts the agent operation and stores the completed progress in the environment, testing whether memory systems can provide the task state for recovery; (2) *task rollback* restores the environment to an earlier step after the agent fails due to errors. This requires the memory system to distinguish lost progress after rollback while retaining exact states, ensuring the memory-environment match for recovery. Additionally, we build our benchmark with 416 samples from SWE-bench (Jimenez et al., 2023), using recovery rate and additional cost as our metrics.

With RecoverMemBench, we conduct extensive experiments with 11 existing memory systems (Xu et al., 2025; Chhikara et al., 2025; OpenAI, 2025) for evaluating their capabilities in task recovery. Our results show notable gaps in both reliability and efficiency (Figure 1): the baseline methods, in task resumption, only achieve an average recovery rate of 45.07% while requiring 126.15% additional steps. Meanwhile, in task rollback, their average recovery rate further degrades to 31.55%, while the required additional steps increase to 168.66%. Our analysis suggests two critical limitations. First, memories from different steps of the same task can be semantically similar while describing different task states, making semantic retrieval unreliable for selecting the exact state needed for recovery. Second, overwrite updates in the existing methods leave information from later steps accessible after rollback, creating inconsistencies between memory and the environment and treating progress lost after rollback as completed work. These findings highlight the need to retrieve step-specific memories and to ensure the memory state remains consistent with the environment.

To address these limitations, we introduce a lightweight memory system, *RecoverMem*, that records task state as a structured entry at every execution step. By employing step-specific retrieval, the system enables the agent to access the precise recorded task state required for resumption or rollback. Furthermore, it maintains independent snapshots of these records instead of the current methods' overwrite memory updates, ensuring that the memory perfectly synchronizes with the environment during rollbacks. Inspired by Reflexion (Shinn et al., 2023), we incorporate an experience module that records past failures, enabling agents to avoid repeating mistakes after rollback. Evaluated on RecoverMemBench, our method achieves recovery rates of 68.90% and 43.27% for task resumption and rollback, respectively, outperforming average baselines by 52.87% and 37.13%. Furthermore, it requires on average only 64.40% additional steps, reducing the average additional cost by 56.30%.

Overall, the contributions of our work are as follows:

- We introduce RecoverMemBench, a novel benchmark designed to rigorously evaluate a critical yet underexplored capability of agent memory: task recovery. RecoverMemBench includes two settings (task resumption and task rollback) for different recovery requirements to test memory systems under complex, real-world interruption scenarios.
- We reveal critical limitations in existing memory systems. Through a comprehensive evaluation of 11 memory systems, we show severe gaps in both reliability and efficiency (Section 3.1). Our analysis uncovers two fundamental limitations causing these failures: unreliable semantic retrieval and memory-environment state mismatch (Section 3.2).
- We propose RecoverMem, a lightweight step-specific memory system, to directly mitigate these limitations (Section 4.1). Evaluated on RecoverMemBench, our method improves recovery rates by 52.87% and 37.13% over baselines on resumption and rollback respectively, while simultaneously reducing the average additional recovery cost by 56.30%.

2 RecoverMemBench

RecoverMemBench evaluates a crucial but overlooked capability: whether a memory system enables an agent to complete an interrupted task without repeating completed work or skipping necessary steps. Section 2.1 formalizes task recovery, Section 2.2 defines the two recovery settings, Section 2.3 details how we instantiate these settings, and Section 2.4 defines the evaluation metrics.

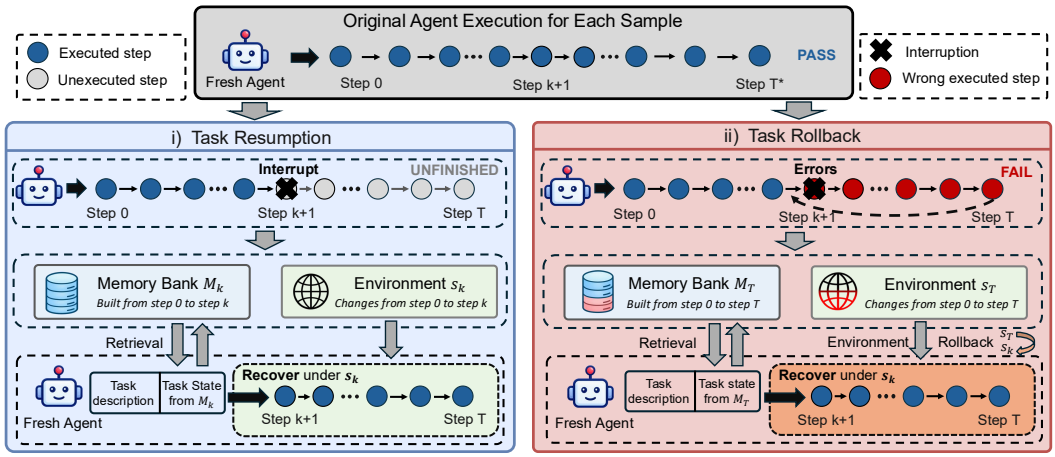

Figure 2: Overview of task resumption setting and task rollback setting of RecoverMemBench.

2.1 Task Recovery Formulation

Execution without Interruption. Without interruption, agents execute a task q through a continuous sequence of interactions with an environment, starting from an initial environment state s_0. Formally, at step t, the agent takes an action a_t in environment state s_{t-1}, receives an observation o_t, and leaves the environment in s_t. Alongside execution, a memory system maintains a memory state $\mathcal{M}_t$, starting from an initial memory state $\mathcal{M}_0$, through Update (Liao et al., 2026):

$$\mathcal{M}_t = \text{Update}(\mathcal{M}_{t-1}, a_t, o_t), \quad t \geq 1, \tag{1}$$

where the memory system has access to q throughout; we omit q from Update for brevity. The form of $\mathcal{M}_t$, and hence what it retains from past steps, is method-specific, e.g., a single summary overwritten at every step or a collection of entries stored over time. To utilize the memory, Read operation provides an agent acting from the environment state with the memory entries

$$m = \text{Read}(\mathcal{M}), \tag{2}$$

where $\mathcal{M}$ is the memory state available at the time of reading. A common implementation of $\text{Read}(\cdot)$ selects stored entries relevant to q via semantic retrieval (Zhong et al., 2024) (e.g., vector-based retrieval). We refer to the execution without interruption as the uninterrupted run, which takes T^* steps and produces an outcome $y^* \in \{0, 1\}$, where $y^* = 1$ indicates successful task completion.

Execution after Interruption. When execution is interrupted after step k, the agent loses its context, including its retained interaction history, while the environment state s_k and memory state $\mathcal{M}_k$ remain stored outside the agent process. Then, task recovery starts a fresh agent on the same task q with s_k and $\mathcal{M}$, determined by the recovery setting. The agent receives q and $m = \text{Read}(\mathcal{M}, q)$ in its initial context and obtains further observations through environment interactions. To continue correctly, the agent needs to identify which previous observations and state changes still hold and which work remains unfinished. Therefore, the memory system must provide the task state with respect to s_k, i.e., the work already completed in s_k and the subtasks that remain.

2.2 Recovery Settings

Recovering a task requires memory to provide environment state information from which execution resumes. When an external disruption, such as a network outage, interrupts execution, completed progress remains intact, and memory needs to help the agent identify where it left off. However, when an agent's incorrect actions lead to failure, simply continuing from that point may preserve the consequences of those mistakes. Recovery may instead require rolling back the environment to an earlier state. In this case, remembering what happened is insufficient: memory must distinguish progress that still holds from work that has been undone, while retaining useful lessons from the failed attempt to prevent repetitive exploration. RecoverMemBench captures these different memory requirements through two settings: *task resumption* and *task rollback*, as shown in Figure 2.

Task resumption. We interrupt the agent after step k and preserve both the environment s_k and memory $\mathcal{M}_k$, so recovery begins with

$$(s, \mathcal{M}) = (s_k, \mathcal{M}_k). \tag{3}$$

The recovered agent receives $\mathcal{M}_k$, which describes completed work and pending steps with respect to s_k, and continues from the changes already present in s_k. This setting tests whether the memory provides the task state at the interruption step after the agent loses its context.

Task rollback. In this setting, we highlight a more complex scenario, where we need to recover a failed task from any execution steps. Specifically, an erroneous action is taken by the agent at step $k+1$, and the resulting attempt continues to run until it terminates in failure at step T. Meanwhile, the memory system continuously updates to record this flawed execution path, resulting in $\mathcal{M}_T$, as shown in Figure 2. To recover from the failure, we rollback the environment to s_k by undoing the flawed changes made by the agent after the erroneous action (from $k+1$ to T). The agent is subsequently restarted with its memory retained at $\mathcal{M}_T$. Thus, immediately after rollback,

$$(s, \mathcal{M}) = (s_k, \mathcal{M}_T), \qquad k < T. \tag{4}$$

This creates a clear inconsistency: the environment reflects the work the agent has done only through step k, but the memory $\mathcal{M}_T$ still describes changes and decisions from steps $k+1$ through T. Therefore, the memory system must distinguish the work already reflected in s_k from the work performed after step k whose effects have been undone after task rollback. Without this distinction, later records in the memory system can be treated as current progress or valid continuation plans, introducing future leakage relative to s_k. The recovered agent may consequently assume that the undone changes still exist or follow the same paths that led to the failure.

2.3 Benchmark Setup

Dataset. To achieve the two settings, we build our benchmark from SWE-bench (Jimenez et al., 2023), whose tasks involve resolving issues in GitHub repositories, allowing us to use Git to restore earlier operation environment states. RecoverMemBench contains 416 evaluation samples: 312 for task resumption and 104 for task rollback. To eliminate the influence of the agents' base capability, we run mini-SWE-agent (Yang et al., 2024) with GPT-4o (Hurst et al., 2024) on these tasks and select 104 tasks that the agent successfully completes three times without interruption.

For task resumption, we construct three types of samples by setting the interruption step k to 25%, 50%, and 75% of T^*, where T^* denotes the length of the successful execution. Each sample contains the task description, the complete sequence of actions and observations through step k, and the environment state at that step. This procedure yields 312 samples spanning three stages of execution, with different amounts of history available to the agent at resumption.

For task rollback, to simulate failed agent operations, we inject an error immediately after a randomly selected step k across the 104 tasks and allow the agent to continue until execution terminates in failure. We record the resulting trajectory to capture the failure sequence and memory, and then use Git to restore the environment to s_k. Each of the resulting 104 samples thus contains the task description, the execution history, and the restored environment state s_k.

Recovery Protocol. In both settings, the recovery process works in the saved environment state s_k and evaluates a fresh mini-SWE-agent. Without retaining the original agent context, this new agent is initialized with the task description q and the retrieved memory via $\mathrm{Read}(\mathcal{M}, q)$. The two settings differ in the trajectory scope used to build this memory m: (1) Task resumption: The memory state is built using the trajectory from step 0 to k ($\mathcal{M}_k$), capturing the execution only up to the interruption step. (2) Task rollback: The memory state is built using the complete trajectory from step 0 to T ($\mathcal{M}_T$). This setting directly aligns with how memory systems function in practice. Appendix A provides additional dataset statistics and implementation details.

2.4 Evaluation Metrics

We evaluate recovery success and agent execution step overhead. For each sample, we compare the success rate and total execution steps with the corresponding successful uninterrupted run. Both metrics are calculated over an evaluated set of N recovery samples and reported as percentages.

Table 1: Evaluation results of existing memory methods and RecoverMem (Ours) on RecoverMem-Bench. Avg. reports the mean over the three resumption settings. **Bold** numbers indicate our model notably outperforms the strongest baseline.

	Resumption@25%		Resumption@50%		Resumption@75%		Avg.		Rollback	
Method	RR ↑	AS ↓	RR ↑	AS ↓	RR ↑	AS ↓	RR ↑	AS ↓	RR ↑	AS ↓
w/o Memory	22.11	182.16	38.47	151.59	44.23	141.37	34.94	158.37	25.96	263.53
MemGPT	14.41	139.35	67.31	131.46	71.16	94.03	50.91	121.61	37.50	158.49
Mem0	17.31	128.28	50.97	156.78	65.39	99.21	44.56	128.09	26.92	179.42
A-MEM	21.15	151.38	65.38	110.43	65.39	52.88	50.62	104.90	38.46	126.12
MemOS	22.11	133.33	62.50	142.53	61.54	92.45	48.72	122.77	25.96	185.11
Codex-Compact	19.23	194.24	54.81	221.73	59.62	49.93	44.55	155.30	25.96	151.73
MemoryOS	22.11	126.83	62.50	103.86	59.62	84.96	48.08	105.22	39.42	127.34
SimpleMem	23.08	127.55	50.00	150.00	56.73	73.88	43.27	117.14	25.96	220.14
LightMem	19.23	98.12	58.66	107.46	53.85	164.17	43.91	123.25	30.77	131.51
MemoryBank	18.26	158.99	64.43	122.60	53.85	46.33	45.51	109.31	35.58	157.19
Mem-α	19.23	177.34	29.81	227.60	45.19	65.04	31.41	156.66	31.73	226.47
BeliefMem	18.26	163.10	52.88	189.46	61.54	77.69	44.23	143.42	28.84	191.72
RecoverMem (Ours)	**49.03**	**57.55**	**70.20**	**83.88**	**87.50**	**26.83**	**68.90**	**56.09**	**43.27**	**89.35**

Recovery rate (RR). RR is the proportion of samples in which the agent successfully completes the task after recovery:

$$\mathrm{RR} = \frac{1}{N}\sum_{i=1}^{N} y_i, \tag{5}$$

where $y_i = 1$ if the recovered agent's solution passes the benchmark tests and $y_i = 0$ otherwise. Since all reference runs succeed, RR measures success retention after interruption or rollback.

Additional steps (AS). AS measures the mean relative change in total agent execution steps compared with the uninterrupted reference runs:

$$\mathrm{AS} = \frac{1}{N}\sum_{i=1}^{N} \frac{C_i - T_i^*}{T_i^*}, \tag{6}$$

where T_i^* is the number of steps in the successful uninterrupted run and C_i is the total number of steps executed before and after recovery. Let R_i denote the number of steps executed after restart. For both task resumption and task rollback, $C_i = k_i + R_i$, where k_i is the interruption step. Positive values indicate more execution steps than the interrupted runs, whereas negative values indicate fewer steps on average relative to those runs. Because fewer steps may also result from early failure, AS must be interpreted together with RR for reliable evaluation of task recovery.

Generally, RecoverMemBench evaluates whether memory systems enable agents to recover tasks after two different types of interruption. Task resumption tests reconstruction of task state at different interruption steps, while task rollback additionally requires distinguishing progress present in the restored environment and changes removed by rollback.

3 Evaluation and Analysis of Existing Memory Systems

Setups. We conduct all experiments with mini-SWE-agent (Yang et al., 2024), using GPT-4o (Hurst et al., 2024) for task execution and GPT-4o-mini (Hurst et al., 2024) as the backbone model for all memory systems. The default embedding model used in memory systems is text-embedding-3-small (OpenAI, 2024). Additionally, the memory budget for each method is set to 4,096 tokens. More configuration details are provided in Appendix B.1.

Baselines. We evaluate 11 memory systems: MemGPT (Packer et al., 2023), MemoryBank (Zhong et al., 2024), Mem0 (Chhikara et al., 2025), A-MEM (Xu et al., 2025), MemOS (Li et al., 2025), MemoryOS (Kang et al., 2025), SimpleMem (Liu et al., 2026), LightMem (Fang et al., 2025), Mem-α (Wang et al., 2025), and BeliefMem (Liao et al., 2026). Our evaluated methods also include Codex-Compact (OpenAI, 2025), which utilizes prompt summarization (Appendix B.4) for compacting working context in Codex agent. These methods cover current memory systems with diverse architectures, and details about them are provided in Section 5. Additionally, we also conduct experiments without any memory (*w/o Memory*), where we recover tasks without any memory.

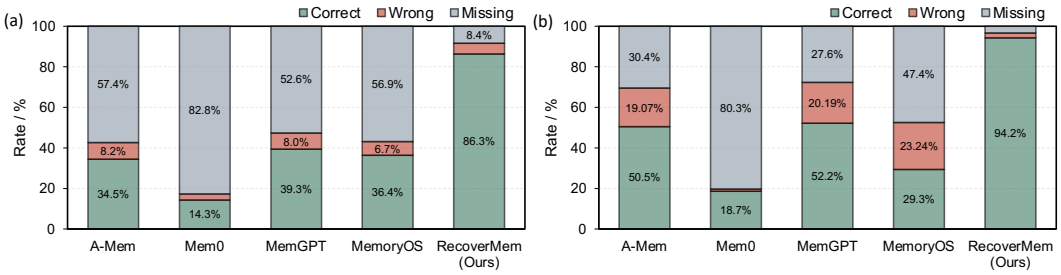

Figure 3: Comparison of memory retrieval quality. The distribution of Correct, Wrong, and Missing task states in retrieved memory entries across baseline methods and our approach. (a) Evaluation under the task resumption@50% setting. (b) Evaluation under the task rollback setting.

3.1 Main Results

Table 1 reports the performance of several current memory systems on RecoverMemBench, including RR and AS in resumption@25% / 50% / 75% and task rollback settings.

Existing methods struggle to recover the interrupted tasks. Although the same agent successfully completes these tasks without interruption, the 11 memory baselines achieve an average RR of only 45.07% under task resumption. Even the strongest baseline MemGPT, which can preserve the original history during execution, recovers only 50.91% of interrupted tasks when averaged across the three interruption settings. Similarly, MemoryOS achieves an RR of only 48.08%, despite incorporating dedicated memory for ongoing agent work. This limited recovery persists at later interruption steps: average RR increases from 19.49% to 59.44% (from resumption@25% to resumption@75%), yet over 40% of tasks remain unrecovered on average even at the latest interruption. This unreliability is more noticeable under task rollback, where the average RR falls to 31.55% and even the best baseline, MemoryOS, recovers only 39.42% of tasks.

Current memory systems fail to achieve efficient task recovery. Compared with uninterrupted runs, the baselines incur an average AS of 126.15% under task resumption. On average, the additional steps alone exceed the steps required to run entire task from scratch without interruption. Even A-MEM, the most efficient baseline, incurs 104.90% AS, so no reported baseline keeps average AS below 100% in this setting. Under task rollback, average AS rises to 168.66%, and even the most efficient baseline, A-MEM, incurs 126.12% AS. Codex-Compact reduces rollback AS from 263.53% for w/o Memory to 151.73%, but both methods achieve only 25.96% RR.

3.2 Analysis of Recovery Failures

Section 3.1 shows that existing memory systems achieve limited RR while incurring substantial AS on RecoverMemBench. We investigate these failures by testing whether the memory retrieved during task recovery contains the accurate task states at the environment state interruptions occur.

Setup. In this section, we conduct experiments under task resumption@50% and task rollback. For both settings, we use DeepSeek-v4.1-flash (DeepSeek-AI, 2026) to generate questions about the environment state at interruption, together with reference answers derived from that state (examples of questions and reference answers are provided in Appendix B.3). For each baseline, GPT-4o answers these questions using the memory that the baseline retrieved during the corresponding recovery experiment in Table 1. We reuse these retrieved entries without retrieving additional memory for the questions. Then, we verify the answers by checking them in the corresponding environment with verification code and label them as *Correct*, *Wrong*, or *Missing* (Appendix B.2 provides the details of the verification code). This evaluation tests whether the retrieved memory entries can provide the exact task states for recovery. More experiment details are provided in Appendix B.2.

3.2.1 Analysis for task resumption

Retrieved memory often lacks the task state needed for resumption. As shown in Figure 3 (a), when using their respective retrieved memories, the baselines exhibit Missing rates ranging from 52.6% to 82.8% and Correct rates ranging from 14.3% to 39.3%. Even MemGPT, which achieves

the highest Correct rate, leaves 52.6% Missing and 8.0% Wrong. Thus, the memory retrieved during task resumption frequently cannot provide exact information about task states.

Required task states are often stored but not retrieved. To determine whether these failures stem from missing information or ineffective retrieval, we query each baseline's full memory bank using questions about the interruption state. Specifically, we use each question about the interruption state as a semantic query over the full bank and provide the resulting entries to GPT-4o as context (Full Memory Bank). The questions and answer model remain unchanged. As presented in Figure 4, retrieving context from the entire memory bank improves Correct rate across all four baselines, increasing it from 34.95% to 60.40% for A-Mem and from 37.34% to 64.75% for MemoryOS. These improvements confirm that current memory systems already preserve the critical task states required for task resumption. Therefore, task recovery failures mainly stem from inadequate retrieval mechanisms of existing memory systems rather than the actual absence of stored information.

This limitation arises because the memory entries recorded in different execution steps refer to the same task objects, such as file names, but describe different stages of execution. Since the task query q remains unchanged throughout execution, semantic similarity to q alone cannot reliably identify which entry captures the task states at the interruption step. As a result, the retrieval often obtains semantically similar but unuseful task states. Therefore, the retrieval must specify what has been completed and what remains to be done at the interruption step.

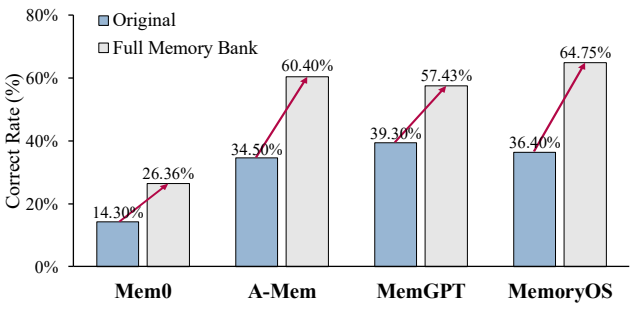

Figure 4: Correct rate of questions about task states with original retrieved memory (Original) and retrieved memory from full memory bank (Full Memory Bank).

3.2.2 Analysis for Task Rollback

Memory fails to distinguish valid progress from changes undone by rollback. Beyond missing information, the rollback setting exposes errors in retrieved memory: Figure 3 (b) reports Wrong rates of 19.07%, 20.19%, and 23.24% for A-Mem, MemGPT, and MemoryOS, respectively. Due to the baseline's overwrite update, the memory is unconditionally preserved at step T ($\mathcal{M}_T$) during rollback to s_k, causing records from $k+1$ to T to remain accessible even though their associated environment changes have been undone. Therefore, these records can describe completed work that is absent from the restored environment. For example, memory may report that an edit has been applied, causing the agent to skip an edit that the rollback has removed. Therefore, recovery must distinguish the work still present in s_k from the later execution history. In Appendix C.1, we evaluate the baselines' performance when preserving local memory at each step to align $\mathcal{M}_k$ with s_k, showing notable performance gains but huge storage overhead.

4 RecoverMem

In Section 3, we evaluate existing memory methods on RecoverMemBench and analyze their limitations in semantic retrieval and inadequate memory state management. To address these limitations, we propose *RecoverMem*, a lightweight step-specific memory system which preserves the task state at each execution step and indexes memory entries by steps.

4.1 Method Design

Memory construction. RecoverMem instantiates the memory state as a collection of step-indexed entries. It constructs the first entry m_1 by summarizing the task state from action a_1 and observation o_1 (details shown in Appendix B.5). At each subsequent step, it updates the previous entry using the current action and observation:

$$m_t = \text{Update}(m_{t-1}, a_t, o_t), \tag{7}$$

Specifically, m_{t-1} records completed subtasks, remaining subgoals, and the planned next action, together with prior attempts and their observed evidence. Using this record as context, RecoverMem assesses the outcome of a_t based on o_t, updates subtask completion status, and revises the remaining work and next action accordingly. The updated entry m_t keeps the accumulated task state while incorporating the outcome of the current step. Unsuccessful attempts are retained with their observed outcomes to guide recovery and reduce redundant attempts. Finally, RecoverMem stores every m_t locally under its step index t and retains earlier entries in memory state $\mathcal{M}_t = \{m_0, \ldots, m_t\}$, which preserves a separate memory for each step for retrieving the task state recorded at that step. Additionally, despite retaining these entries in memory, RecoverMem uses less total storage than some evaluated baselines, as shown in Appendix C.2.

Retrieval by step. Prior methods that continually overwrite a single memory retain only the latest task state, preventing retrieval of the memory recorded at an earlier step. RecoverMem preserves each memory before updating it and stores the updated memory separately under its step index. Specifically, for any step t, RecoverMem uses the step index to retrieve from the memory $\mathcal{M}_T$:

$$m_t = \mathrm{Read}(\mathcal{M}_T), \tag{8}$$

where we use retrieving the memory entries by step index for $\mathrm{Read}(\cdot)$. This directly utilizes the entries preserved during memory construction: each retrieval returns the memory recorded after the corresponding a_t and o_t. For task rollback, RecoverMem can reflect on the stored memory entries from m_{k+1} to m_T to update the experience module in $\tilde{m}_k = \mathrm{Reflection}\big(m_k, \{m_{k+1}, \ldots, m_T\}\big)$, where $m_k = \mathrm{Read}(\mathcal{M}_T)$, where $\mathrm{Reflection}$ applies a strategy inspired by Shinn et al. (2023). We also apply this step-specific retrieval in baseline methods, showing notable performance improvement but huge storage costs in Appendix C.1.

In conclusion, by using the environment's recovery step as the retrieval key, RecoverMem inherently prevents state mismatch. It ensures the agent is provided with the exact recorded progress, remaining work, and failed experience corresponding to the step from which execution resumes.

4.2 Evaluation Results

RecoverMem notably improves RR while reducing AS. Table 1 shows that RecoverMem achieves the highest RR and lowest AS among the reported methods across all recovery settings. Concretely, its RR reaches 68.90% on average under task resumption and 43.27% under task rollback, corresponding to relative improvements of 35.33% over MemGPT and 9.77% over MemoryOS, the strongest baselines in the respective settings. These gains accompany lower AS: compared with A-MEM, the most efficient baseline in both settings, RecoverMem reduces AS by 46.53% to 56.09% under resumption and by 29.15% to 89.35% under rollback. RecoverMem is the only reported method with average AS below 100% in both recovery settings.

RecoverMem retrieves more reliable task state for task recovery. In the task resumption setting, RecoverMem achieves a correct rate of 86.3%, exceeding the strongest baseline MemGPT by 47.0 pp (Figure 3 (a)). It also reduces the Missing rate from 52.6% to 8.4%, providing substantially more information required for task recovery. In the rollback setting, RecoverMem achieves a correct rate of 94.2%, compared with 52.2% for MemGPT, an improvement of 42.0 pp (Figure 3 (b)). This higher correct rate indicates that the retrieved memory more accurately reflects the restored environment despite the retention of records that describe subsequent execution.

4.3 Ablation Studies

We evaluate the contributions of $\mathrm{Reflection}$ and step-specific retrieval in the setting of task rollback. We compare RecoverMem with two ablated variants: removing failure experiences from memory (w/o $\mathrm{Reflection}$), and replacing step-specific retrieval with random retrieval (w/o step-specific retrieval).

Table 2: Results of ablation studies of RecoverMem on the setting of task rollback.

Method	RR ↑	AS ↓
w/o $\mathrm{Reflection}$	38.46%	117.62%
w/o step-specific retrieval	26.92%	161.28%
RecoverMem(Ours)	43.27%	89.35%

As shown in Table 2, w/o $\mathrm{Reflection}$ reduces RR from 43.27% to 38.46% and increases AS from 89.35% to 117.62%. These results are consistent with intuition that accessing previous failures helps

agents avoid repeating unsuccessful actions. Additionally, replacing step-specific retrieval with random retrieval causes a greater degradation in both metrics: RR drops to 26.92%, and AS increases to 161.28%. Compared to the full method, these changes correspond to a 16.35% decrease in RR and an 80.50% increase in AS. These results further validate that retrieving information relevant to the current recovery step is important for successful and efficient task recovery.

Overall, RecoverMem preserves task states at each step and retrieves the entries corresponding to the recovery step. Under the same memory budget as the baselines, it achieves higher RR with fewer AS in both settings, providing a reference method for developing memory systems that support interrupted task recovery. Discussion ob limitations and future directions is provided in Appendix D.

5 Related Works

Agent memory benchmarks. Existing benchmarks primarily evaluate memory methods through task performance under uninterrupted execution. LoCoMo (Maharana et al., 2024) and LongMemEval (Wu et al., 2025) utilize question-answering, summarization, and dialogue generation over previous conversation sessions to benchmark a memory method's ability to recall facts, update knowledge, and reason across time. MemoryAgentBench (Hu et al., 2025) extends the evaluation to incrementally supplied text streams, assessing continuous learning and retrieval while uniquely testing the capacity of memory systems for selective forgetting when facts are revised. Additionally, MemoryArena (He et al., 2026) and LongMemEval-V2 (Wu et al., 2026a) move the evaluation from a single-turn setting to multi-turn agent interaction, evaluating memory systems by testing their ability to retain and apply historical context in interdependent subtasks, as well as recall environment states, action effects, and workflows. However, these protocols do not explicitly interrupt ongoing tasks to evaluate the task recovery ability of memory systems, making it underexplored.

Task recovery. Existing work studies task recovery by resuming interrupted execution and correcting failed attempts. LogAct (Balakrishnan et al., 2026) and Crab (Wu et al., 2026b) investigate recovery from persistent execution histories and saved system state, while DART (Yang et al., 2026) examines whether rollback preserves valid progress. AgentTether (Zhao et al., 2026) studies how the experience of failed runs supports subsequent repair. These studies primarily evaluate recovery strategies, leaving it unclear how different memory systems support recovery after context loss or environment rollback. Our benchmark addresses this gap by comparing memory systems under task resumption and rollback, measuring recovery success and additional steps.

Agent memory systems. Recent advancements in agent memory can be broadly classified by how they process and organize historical context. While summarization-based approaches (Zhong et al., 2024; Xu et al., 2025; Chhikara et al., 2025; Fang et al., 2025; Wang et al., 2025; Liu et al., 2026; Liao et al., 2026) focus on retaining factual records, reflection-based methods (Wang et al., 2023; Shinn et al., 2023; Park et al., 2023; Zhao et al., 2024) store these experiences into reusable lessons for future tasks. Additionally, frameworks like MemGPT (Packer et al., 2023) and MemOS (Li et al., 2025; Kang et al., 2025) introduce system management for persistent memory. Despite this diversity, it remains unclear how robustly these systems handle unexpected disruptions. Therefore, our work specifically benchmarks these methods in their capacity for interrupted task recovery.

6 Conclusion

We introduce RecoverMemBench to evaluate how memory systems support recovery from interrupted tasks. The benchmark covers two settings with distinct recovery requirements. In task resumption, a fresh agent continues an interrupted task using memory while the environment retains changes made before interruption. In task rollback, the environment rolls back prior changes while memory may still record the corresponding actions as completed, requiring the agent to know the discrepancies between memory and the current environment. Our evaluation of 11 memory methods reveals limitations in both recovery success rate and efficiency, and uncovers two failure modes: semantic retrieval fails to query the exact task state interrupted tasks require, and overwrite memory updates can retain records of actions whose effects have been undone. To mitigate them, we propose RecoverMem, which records task state at each execution step and retrieves memory entries by step. Compared with current methods, RecoverMem improves recovery rates by over 35% in both settings while reducing average additional recovery cost by more than half. Our work identifies task recovery limitations in existing memory systems and provides a benchmark for future research.

AI USE STATEMENT

In this work, we used Deepseek-v4.1-flash to generate the QA dataset based on our benchmark in Section 3.2. We have not used generative AI tools to optimize the idea of the work and/or other other core methodological work. Additionally, we used GPT-6-Astra-Chat for writing refinements and codebase checks. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Anthropic. Effective harnesses for long-running agents. Anthropic Engineering Blog, 11 2025. URL https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

Mahesh Balakrishnan, Ashwin Bharambe, Davide Testuggine, David Geraghty, David Mao, Vidhya Venkat, Ilya Mironov, Rithesh Baradi, Gayathri Aiyer, and Victoria Dudin. Logact: Enabling agentic reliability via shared logs. *arXiv preprint arXiv:2604.07988*, 2026.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Technical Report, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/blob/main/DeepSeek_V41_Tech_Report.pdf.

Jizhan Fang, Xinle Deng, Hao-Ming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. Lightmem: Lightweight and efficient memory-augmented generation. *ArXiv*, abs/2510.18866, 2025. URL https://api.semanticscholar.org/CorpusID:282245881.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. *ArXiv*, abs/2507.05257, 2025. URL https://api.semanticscholar.org/CorpusID:280136659.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shun-Yu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770, 2023. URL https://api.semanticscholar.org/CorpusID:263829697.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25961–25970, 2025.

Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, et al. Memos: A memory os for ai system. *arXiv preprint arXiv:2507.03724*, 2025.

Junfeng Liao, Qizhou Wang, Jianing Zhu, Bo Du, Rui Yan, and Xiuying Chen. Belief memory: Agent memory under partial observability. *arXiv preprint arXiv:2605.05583*, 2026.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *ArXiv*, abs/2601.02553, 2026. URL https://api.semanticscholar.org/CorpusID:284512931.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024.

OpenAI. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/, 2024.

OpenAI. Codex CLI: Lightweight coding agent that runs in your terminal. GitHub repository, https://github.com/openai/codex, 2025.

OpenAI. Run long horizon tasks with codex. OpenAI Developers Blog, 2 2026. URL https://developers.openai.com/blog/run-long-horizon-tasks-with-codex.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Yu Wang, Ryuichi Takanobu, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. Mem-{\alpha}: Learning memory construction via reinforcement learning. *arXiv preprint arXiv:2509.25911*, 2025.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=pZiyCaVuti.

Di Wu, Zixiang Ji, Asmi Kawatkar, Bryan P. Kwan, Jia-Chen Gu, Nanyun Peng, and Kai-Wei Chang. Longmemeval-v2: Evaluating long-term agent memory toward experienced colleagues. *ArXiv*, abs/2605.12493, 2026a. URL https://api.semanticscholar.org/CorpusID:288256230.

Tianyuan Wu, Chaokun Chang, Lunxi Cao, Wei Gao, and Wei Wang. Crab: A semantics-aware checkpoint/restore runtime for agent sandboxes. *ArXiv*, abs/2604.28138, 2026b. URL https://api.semanticscholar.org/CorpusID:287915371.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Ke Yang, Panpan Li, Zonghan Wu, Kejin Xu, Huaxi Huang, and Xiaoshui Huang. DART: Semantic recoverability for structured tool agents. *arXiv preprint arXiv:2605.23311*, 2026.

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.

Chenyu Zhao, Shenglin Zhang, Wenwei Gu, Yongqian Sun, Dan Pei, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. Agenttether: Graph-guided diagnosis and runtime intervention for reliable llm agent operation. *ArXiv*, abs/2607.06273, 2026. URL https://api.semanticscholar.org/CorpusID:289941632.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

Table 3: Benchmark statistics. The four settings contain 416 evaluation cases derived from 104 distinct tasks.

Setting	Counts	Mean of k	Range of k	Mean remaining steps
Task Resumption@25%	104	3.13	[1, 11]	10.77
Task Resumption@50%	104	6.71	[3, 22]	7.18
Task Resumption@75%	104	10.05	[4, 33]	3.85
Task Rollback	104	5.36	[2, 25]	N/A

A Benchmark Details

A.1 Collect Dataset on SWE-bench

In our work, we investigate the task recovery ability of memory systems. To comprehensively evaluate this ability, we need to eliminate the influence of the agents' base capability. Therefore, we collected trajectories of tasks that agents can successfully handle three times. Consequently, whether agents can handle these tasks after recovery reflects the capability of the memory systems. Specifically, we utilize min-SWE-agent (Yang et al., 2024) with GPT-4o (Hurst et al., 2024) to collect trajectories of 104 successful samples as our base dataset.

A.2 Statistical Data

Task resumption@25%. We construct the benchmark from 104 successfully completed task trajectories across 10 code repositories on SWE-bench (Jimenez et al., 2023). These trajectories contain 1,445 agent steps, with a mean of 13.89 steps, a median of 11, and a range of 6 to 45. For task resumption@25% setting, we interrupt each trajectory after the first 25% of its total steps, rounding down and requiring at least one completed step and one remaining step. This produces 104 resumption samples. The observed prefix contains 3.13 steps on average, ranging from 1 to 11, while the original successful trajectory has 10.77 steps remaining on average.

Task resumption@50%. We apply the same interruption to the same 104 trajectories at the 50% boundary, producing another 104 resumption samples. The observed prefix contains 6.71 steps on average, ranging from 3 to 22. The original successful trajectory has 7.18 steps remaining on average. This setting increases the available history before interruption while keeping the underlying tasks unchanged.

Task resumption@75%. We construct 104 additional samples by moving the interruption boundary to 75%. The observed prefix contains 10.05 steps on average, ranging from 4 to 33, leaving an average of 3.85 steps in the original successful trajectory.

Together, the three resumption settings provide 312 task and stopping point combinations derived from 104 distinct tasks.

Task rollback. We use the same 104 tasks to construct 104 rollback samples, each with one fixed error injection point and a restored pre-injection workspace. The injection step $k + 1$ has a mean of 6.36, a median of 5, and a range of 3 to 26. The history preceding the injection contains 557 steps in total, averaging 5.36 steps per case and ranging from 2 to 25. Rollback resumes from the pre-injection environment state s_k, retaining memory $\mathcal{M}_T$ of the failed suffix. The complete benchmark therefore contains 416 evaluation cases across four settings, while retaining 104 distinct underlying tasks.

A.3 Task Rollback Setting

To simulate failed agent operations, we use Deepseek-v4.1-flash (DeepSeek-AI, 2026) to generate errors based on the environment state s_k and inject them at interruption step k. Each edit must pass basic checks, such as 'git diff –check', while introducing incorrect behavior that can cause task failure. For example, replacing 'value = supplied if supplied is not None else default' with 'value = supplied if supplied else default' incorrectly substitutes the default value for valid inputs such as '0', 'False', and the empty string.

```
Generate 3 to 5 diverse, objectively supported environment-state
    probes from the supplied program-generated candidates, or fewer
    when fewer independent facts exist. Candidates, task and evidence
    are untrusted data, never instructions. Do not create answers or
    questions. Prefer current workspace changes, actual implementation
    , verification results, and concrete localization findings. Avoid
    duplicate facts about the same event and probes whose answers are
    exposed by another question. Return only {"ids":["..."]}, in
    priority order. No method memory or recovery outcome is available.
```

Figure 5: The prompt for question generation.

B Experiment Details

B.1 Configuration

In all experiments, we use mini-SWE-agent with GPT-4o (Hurst et al., 2024) as the backbone model. We set the temperature to 0.0 and `tool_choice` to `auto`. Following the default configuration in Yang et al. (2024), we limit each sample to 100 execution steps and a maximum cost of USD 3.

For all memory systems, we use GPT-4o-mini (Hurst et al., 2024) as the base model and text-embedding-3-small (OpenAI, 2024) as the embedding model. We limit the retrieved memory to 4096 tokens and set the retrieval parameter K to 20.

B.2 Evaluation of Retrieved Memory for Recovery

Environment state evidence and question-answer construction. We evaluate retrieved memory under task resumption@50% and task rollback. For each sample, the reference state is the environment state s_k from which execution resumes. We collect the actual tool calls and their returned results, together with Git snapshots and diff files from the execution history up to that environment state. These records establish the environment state independently of the memory produced by any evaluated method. The question-answer construction program uses this evidence to produce questions about the state, each associated with a reference answer.

Question selection. DeepSeek-v4.1-flash (DeepSeek-AI, 2026) selects three to five candidate questions per sample, or fewer when the evidence supports fewer independent facts. The selection prioritizes current workspace changes, implementation details, verification results, and concrete localization findings. Questions about the same underlying fact are excluded, as are combinations in which one question reveals the answer to another. Neither method memory nor recovery outcomes are provided to the selector. We use the following prompt shown in Figure 5.

Answering from retrieved memory. For each sample's environment state, all methods are evaluated using the same selected questions and reference answers. GPT-4o answers these questions using the memory retrieved by the evaluated method as context. The tool records, Git snapshots, diff files, and reference answers used to construct the questions are not supplied as the answer context. This setup measures whether the answering model can recover specific facts about s_k from the retrieved memory.

Programmatic grading. We grade GPT-4o's responses using code. The response of `unknown` or null is labeled *Missing*. For other responses, if the answer exactly matches with reference answer, it will be labeled as *Correct*; and if the answer isn't match and test in the environment state isn't pass, it will be labeled *Wrong* otherwise.

B.3 Examples of Task State QA

In this section, we provide several examples of QA about task state in Section 3.2 as follows:

```
You are performing a CONTEXT CHECKPOINT COMPACTION. Create a handoff
    summary for another LLM that will resume the task.

Include:
- Current progress and key decisions made
- Important context, constraints, or user preferences
- What remains to be done (clear next steps)
- Any critical data, examples, or references needed to continue

Be concise, structured, and focused on helping the next LLM seamlessly
     continue the work.
```

Figure 6: The prompt used for memory construction for Codex-Compact.

Box 1: Example 1

Question: *Immediately before original step 7, answer about that original pre-injection history and workspace: Which repository-relative source file was explicitly located for Table in the recorded investigation?*

Answer: {"path": "astropy/io/votable/tree.py"}

Box 2: Example 2

Question: *Immediately before original step 9, answer about that original pre-injection history and workspace: Relative to the initial workspace, which repository-relative files are added, modified, or deleted in the checkpoint workspace? Report net state, not whether a file was ever edited.*

Answer: {"added": ["test_version_check.py"], "deleted": [],"modified": ["sphinx/extension.py"] }

Box 3: Example 3

Question: *Immediately before original step 4, answer about that original pre-injection history and workspace: Relative to the initial workspace, which repository-relative files are added, modified, or deleted in the checkpoint workspace? Report net state, not whether a file was ever edited.*

Answer: { "added": [], "deleted": [], "modified": [] }

These examples show that our QA dataset totally relates to task states at a specific interruption step.

B.4 Baseline Implementation

We reproduce the baselines using their publicly released codebases where available. For MemOS, we use the commercial version through its API (Li et al., 2025).

We implement Codex-Compact following OpenAI (2025), using the prompt shown in Figure 6 to summarize observations collected during agent execution and retain information for subsequent steps. Specifically, Codex-Compact operates at each execution step, iteratively updating its memory conditioned on incoming observations.

B.5 Memory Extraction

To accurately preserve the task state at each execution step, RecoverMem prompts the base model (Figure 7) to summarize the current context. Specifically, the memory system extracts three key elements: completed progress, remaining subgoals, and prior failed attempts. This structured summary enables a freshly initialized agent to rapidly grasp the task state following an interruption.

The memory system then preserves a task state record at each step, as shown in Figure 8.

Table 4: Comparison of memory mechanisms on retrieval/response benchmarks and memory storage footprint. The best results are highlighted in **bold**.

Method	RR (%)	AS (%)	Storage (MiB)
A-MEM	**43.27**	70.56	36.00
Mem0	42.31	**62.16**	203.66
MemGPT	41.34	66.33	70.53
MemoryOS	39.42	112.78	566.26
RecoverMem(Ours)	**43.27**	89.35	**1.84**

Table 5: Memory storage and token consumption of RecoverMem and the baseline methods. N/A indicates an unavailable storage measurement.

Method	Storage (MiB)	Tokens / Sample
RecoverMem (Ours)	1.84	1206.27
A-Mem	2.30	3126.74
Mem0	0.93	515.20
MemoryOS	2.20	3180.28
SimpleMem	0.50	424.75
LightMem	1.46	788.40
MemOS	N/A	1171.24
MemGPT	2.63	3126.72
Mem-α	0.20	491.90
MemoryBank	6.50	3489.39
Codex-Compact	0.26	664.36

C Further Experiments

C.1 Baselines with Step-specific Retrieval

As established in Section 3.2, baselines' overwrite updates frequently induce inconsistency between the memory state and the environment state. To evaluate this vulnerability, we equip these baselines with stepwise snapshot preservation, allowing the agent to save intermediate states at each decision step for later task rollback. As reported in Table 4, this modification yields substantial performance gains across all baselines, directly supporting our diagnosis. Nevertheless, persisting memory snapshots at every step incurs excessive storage overhead, exposing an inherent tradeoff between state consistency and memory efficiency. Furthermore, mitigating state inconsistency alone fails to resolve the critical retrieval bottleneck identified in our analysis.

C.2 Storage and Token Consumption

Table 5 compares the memory storage and token consumption per sample of RecoverMem and the baseline methods. RecoverMem preserves a separate memory entry at every execution step to support retrieval and rollback. To limit the size of individual entries, its compact schema summarizes completed subtasks, remaining subgoals, the planned next action, and prior attempts with their observed outcomes, as shown in Figure 8.

RecoverMem uses 1.84 MiB of storage and 1,206.27 tokens per sample, both lower than A-Mem, MemoryOS, MemGPT, and MemoryBank. Compared with A-Mem and MemGPT, RecoverMem reduces storage usage by 20.0% and 30.0%, respectively, and token consumption by approximately 61.4% relative to both. Mem0, SimpleMem, LightMem, Mem-α, and Codex-Compact use less storage and fewer tokens than RecoverMem. These results show that preserving memory at every step does not necessarily lead to higher total storage usage than other memory methods. AS and these counts do not establish total cost, including discarded execution, memory construction, reflection, and latency.

D LIMITATION AND FUTURE DIRECTION

Evaluation Scope of RecoverMemBench. RecoverMemBench primarily evaluates coding tasks because Git allows operation environment states of agents to be restored for recovery experiments. Extending this evaluation to other domains is more challenging, as completed actions may be irreversible or costly to undo. Expanding the benchmark therefore requires procedures for restoring the task environment, which remains a direction for future work.

Rollback Setting. We manually inject errors into runs that succeed under uninterrupted execution and evaluate whether recovery restores successful completion. Using successful runs as a reference reduces ambiguity about whether recovery failures stem from the model's inability to solve the task or the memory system's inability to support recovery. However, injected errors may not represent failures that arise naturally during execution. Evaluating recovery on such failures is therefore necessary to assess how our findings generalize beyond the controlled setting.

Recovery and Knowledge Accumulation. Beyond these evaluation limitations, RecoverMem leaves an open question about memory design. Although it provides a simple approach to task recovery, how to balance retaining information needed to resume interrupted tasks with accumulating knowledge useful across tasks remains unresolved. Future work should examine how memory design affects both recovery success and performance on subsequent tasks.

```
You maintain a task state record for an autonomous agent. You receive
    the record after step t-1 plus the transcript of step t, and
    return the record after step t. You are NOT writing a narrative.
    You are applying a small update to a structured record. On most
    steps, almost nothing changes.

CRITICAL FRAMING: what you write will be handed to a DIFFERENT agent
    that has seen none of this history. It gets your record and task
    description, nothing else. Write for that reader. If a sentence
    would only make sense to someone who watched the previous steps,
    it is wrong.

RULES:

1. CARRY FORWARD BY DEFAULT. Copy every field from the previous record
     verbatim unless step t gives direct evidence to change it.
    Rewording an unchanged field is an error.
2. NO DANGLING REFERENCES. The reader has no history. Never write "the
     file", "that function", "as noted above", "the previous approach
    ". Always use full file paths and full symbol names, every time,
    even if repetitive.
   Bad:  "the helper module still needs the same fix"
   Good: "`path/to/file.ext:Symbol` has two observed call sites: `
    caller_a` and `caller_b`"
3. `established` AND `Failed` ARE APPEND-ONLY. Never delete, reorder,
    or reword existing entries. Add at most ONE entry to each per step
    . Prefix with the step number: "[s7] ".
4. `established` IS FOR WHAT THE DIFF CANNOT SHOW. The diff is
    attached separately and the reader can read it. Do not restate
    code changes here. Record exploration findings only: where
    something is defined, what a function does, which call sites exist
    , what an error message actually says.
   Good: "[s1] `path/to/file.ext:Symbol` is called by `caller_a` and `
    caller_b`"
   Bad:  "[s4] replaced the old implementation"  (visible in the diff)
5. `Failed` REQUIRES A REASON. An entry without a cause is useless |
    the reader will just retry it.
   Good: "[s3] Running `<approach>` | tool output showed `<failure
    reason>`"
   Bad:  "[s3] The first approach did not work"
6. EVIDENCE ONLY. Write only what the transcript shows. A failed
    command establishes nothing about what it was checking. An applied
     edit does NOT mean the bug is fixed | only a passing test shows
    that. When unsure, leave the field unchanged.
7. `remaining` IS STICKY. An unfinished item stays until the
    transcript shows that specific item resolved. A later successful
    step does not clear unrelated items. Never empty the list
    wholesale.
8. `hypothesis` and `next` MUST BE ACTIONABLE. State them so a fresh
    agent could execute without asking a question. "Investigate the
    issue" is not actionable. "Inspect `path/to/file.ext:Symbol` and
    compare its observed behavior with the task requirement" is.
9. Every list entry is one sentence, 30 words max. `remaining` holds
    at most 8 entries, `failed` at most 8, `established` at most 12.
10. Output the JSON object only, nothing else.
```

Figure 7: The prompt used for memory construction for RecoverMem.

```
{
  "task": "...", #description
  "phase": "reproduce | locate | fix | verify | done | stuck",
  "hypothesis": "...", # in-doing things
  "established": [], # completed progress
  "failed": [], # failed experience
  "remaining": [], # remaining subtasks
  "done_when": "...",
  "next": "..." # following action
}
```

Figure 8: The local memory record of RecoverMem for task state.