EVORELAY: ASYNCHRONOUS ONLINE EXPERIENCE LEARNING FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents increasingly solve long-horizon tasks in interactive environments, where past interactions provide reusable experience for future decisions. Existing memory-based self-evolution methods commonly organize experience at the task level, retrieving historical experience by global task similarity and making newly acquired experience available only after task completion. However, long-horizon tasks typically unfold through functional stages with distinct objectives. This task-level design creates both a granularity mismatch and an efficiency bottleneck: global similarity may fail to identify experience relevant to the current stage, while newly acquired experience cannot benefit subsequent tasks until the source task completes, limiting the efficiency of online experience learning. To address these limitations, we propose Evorelay, an asynchronous online experience learning framework that unifies experience retrieval, extraction, and cross-task propagation around functional stages. Within this unified organization, historical experience is retrieved according to the current stage objective, and experience learned from completed stages becomes available to subsequent tasks while the source task continues. This asynchronous relay reduces cross-task experience propagation latency and substantially improves the efficiency of online experience learning. Across four backbone models, Evorelay improves task success over Vanilla Agent by an average of **6.1 percentage points** on SWE-bench Verified and **3.8 percentage points** on AppWorld. It achieves the highest average task success across the evaluated settings, with **2.6–3.4**$\times$ mean end-to-end speedups over two representative memory-based self-evolution baselines.

1 INTRODUCTION

Large language model (LLM) agents are increasingly applied to complex, long-horizon tasks such as software engineering and multi-application digital workflows (Jimenez et al., 2023; Trivedi et al., 2024). During long-term deployment, these agents naturally operate over a stream of new tasks (Ouyang et al., 2026; Yu et al., 2026). Although task objectives differ, past interactions may contain reusable strategies and failure lessons. Memory-based self-evolution therefore aims to accumulate experience from past interactions and improve subsequent decisions without updating the underlying model parameters (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025).

Existing methods commonly extract experience only after task completion and retrieve it based on global task similarity (Zhao et al., 2024; Zheng et al., 2024; Ouyang et al., 2026). Long-horizon tasks, however, often proceed through multiple functional stages with distinct intermediate objectives; a software engineering agent, for example, may proceed through Analyze, Reproduce, Edit, and Verify. This task-level design creates a granularity mismatch: global similarity may fail to identify experience relevant to the current stage, since globally similar tasks can require different stage-specific strategies while globally dissimilar tasks may share the same local strategy. Figure 1(a) illustrates this mismatch: globally similar export tasks need different fixes for text encoding and reading all pages, whereas globally dissimilar export and contact-counting tasks both need to read all pages. This motivates retrieval by the current stage objective rather than global similarity.

The task-level design also limits the efficiency of online experience learning: experience acquired in an early stage cannot become available to subsequent tasks until the source task completes (Ouyang et al., 2026; Cao et al., 2026). As a result, useful experience may remain unavailable even after it has

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.

been generated, delaying its use by subsequent tasks and producing the cross-task delay illustrated in Figure 1(b).

To address these limitations, we propose Evorelay, an asynchronous online experience learning framework. Evorelay organizes experience retrieval, experience extraction, and cross-task propagation within a unified online learning process, where functional stages serve as the shared organizing structure across these processes. Within this unified organization, historical experience is retrieved according to the current stage objective rather than global similarity when the agent enters a new stage. Upon stage completion, the stage trajectory is distilled into multiple atomic experiences capturing transferable strategies. Once the corresponding update completes, the resulting record becomes available to subsequent tasks while the source task continues execution.

Our main contributions are:

- **Unified online experience learning.** Evorelay unifies experience retrieval, extraction, and cross-task propagation using functional stages as a shared organizing structure. It retrieves historical experience according to the current stage objective rather than global task similarity, distills completed stages into reusable atomic experiences, and makes newly acquired experience available to subsequent tasks.
- **Asynchronous cross-task experience propagation.** Evorelay makes experience learned from a completed stage available to subsequent tasks while the source task continues to later stages. By decoupling experience propagation from source-task completion, it reduces the delay between experience acquisition and cross-task reuse, enabling continuous experience propagation across the task stream.
- **Empirical evaluation.** Across four backbone models on SWE-bench Verified and AppWorld, Evorelay improves average task success over Vanilla Agent by 6.1 and 3.8 percentage points, respectively. It achieves the highest average task success across all evaluated settings, with 2.6–3.4× mean end-to-end speedups over the representative memory baselines. Further analysis shows that stage-aligned experience improves stage-specific execution behavior and yields larger gains on tasks with higher interaction complexity.

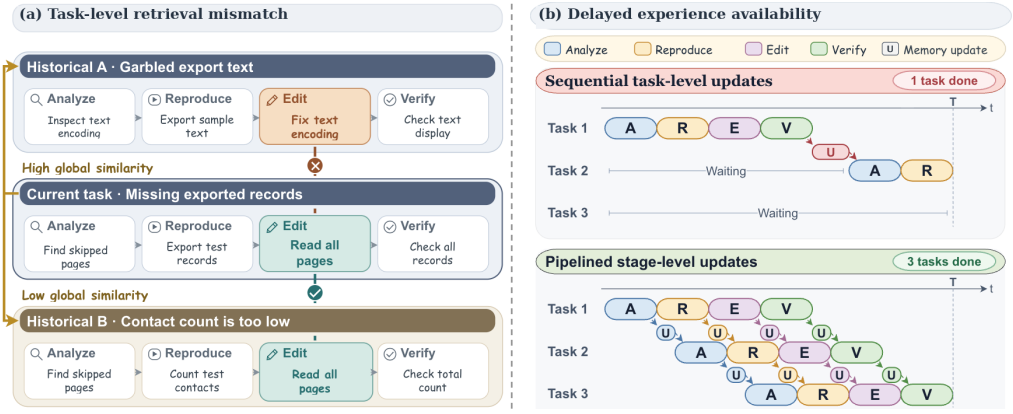


Figure 1: **Task-level retrieval and update bottlenecks.** (a) Global task similarity may miss stage-relevant strategies. (b) Task-level updates defer reuse until source-task completion, while Evorelay makes completed-stage experience available earlier.

2 Related Work

Long-horizon agents. Long-horizon tasks require agents to coordinate reasoning, action, state tracking, and recovery over extended interactions, where intermediate decisions may accumulate and affect later execution (Dong et al., 2026; Wu et al., 2026). Prior work improves long-horizon agents through planning, memory management, tool use, verification, and reusable skills (Yao et al., 2022; Yang et al., 2024; Packer et al., 2023; Zhou et al., 2025; Wang et al., 2023; Yang et al., 2025; Li et al., 2026). As agents operate over task streams, past interactions become a source of reusable experience for subsequent tasks. Evorelay studies how this experience is organized and made available for reuse.

Self-evolving memory for LLM agents. Memory-based self-evolution enables agents to accumulate experience from past interactions and improve future decisions without updating the underlying model parameters (Zhang et al., 2026). Existing methods represent such experience as verbal reflections, general lessons, reusable workflows, or distilled reasoning strategies (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025; Ouyang et al., 2026; Wang et al., 2026). These approaches typically organize experience around complete task trajectories, using finished tasks as the basic unit for experience extraction and future decision making. However, long-horizon tasks often consist of multiple functional stages with distinct intermediate objectives, making complete-task organization less suitable for capturing stage-specific experience needs. Recent work has refined experience learning through context-aware retrieval, structured memory organization, and finer-grained experience extraction (Fu et al., 2024; Yu et al., 2026; Shen et al., 2026; Li et al., 2026; Nottingham et al., 2024). These advances improve the alignment between stored experience and local execution needs. However, improving experience relevance does not by itself determine when newly acquired experience becomes available across tasks. Under task-completion-based updates, experience generated during intermediate execution may remain unavailable to subsequent tasks until the source task finishes.

Evorelay addresses these limitations by organizing online experience learning around functional stages as a shared organizing structure. This unified organization jointly addresses two challenges in long-horizon experience learning: aligning historical experience with the objective of the current stage and enabling newly acquired experience to become available across tasks without waiting for source-task completion. By coordinating these processes through functional stages, Evorelay enables asynchronous cross-task experience learning over long-horizon task streams.

3 METHOD

3.1 PROBLEM FORMULATION AND OVERVIEW

We study online experience learning over a stream of long-horizon tasks $\mathcal{S} = (\tau_1, \ldots, \tau_N)$ while keeping the backbone LLM parameters θ fixed. The trajectory of task τ_i is partitioned into functional stages, $\xi_i = \xi_{i,1} \oplus \cdots \oplus \xi_{i,J_i}$, where $\oplus$ denotes temporal concatenation and $\xi_{i,j}$ is the trajectory of stage j.

As illustrated in Figure 2, Evorelay organizes experience retrieval, extraction, and cross-task propagation around functional stages. Upon entering a stage, the agent retrieves a stage record aligned with the current functional category and objective to guide execution. After stage completion, its trajectory is distilled into transferable atomic experiences and stored as a new stage record. Once the update finishes, the record becomes available to subsequent tasks while the source task may continue.

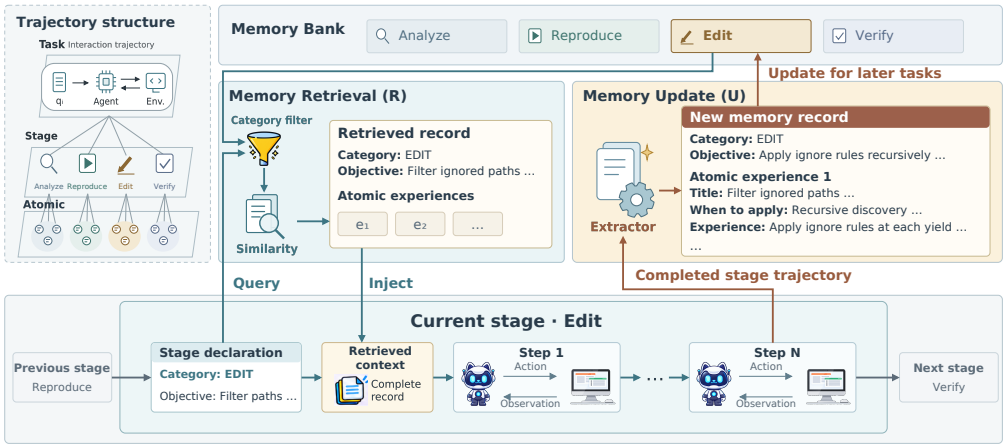


Figure 2: **Evorelay's stage-aligned experience workflow.** When entering a new stage, Evorelay retrieves a complete record matching the category and objective to guide execution. After the stage update, the new record becomes available to subsequent tasks while the source task continues.

3.2 Stage Segmentation and Stage Memory Records

Dynamic stage segmentation. Long-horizon tasks often unfold through multiple functional stages with distinct intermediate objectives. Evorelay uses these functional stages as the organizational units for online experience learning. When entering a new stage, the agent emits a structured stage declaration $\phi_{i,j} = (c_{i,j}, o_{i,j})$, where $c_{i,j}$ denotes the functional category and $o_{i,j}$ describes the local objective together with stage-relevant entities and operations. The declaration marks the beginning of a stage, and the next declaration or task termination determines its boundary, yielding the corresponding trajectory $\xi_{i,j}$. For example, a software-engineering agent may progress through functional stages such as ANALYZE, REPRODUCE, EDIT, and VERIFY. Stage boundaries emerge online according to changes in functional objectives. Appendix C.2 lists the functional-stage sets used for SWE-bench Verified and AppWorld.

Stage memory record. A completed stage may contain multiple transferable strategies with different applicability conditions. Evorelay therefore distills them into separate atomic experiences and groups them within a stage-level memory record:

$$m_{i,j} = \big(\phi_{i,j}, I_{i,j}\big), \quad I_{i,j} = \big(e_{i,j}^{(1)}, \ldots, e_{i,j}^{(L_{i,j})}\big). \tag{1}$$

The stage declaration $\phi_{i,j}$ identifies the record's functional category and structured objective. $I_{i,j}$ is a list of atomic experiences distilled from the stage trajectory, and $L_{i,j}$ denotes the number of items in that list. Each atomic experience is a compact, structured item that captures a transferable strategy, reasoning insight, or failure lesson while abstracting away instance-specific execution details.

3.3 Stage-Aligned Experience Retrieval

Evorelay first filters available records by functional category $c_{i,j}$. It then selects the record whose historical stage objective is most similar to the current objective:

$$m_{i,j}^{*} = \underset{m \in \mathcal{A}_{i,j}}{\arg\max}\, \mathrm{Cosine}(E(o_{i,j}), E(o_m))\,, \tag{2}$$

where E is the text encoder, o_m is the structured objective stored in historical record m, and $\mathcal{A}_{i,j}$ contains historical records whose updates have completed before the current stage begins and whose functional category matches $c_{i,j}$. Evorelay uses the selected record and its atomic experiences to guide execution.

3.4 Asynchronous Cross-Task Experience Propagation

Stage-triggered experience extraction. At stage completion, Evorelay uses the same backbone model as the acting agent to assess the completed stage against its declared objective and distill the trajectory into reusable experience. Successful stages yield effective strategies, while unsuccessful stages provide failure-avoidance lessons. The resulting experiences form a stage record, which becomes available to subsequent tasks once the corresponding update completes, while the source task may continue.

Experience availability and cross-task propagation. Online self-evolution relies on newly acquired experience becoming available for subsequent retrieval. Task-level updating delays this availability until source-task completion, making task completion the boundary for cross-task propagation. Evorelay shifts this boundary to stage-update completion, allowing completed-stage experience to guide subsequent tasks while the source task continues.

To formalize this asynchronous execution, let $s_{i,j}$ denote the start time of stage (i, j), and $f_{i,j-1}$ denote the finish time of the preceding stage within the same task. Let $a_{i,j}$ denote the completion time of the corresponding functional-stage experience update from the immediately preceding task. The start of each stage satisfies

$$s_{i,j} \geq \max\left(f_{i,j-1}, a_{i,j}\right). \tag{3}$$

This constraint captures both within-task execution dependencies and cross-task experience availability. Evorelay makes each completed-stage record available once its update finishes, without waiting for source-task completion. This allows experience extraction, continued source-task execution, and downstream experience reuse to overlap across tasks.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks and metrics. We evaluate Evorelay on two long-horizon agent benchmarks, SWE-bench Verified (Jimenez et al., 2023) and AppWorld (Trivedi et al., 2024). SWE-bench Verified contains 500 real-world repository-level issue-resolution tasks and uses the official patch evaluator to compute Pass@1. AppWorld evaluates an agent's ability to complete user goals through cross-application interaction and uses the official task goal completion (TGC) metric. We use standard mini-SWE-agent on SWE-bench Verified and the official ReAct agent on AppWorld.

Backbones, baselines, and task-stream protocol. We evaluate Gemini 2.5 Pro, Gemini 2.5 Flash, GPT-5.2, and DeepSeek V4-Pro. We compare against a Vanilla Agent and two representative memory-based self-evolution baselines: ReasoningBank (Ouyang et al., 2026), a strong task-level memory baseline that distills completed trajectories into instance-level reasoning experience, and SASM (Shen et al., 2026), a structured-memory method for software-engineering agents.

For each backbone–benchmark setting, all methods process tasks in the same order with the same tools, execution and worker budgets, concurrency limit, and environment configuration. In a task stream, memory-based self-evolution methods improve later decisions by leveraging experience accumulated from earlier interactions. For baseline implementations, newly acquired experience becomes available to subsequent tasks only after the corresponding source task completes. Evorelay instead makes experience from a completed stage available once its update finishes, allowing subsequent tasks to benefit from newly acquired experience while the source task continues execution.

To account for task-order variation, we construct three random task orders, and all methods use the same three orders. Every run begins with empty memory, and memory-based methods continuously accumulate and reuse experience while processing the task stream. We set the model temperature to 0 and keep all other non-memory configurations identical. We report the mean task success rate and sample standard deviation over the three runs, Best@3 (the best result among the three runs), and the mean end-to-end task-stream time, including task execution, experience retrieval and extraction, memory updates, and associated waiting time.

4.2 OVERALL PERFORMANCE AND EFFICIENCY

Overall task performance. As shown in Table 1, Evorelay achieves consistent gains across backbone models and task domains. Compared with Vanilla Agent, Evorelay improves average task success by **6.1 percentage points** on SWE-bench Verified (Figure 3) and **3.8 percentage points** on AppWorld. Evorelay outperforms both ReasoningBank and SASM across all eight backbone–benchmark settings.

End-to-end efficiency. Evorelay is consistently the fastest memory-based method. Across settings, mean end-to-end task-stream time reductions are 60.4% relative to ReasoningBank and 69.9% relative to SASM. Mean per-setting speedups are approximately **2.6× and 3.4×**, respectively. End-to-end wall-clock time includes task execution, experience retrieval and extraction, memory updates, and waiting time (Figure 9).

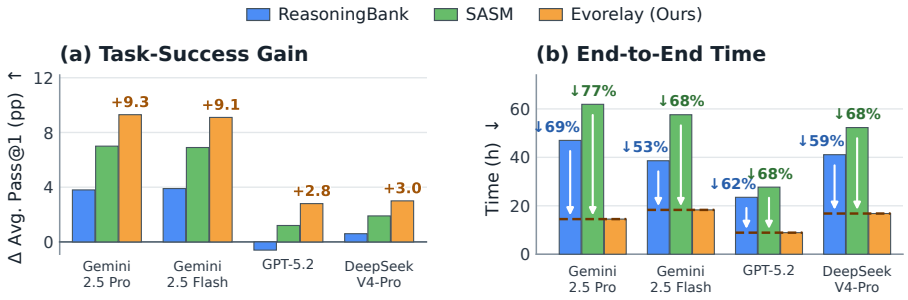


Figure 3: **Results on SWE-bench Verified.** (a) Average Pass@1 gain (pp) over the Vanilla Agent; Evorelay has the largest gain on all four backbones. (b) End-to-end time (h); percentages report Evorelay's reductions relative to each memory baseline.

Table 1: **Task success and runtime on SWE-bench Verified and AppWorld.** Task-success scores are percentages. Best@3 is the highest score across three runs; Avg. reports the mean and sample standard deviation (Pass@1 for SWE-bench Verified, TGC for AppWorld). Bold marks the highest Best@3 and mean, or the shortest runtime, within each backbone and benchmark; Vanilla Agent runtimes are omitted.

	SWE-bench Verified			AppWorld		
Method	Best@3↑	Avg. Pass@1↑	Time (h)↓	Best@3↑	Avg. TGC↑	Time (h)↓
Gemini 2.5 Pro						
Vanilla Agent (with no memory)	53.8	52.7 ± 0.95	–	70.2	69.2 ± 1.24	–
ReasoningBank	57.6	56.5 ± 1.03	47.0	72.6	70.6 ± 2.48	6.7
SASM	61.2	59.7 ± 1.40	61.9	72.6	71.0 ± 1.82	8.2
Evorelay (Ours)	**62.8**	**62.0** ± 1.22	**14.5**	**75.0**	**73.2** ± 1.57	**3.0**
Gemini 2.5 Flash						
Vanilla Agent (with no memory)	34.8	34.1 ± 0.61	–	62.5	60.9 ± 1.50	–
ReasoningBank	39.2	38.0 ± 1.44	38.6	59.5	58.1 ± 1.91	4.7
SASM	41.8	41.0 ± 0.72	57.6	64.9	63.7 ± 1.19	5.9
Evorelay (Ours)	**44.2**	**43.2** ± 0.92	**18.3**	**67.9**	**66.9** ± 1.24	**1.4**
GPT-5.2						
Vanilla Agent (with no memory)	73.4	72.1 ± 1.14	–	91.1	90.3 ± 0.91	–
ReasoningBank	72.6	71.5 ± 1.01	23.5	92.9	91.7 ± 1.03	4.5
SASM	73.8	73.3 ± 0.76	27.7	92.3	91.5 ± 1.37	6.4
Evorelay (Ours)	**75.6**	**74.9** ± 0.83	**8.9**	**93.5**	**92.1** ± 1.24	**1.8**
DeepSeek V4-Pro						
Vanilla Agent (with no memory)	82.2	81.2 ± 0.87	–	93.5	92.5 ± 0.91	–
ReasoningBank	82.6	81.8 ± 1.06	41.1	94.6	93.3 ± 1.50	7.7
SASM	83.8	83.1 ± 0.99	52.3	95.2	94.4 ± 0.69	10.6
Evorelay (Ours)	**84.8**	**84.2** ± 0.72	**16.8**	**96.4**	**95.8** ± 0.60	**3.5**

4.3 Ablation Studies

We conduct ablations on the full SWE-bench Verified task stream with DeepSeek V4-Pro to examine functional stage decomposition, experience representation, retrieval policy, and cross-task experience availability. Within each comparison, we vary the factor under study while keeping the remaining configuration fixed, including task order, tools, execution budgets, and the shared four-worker budget.

Effect of stage decomposition. We first examine whether introducing functional stages alone improves task success. To this end, we retain Evorelay's stage declarations and transitions but disable cross-task experience retrieval and memory updates. As shown in Table 2(a), this variant achieves 82.0% Pass@1, 0.8 percentage points above the Vanilla Agent (81.2%). Full Evorelay reaches 84.2%, a further 2.2-percentage-point gain over the stage-only variant. These results show that functional stages alone provide only a modest benefit, while the larger improvement comes from stage-level experience learning and reuse.

Effect of experience representation. We compare three representations of completed stage trajectories: the raw trajectory, a single stage summary, and a stage record containing multiple atomic experiences. As shown in Table 2(b), abstracting the raw trajectory into a stage summary raises Pass@1 from 81.4% to 83.2%, and organizing the record as multiple atomic experiences further raises it to 84.2%. The two gains show that both experience abstraction and fine-grained organization contribute to transfer.

Effect of stage-aligned retrieval. Global-task retrieval first selects the most similar historical task using the full problem description and then uses its record for the current stage. Evorelay instead selects among same-category stage records according to the current stage objective. As shown in Table 2(c), stage-objective retrieval improves Pass@1 from 82.8% to 84.2%, a gain of 1.4 percentage

Table 2: **Accuracy ablations on SWE-bench Verified.** DeepSeek V4-Pro Pass@1; Δ in (a) is relative to Vanilla Agent.

(a) Functional stage decomposition

Configuration	Pass@1 (%)↑	Δ (pp)
Vanilla Agent	81.2	–
Stage decomposition only	82.0	+0.8
Evorelay (full)	**84.2**	**+3.0**

(b) Experience representation

Representation	Pass@1 (%)↑
Raw stage trajectory	81.4
Stage summary	83.2
Multiple atomic experiences	**84.2**

(c) Retrieval policy

Retrieval policy	Pass@1 (%)↑
Global-task retrieval	82.8
Stage-objective retrieval	**84.2**

Table 3: **Cross-task availability ablation on SWE-bench Verified.**

Cross-task availability	Pass@1 (%)↑	Time (h)↓	Speedup↑
After task completion	84.4	48.7	1.00×
After completed-stage update	84.2	**16.8**	**2.90**×

points. This controlled comparison complements the ranking analysis in Figure 4, showing that stage-aligned retrieval not only prioritizes different historical sources but also improves task success.

Effect of cross-task experience availability. We compare two Evorelay variants that differ in when completed stage records become available to subsequent tasks. In the task-completion variant, records become available only after the source task finishes; in the stage-update variant, each record becomes available once the corresponding stage update completes. This earlier availability enables greater overlap across tasks, reducing end-to-end runtime from 48.7 to 16.8 hours (2.90×), while Pass@1 remains essentially unchanged (84.4% vs. 84.2%).

4.4 Analysis

We further analyze Evorelay on SWE-bench Verified, examining observed stage-progression patterns, differences between global-task and stage-objective rankings, stage-specific behavioral changes, long-range experience reuse, and performance across interaction complexity. Unless otherwise specified, the analyses use DeepSeek V4-Pro.

Table 4: **Forward stage-sequence patterns on SWE-bench Verified (%).** Complete denotes $A \rightarrow R \rightarrow E \rightarrow V$; Other forward denotes order-preserving shorter paths.

Backbone	Complete	Other forward	Forward total
Gemini 2.5 Pro	88.8	6.8	95.6
Gemini 2.5 Flash	60.8	37.2	98.0
GPT-5.2	98.2	0.6	98.8
DeepSeek V4-Pro	99.8	0.2	100.0

Stage progression patterns. We analyze whether stage transitions exhibit consistent structure across task executions. As shown in Table 4, 60.8–99.8% of trajectories follow the complete Analyze→Reproduce→Edit→Verify sequence. When shorter order-preserving paths are included, 95.6–100.0% of trajectories exhibit forward progression across backbone models. These results characterize stage transitions as highly structured and predominantly forward-moving across backbone models, while still allowing stages to be skipped. This predominantly forward stage structure provides a favorable basis for overlapping stage execution, experience updates, and cross-task experience reuse.

Effectiveness of stage-aligned experience reuse. We first compare stage-aligned retrieval with a ranking of historical tasks by global task similarity. Only 21.1% of Evorelay's selected records come from the top-ranked historical task, with a median task-level rank of 6 (Appendix D.4, Table 10). Thus, the experience selected for the current stage is often not prioritized by global similarity.

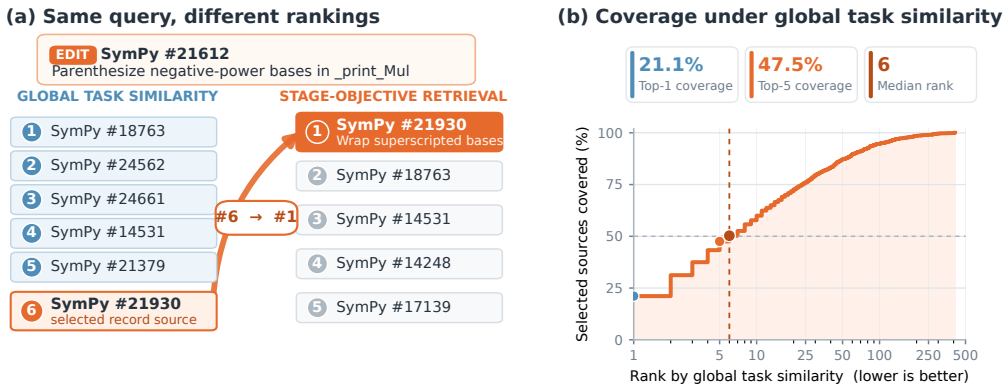


Figure 4: **Global-task and stage-objective rankings prioritize different sources.** (a) Source-task rank 6 versus Edit-record rank 1. (b) Global-task ranks of stage-selected sources.

Figure 4 illustrates this difference. For an Edit-stage query, `sympy__sympy-21930` ranks sixth under global similarity but rises to first under the current Edit objective, providing guidance on parenthesizing superscripted power bases. The aggregate distribution in Figure 4(b) shows that such rank shifts occur systematically across retrieval events.

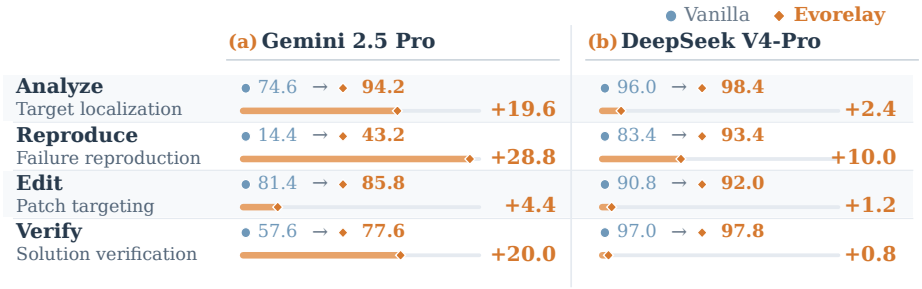


Figure 5: **Stage-wise behavioral indicators: Vanilla Agent versus Evorelay (%; gains in pp).**

We further examine whether stage-aligned reuse is accompanied by improvements in stage-specific behavior. Following Kimi-Dev (Yang et al., 2025), we measure target localization, failure reproduction, patch targeting, and solution verification for ANALYZE, REPRODUCE, EDIT, and VERIFY, respectively. Across Gemini 2.5 Pro and DeepSeek V4-Pro, Evorelay improves all four behavioral indicators.

Together, these results show that stage-aligned retrieval not only surfaces historical experience beyond what global similarity prioritizes, but is also accompanied by consistent improvements in stage-specific execution behavior. Section 4.5 further illustrates the downstream effect in a concrete task.

Long-range experience reuse. We examine whether stage-level experience reuse is concentrated among nearby tasks, defining source–target distance as $\delta = i - r$, where i and r are target and source task positions. The median distance is 90 tasks (Figure 6), with 46.6% of retrievals spanning at least 100 tasks, showing that experience persists over long task-stream distances.

Performance across interaction complexity. Using Vanilla Agent environment-interaction steps as a fixed complexity proxy, we group the 500 tasks into Easy ($\leq$ 27 steps, $n = 177$), Medium (28–47, $n = 156$), and Hard ($\geq$ 48, $n = 167$). We use the same partition for all methods so that Evorelay's own execution behavior does not affect task grouping.

As shown in Figure 7, Evorelay improves Pass@1 by 1.1, 0.6, and 7.2 percentage points on Easy, Medium, and Hard tasks, respectively, with the largest gain on Hard tasks. This result suggests that stage-level historical experience plays a larger role as task interactions become longer and require more intermediate decisions and exploration.

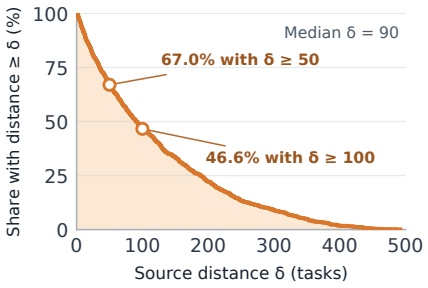


Figure 6: **Long-range experience reuse.** Fraction of retrievals spanning at least δ tasks; the median source–target distance is 90 tasks.

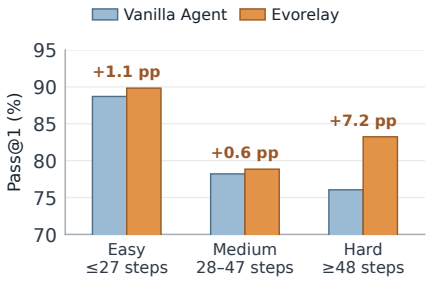


Figure 7: **Pass@1 by interaction complexity.** Comparison of Vanilla Agent and Evorelay; labels show Evorelay's gains in percentage points.

4.5 Case Study

Figure 8 examines `sympy__sympy-23262`. For a one-element tuple, `lambdify` generates `return (1)` instead of `return (1,)`, changing the result from a tuple to an integer. Retrieval based on global similarity selects memories about reflected operators and operator priority. The agent investigates printer dispatch and patches the wrong path, leaving the issue unresolved.

Evorelay's Analyze experience identifies the missing comma produced by joining the elements of a singleton tuple. Its Edit experience supplies the matching repair rule: render one-element tuples as `(x,)`. The agent applies this special case in `_recursive_to_string` while preserving list behavior, and the resulting patch passes verification. The case shows how stage-aligned memories can support both diagnosis and a targeted edit when global similarity prioritizes guidance for a different mechanism.

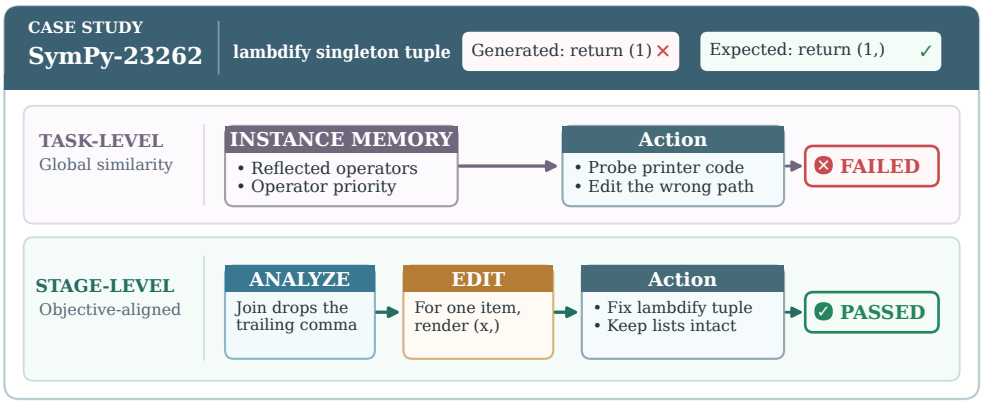


Figure 8: **Case study on `sympy__sympy-23262`.** Global similarity supplies operator-related memories, while the Analyze and Edit records identify the missing tuple comma and its repair.

5 Conclusion

We present Evorelay, an asynchronous online experience learning framework for long-horizon agents. Evorelay organizes experience retrieval, extraction, and cross-task availability around functional stages: the current stage objective determines which historical experience is retrieved, completed stages are distilled into reusable atomic experiences, and newly learned records become available to subsequent tasks once their updates finish rather than waiting for source-task completion. This shared stage structure enables experience learning, source-task execution, and downstream experience propagation to overlap. Across four backbone models, Evorelay improves average task success over Vanilla Agent by 6.1 percentage points on SWE-bench Verified and 3.8 percentage points on AppWorld, while achieving 2.6–3.4× mean end-to-end speedups over the memory baselines. These results demonstrate the potential of asynchronous online experience learning to improve both the effectiveness and efficiency of long-horizon task streams.

AI USE STATEMENT

The large language model (LLM) was utilized as a writing assistant during the preparation of this manuscript. Its application included improving the clarity and grammatical accuracy of the text, as well as assisting with selected aspects of data analysis. Specific uses included rephrasing sentences for better flow and translating initial concepts and drafts from Chinese to English. All core scientific contributions, including the conceptualization of our Evorelay framework, the design of the methodology and experiments, and the analysis and interpretation of the results, are solely the work of the authors. The authors take full responsibility for all claims and the final content of this paper.

ETHICS STATEMENT

The experiments use benchmark tasks from SWE-bench Verified and AppWorld. In deployments involving private code repositories or user data, stored experience may contain sensitive information and influence subsequent tool actions. Access controls should therefore be applied to memory records and agent tools.

REPRODUCIBILITY STATEMENT

We describe Evorelay's formulation and evaluation protocol in Sections 3 and 4. Appendices A, B, C, and E provide the algorithm, scheduling rules, benchmark and model settings, experience extraction and retrieval configuration, and acting and extraction prompts. We also provide systematic ablation studies and additional analyses in Section 4 and Appendix D to examine key design choices and support interpretation of the experimental results. We will open-source the code upon acceptance of the paper.

REFERENCES

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember Me, Refine Me: A Dynamic Procedural Memory Framework for Experience-Driven Agent Evolution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.829. URL https://aclanthology.org/2026.findings-acl.829/.

Guanting Dong, Xiaoshuai Song, Yuyang Hu, Jiajie Jin, Chenghao Zhang, Yifei Chen, Xiaoxi Li, Huaying Yuan, Xinyu Yang, Tongyu Wen, Jiejun Tan, Hongjin Qian, Shijue Huang, Junting Lu, Zhenyu Li, Wanjun Zhong, Yutao Zhu, Tat-Seng Chua, Zhicheng Dou, and Ji-Rong Wen. Towards Long-Horizon Agents: A Survey. *Preprints*, July 2026. doi: 10.20944/preprints202607.1328.v1. URL https://doi.org/10.20944/preprints202607.1328.v1.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated Generation and Selection of Context-Aware Guidelines for Large Language Model Agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 119919–119948. Curran Associates, Inc., 2024. doi: 10.52202/079017-3811. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/d8efbb5dd415974eb095c3f06bff1f48-Paper-Conference.pdf.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770, 2023. URL https://arxiv.org/abs/2310.06770. Preprint.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning Self-Evolving Skills for Coding Agents. arXiv:2605.25430, 2026. URL https://arxiv.org/abs/2605.25430. Preprint.

Kolby Nottingham, Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Sameer Singh, Peter Clark, and Roy Fox. Skill Set Optimization: Reinforcing Language Model Behavior via Transferable Skills. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 38409–38425. PMLR, 2024. URL https://proceedings.mlr.press/v235/nottingham24a.html.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560, 2023. URL https://arxiv.org/abs/2310.08560. Preprint.

Kangning Shen, Jingyuan Zhang, Chenxi Sun, Wencong Zeng, and Yang Yue. Structurally Aligned Subtask-Level Memory for Software Engineering Agents. arXiv:2602.21611, 2026. URL https://arxiv.org/abs/2602.21611. Preprint.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366, 2023. URL https://arxiv.org/abs/2303.11366. Preprint.

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. arXiv:2407.18901, 2024. URL https://arxiv.org/abs/2407.18901. Preprint.

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:2305.16291, 2023. URL https://arxiv.org/abs/2305.16291. Preprint.

Junjie Wang, Yiming Ren, and Haoyang Zhang. From Procedural Skills to Strategy Genes: Towards Experience-Driven Test-Time Evolution. arXiv:2604.15097, 2026. URL https://arxiv.org/abs/2604.15097v2. Technical report.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Kai Wu, Hao Lyu, Zhen Luo, Chaofan Wang, Siyu Ye, Jinghao Lin, Xiaozhong Ji, Boyuan Jiang, Yiwen Ye, Zimu Wang, Wenzhe Liu, Ruobing Wang, Kai Cai, Shengzhi Wang, and Qingwen Liu. Building Reliable Long-Horizon Agents: A Survey. Preprint, July 2026. URL https://building-reliable-long-horizon-agent.github.io/.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, Yanhao Li, Yue Liu, Zhenxing Hu, Kaitai Zhang, Shuyi Wang, Huarong Chen, Flood Sung, Yang Liu, Yang Gao, Zhilin Yang, and Tianyu Liu. Kimi-Dev: Agentless Training as Skill Prior for SWE-Agents. arXiv:2509.23045, 2025. URL https://arxiv.org/abs/2509.23045. Preprint.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629, 2022. URL https://arxiv.org/abs/2210.03629. Preprint.

Zhaochen Yu, Yingcheng Wu, Zhenfei Yin, Kaiyuan Chen, Zhe Zhao, Mengdi Wang, Shuicheng Yan, and Ling Yang. Recursive Experiential–Working Memory Evolution for Long-Horizon Agent Harnesses. arXiv:2608.24876, 2026. URL https://arxiv.org/abs/2608.24876. Preprint.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv:2510.04618, 2026. URL https://arxiv.org/abs/2510.04618.

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. doi: 10.1609/aaai.v38i17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-Exemplar Prompting with Memory for Computer Control. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Pc8AU1aF5e.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to Synergize Memory and Reasoning for Efficient Long-Horizon Agents. arXiv:2506.15841, 2025. URL https://arxiv.org/abs/2506.15841.

A Formalization of Evorelay

This appendix provides additional formal details of Evorelay's online experience learning, including dynamic functional stages, stage-level experience representation, stage-aligned retrieval, and asynchronous cross-task experience propagation.

A.1 Online Task Stream and Dynamic Functional Stages

We study online experience learning over a stream of long-horizon tasks

$$\mathcal{S} = (\tau_1, \tau_2, \ldots, \tau_N), \tag{4}$$

with fixed backbone-model parameters. Executing task τ_i produces an interaction trajectory ξ_i, which the acting agent organizes into temporally ordered functional stages:

$$\xi_i = \xi_{i,1} \oplus \xi_{i,2} \oplus \cdots \oplus \xi_{i,J_i}, \tag{5}$$

where $\oplus$ denotes temporal concatenation and J_i is determined by the actual execution path. Different tasks may therefore contain different numbers and sequences of functional stages.

Each stage is associated with a structured declaration

$$\phi_{i,j} = (c_{i,j}, o_{i,j}), \qquad o_{i,j} = (d_{i,j}, k_{i,j}), \tag{6}$$

where $c_{i,j}$ denotes the functional category and $o_{i,j}$ denotes the structured stage objective. Within $o_{i,j}$, $d_{i,j}$ describes the current technical objective and $k_{i,j}$ contains a small set of retrieval terms. The functional category restricts retrieval to experience of the same general execution type, while the stage objective captures the local need of the current stage.

The acting agent declares an initial stage at task entry and opens a new stage when its functional objective changes substantially. Stage boundaries therefore arise dynamically during execution rather than from fixed-length segmentation or post-hoc clustering.

Table 5: **Notation used in the formalization.**

Symbol	Meaning
τ_i, ξ_i	Task i and its complete interaction trajectory.
$\xi_{i,j}$	Trajectory generated during stage j of task i.
$\phi_{i,j}$	Stage declaration containing category and stage objective.
$o_{i,j}$	Structured stage objective for retrieval.
$m_{i,j}$	Stage memory record.
$I_{i,j}$	Atomic experiences extracted from stage (i, j).
$\mathcal{M}$	Stage-memory bank currently available for retrieval.

A.2 Stage Records and Atomic Experiences

A completed stage may yield several complementary transferable strategies. For each backbone setting, Evorelay uses the same backbone model as the acting agent to extract experience from the completed stage. Given the stage declaration and trajectory, the extractor produces

$$I_{i,j} = F_{\text{ext}}(\phi_{i,j}, \xi_{i,j}), \tag{7}$$

where

$$I_{i,j} = \left(e_{i,j}^{(1)}, \ldots, e_{i,j}^{(L_{i,j})}\right), \qquad 0 \le L_{i,j} \le L_{\max}. \tag{8}$$

The corresponding stage memory record is

$$m_{i,j} = (\phi_{i,j}, I_{i,j}). \tag{9}$$

The stage declaration specifies the functional category and structured objective of the historical stage, while $I_{i,j}$ contains the reusable experience distilled from its execution trajectory.

Conceptually, an atomic experience is represented as

$$e = (t, w, x), \tag{10}$$

where t is a concise title describing the transferable lesson, w specifies observable conditions under which the experience is applicable, and x provides the corresponding reusable strategy, constraint, or action guideline.

The extractor aims to preserve applicability conditions that can be recognized in future tasks and guidance that can influence subsequent decisions, while abstracting away one-off identifiers and execution-specific details. It may produce multiple items when a stage contains distinct transferable lessons, a single item when only one lesson is supported, or an empty list when no reusable experience can be extracted.

A.3 Stage-Aligned Retrieval

When entering stage (i, j), Evorelay considers records from preceding tasks whose updates have finished before stage entry. Let $\mathcal{M}_{i,j}^{\text{avail}}$ denote this available memory. The category-matched candidates are

$$\mathcal{A}_{i,j} = \left\{ m \in \mathcal{M}_{i,j}^{\text{avail}} \mid c_m = c_{i,j} \right\}, \tag{11}$$

where c_m denotes the functional category stored in record m.

Among records in the same category, Evorelay selects the record whose historical stage objective is most similar to the current objective:

$$m_{i,j}^{*} = \arg\max_{m \in \mathcal{A}_{i,j}} \text{Cosine}\left(E(o_{i,j}), E(o_m)\right), \tag{12}$$

where E is the `sentence-transformers/all-MiniLM-L6-v2` encoder used to represent stage objectives for similarity computation, and o_m is the structured objective stored in historical record m.

Category filtering restricts retrieval to functionally compatible experience, while objective similarity distinguishes different local needs within the same functional category. The selected stage record is then provided to the acting agent as guidance for the current stage.

A.4 Online Experience Learning

Evorelay decouples cross-task experience availability from source-task completion. After a functional stage ends, its declaration and trajectory are submitted to the experience extractor while the source task may continue to later stages.

Once extraction finishes, the resulting nonempty stage record is added to the stage-memory bank, completing the update and making the experience available to subsequent tasks before the source task finishes. Experience extraction, continued source-task execution, and downstream experience reuse can therefore overlap in time.

Rather than making newly acquired experience available only after source-task completion, Evorelay moves the experience-availability boundary to completed-stage updates. As soon as a stage update finishes, the resulting record can participate in retrieval for subsequent tasks while the source task continues execution.

Algorithm 1 summarizes this event-driven experience-learning process; worker dispatch and asynchronous task scheduling are detailed in Appendix B.

Algorithm 1 Stage-aligned retrieval and asynchronous experience update.

Require: Ordered task stream $\mathcal{S} = (\tau_1, \ldots, \tau_N)$, retrieval encoder E, experience extractor F_{ext}
Ensure: Task outputs and updated stage-memory bank $\mathcal{M}$

$\mathcal{M} \leftarrow \emptyset$
while an unfinished task or extraction job remains **do**
  if stage (i, j) begins with declaration $\phi_{i,j} = (c_{i,j}, o_{i,j})$ **then**
    Filter $\mathcal{M}_{i,j}^{\text{avail}}$ by category $c_{i,j}$ to form $\mathcal{A}_{i,j}$
    if $\mathcal{A}_{i,j} \neq \emptyset$ **then**
      Select $m_{i,j}^{*}$ by stage-objective similarity
    end if
    Dispatch stage execution with the selected record, if any
  end if
  if stage (i, j) completes with trajectory $\xi_{i,j}$ **then**
    Launch $I_{i,j} \leftarrow F_{\text{ext}}(\phi_{i,j}, \xi_{i,j})$ asynchronously
    Allow the source task to proceed during extraction
  end if
  if extraction for stage (i, j) finishes with $I_{i,j} \neq \emptyset$ **then**
    Add $m_{i,j} = (\phi_{i,j}, I_{i,j})$ to $\mathcal{M}$
  end if
end while
return task outputs and $\mathcal{M}$

B Pipeline Scheduling and Stage Progression

B.1 Stage-Level Pipeline Scheduling

Evorelay allows completed-stage experience to become available across tasks independently of source-task completion. After a stage ends, experience extraction can overlap with subsequent execution of the source task. Once the corresponding stage record becomes available, later tasks can retrieve and reuse the experience when entering the corresponding functional stage, without waiting for the source task to finish.

As a result, experience extraction, continued source-task execution, and cross-task experience propagation can overlap in time, forming a stage-level pipeline across tasks.

For example, after a task completes its ANALYZE stage, the corresponding Analyze experience can be extracted while the source task continues to later stages. Once the stage record becomes available, a subsequent task entering ANALYZE can retrieve and reuse that experience even if the source task is still executing EDIT or VERIFY.

Compared with task-completion-gated availability, making completed-stage records available immediately after their updates reduces the delay between experience generation and propagation to subsequent tasks.

B.2 Variable Stage Paths and Stage Progression

In practice, different tasks may exhibit different functional-stage paths, with some stages absent from particular executions. If the immediately preceding task does not contain the corresponding functional stage, the current stage proceeds without an additional cross-task availability constraint. To characterize the structure of these stage transitions, we analyze the stage sequences produced on SWE-bench Verified.

We refer to a trajectory as *forward-compatible* when its observed stages follow the typical functional progression while allowing some stages to be omitted. For example,

$$\textsc{Analyze} \rightarrow \textsc{Edit} \rightarrow \textsc{Verify}$$

and

$$\textsc{Analyze} \rightarrow \textsc{Verify}$$

are both forward-compatible paths.

We reconstruct the stage sequences of completed SWE-bench Verified tasks from the stage declarations produced during execution. Across backbone models, 60.8–99.8% of tasks follow the complete $\textsc{Analyze}\rightarrow\textsc{Reproduce}\rightarrow\textsc{Edit}\rightarrow\textsc{Verify}$ sequence. When shorter paths that preserve this forward progression are also included, 95.6–100.0% of trajectories are forward-compatible. Table 4 reports the aggregate statistics.

These results show that, despite variation in the stages encountered by individual tasks, stage transitions remain highly structured and predominantly forward-moving in practice. This regularity provides a favorable structure for overlapping stage execution, experience updates, and experience reuse across tasks.

B.3 Pipeline Efficiency

We study online self-evolution over long-horizon task streams. To maintain the continuous feedback loop in which experience learned from earlier execution informs later decisions, newly acquired experience should become available before subsequent retrieval, so that it can be considered rather than being bypassed. Under task-level updating, however, such experience becomes available only after the source task completes, making task completion the experience-availability boundary for cross-task propagation. Evorelay moves this boundary to completed-stage updates: once a stage update finishes, its record becomes available to subsequent tasks while the source task continues to later stages.

To evaluate the effect of experience availability, we compare the task-completion and stage-update variants of Evorelay. Both variants trigger asynchronous extraction at stage completion, using the same extraction method and scheduling policy. The stage-experience representation, retrieval rule, task order, and execution resources are fixed. The variants differ only in when new stage records become visible to subsequent tasks. In the task-completion variant, records become available after source-task completion. In the stage-update variant, each record becomes available after its stage update, allowing stages from different tasks to overlap.

SWE-bench Verified uses four functional stages, and both variants have the same total concurrency budget of four workers. The experience extractor uses the same backbone model as the acting agent to assess the completed stage trajectory against its declared objective and distill transferable experience into a stage record. Agent execution and experience extraction share these four workers. Every model request from either operation must occupy one of the four shared concurrency slots, limiting the total number of concurrent backbone-model calls to four. End-to-end timing includes task execution, experience retrieval, experience extraction, memory updates, and associated waiting time.

As shown in Table 3, moving the experience-availability boundary from task completion to completed-stage updates reduces end-to-end task-stream time from 48.7 to 16.8 hours, a 2.90×

speedup. Pass@1 remains similar across the two variants (84.4% vs. 84.2%). This comparison holds the experience representation, extraction and retrieval mechanisms, task order, and total concurrency budget fixed, changing only when stage records become visible to subsequent tasks and thereby isolating the effect of experience availability on end-to-end execution efficiency.

C Experimental and Reproducibility Details

C.1 Benchmarks and Backbone Settings

We evaluate Evorelay on SWE-bench Verified and AppWorld. Table 6 summarizes the benchmark settings and agent scaffolds used in our experiments.

Table 6: **Benchmark and evaluation settings.** TGC denotes task goal completion.

Benchmark	Evaluation set	Tasks	Metric	Agent scaffold	Per-task budget
SWE-bench Verified	Official Verified set	500	Pass@1	mini-SWE-agent	250 agent steps
AppWorld	`test_normal`	168	TGC	ReAct	40 actions

SWE-bench Verified uses the official patch evaluator to determine whether a generated patch passes the target tests. AppWorld (Trivedi et al., 2024) uses the official task-goal evaluator for cross-application interactions.

We evaluate four backbone models: Gemini 2.5 Pro, Gemini 2.5 Flash, GPT-5.2, and DeepSeek V4-Pro. Their settings are summarized in Table 7.

Table 7: **Backbone model settings.**

Backbone	`max_tokens`	Reasoning setting	Temperature
Gemini 2.5 Pro	32,768	`high`	0
Gemini 2.5 Flash	32,768	`high`	0
GPT-5.2	32,768	`high`	0
DeepSeek V4-Pro	32,768	`high`	0

All backbone models use the high reasoning setting and a temperature of zero to reduce variation from generation sampling.

C.2 Benchmark-Specific Functional Stages

Long-horizon tasks often unfold through functional stages with distinct local objectives. We claim that experience extraction, retrieval, and cross-task propagation in long-horizon tasks should be organized at the functional-stage level. We do not assume a fixed stage partition across domains: stage categories are adapted to each environment's interaction structure, and the agent dynamically declares its current stage during execution. Table 8 summarizes the stage sets used in our experiments.

Table 8: **Functional stages used for each benchmark.**

Benchmark	Functional stages
SWE-bench Verified	Analyze, Reproduce, Edit, Verify
AppWorld	Explore, Authenticate, Query, Execute, Verify

For SWE-bench Verified, the stages capture the major functional phases of software issue resolution. For AppWorld, they characterize the major functional phases of cross-application task execution. Stage boundaries and execution paths follow the objectives pursued during each task rather than a fixed complete workflow.

C.3 Experience Extraction and Retrieval Configuration

For each backbone setting, Evorelay uses the same backbone model as the acting agent to extract experience from completed stages. The extractor receives the stage declaration and completed trajec-

tory and produces reusable atomic experiences, or an empty result when no transferable experience can be identified.

For retrieval, we use `sentence-transformers/all-MiniLM-L6-v2` as the text encoder. When entering a new functional stage, Evorelay first filters historical stage records by functional category and then ranks records within the same category according to the similarity between their stage objectives:

$$m_{i,j}^{*} = \arg\max_{m \in \mathcal{A}_{i,j}} \text{Cosine}\left(E(o_{i,j}), E(o_m)\right), \tag{13}$$

where E denotes the `sentence-transformers/all-MiniLM-L6-v2` encoder, $o_{i,j}$ is the current stage objective, and o_m is the objective stored in historical record m. The text representation of each objective contains its technical description and retrieval keywords, which are encoded together for similarity computation.

Retrieval is performed over complete stage records. Once a record is selected, its atomic experiences are provided together to the acting agent as guidance for the current stage.

Evorelay currently maintains an append-only memory bank. For substantially longer task streams, pruning, indexing, and other memory-management mechanisms may further improve the scalability of experience reuse.

C.4 Matched Comparison Protocol

For each backbone–benchmark combination, we construct three random task orders and apply the same three orders to all compared methods. Every run starts from an empty memory bank, after which each memory-based method continuously accumulates and reuses experience throughout the task stream.

All methods share the same backbone model, tools, environment, task order, per-task execution budget, and non-memory configurations. On SWE-bench Verified, each method uses a shared pool of four workers for agent execution and experience extraction. Every LLM request from either operation must acquire one of these four shared concurrency slots before being issued, limiting their combined LLM-request concurrency to four. We report the mean task success rate and sample standard deviation across the three task orders. Best@3 denotes the highest task success rate among the three orders.

End-to-end task-stream time is measured from the start of the task stream until all task executions, experience extraction jobs, and memory updates are complete. It includes agent execution, experience retrieval, experience extraction, memory updates, and associated waiting time.

The ablation studies follow a single-factor substitution protocol. The stage-decomposition ablation retains functional-stage declarations and transitions while disabling experience retrieval and memory updates. The experience-representation ablation keeps stage boundaries, retrieval, and other settings fixed while changing only the representation stored for each completed stage. The retrieval-policy ablation keeps the stage-experience representation, extraction method, experience-availability rule, task order, and other settings fixed, varying only the retrieval policy. Global-task retrieval first selects the most similar historical task and uses its record for the current stage, whereas stage-objective retrieval selects the most relevant same-category stage record according to the current stage objective. The scheduling ablation keeps the stage records, retrieval rule, task order, and other experimental settings fixed while changing only when newly acquired stage experience becomes available to subsequent tasks and, consequently, when subsequent tasks can progress.

C.5 Complete Per-Backbone Results

Figure 9 expands the SWE-bench summary in Figure 3 to all backbone models on both benchmarks.

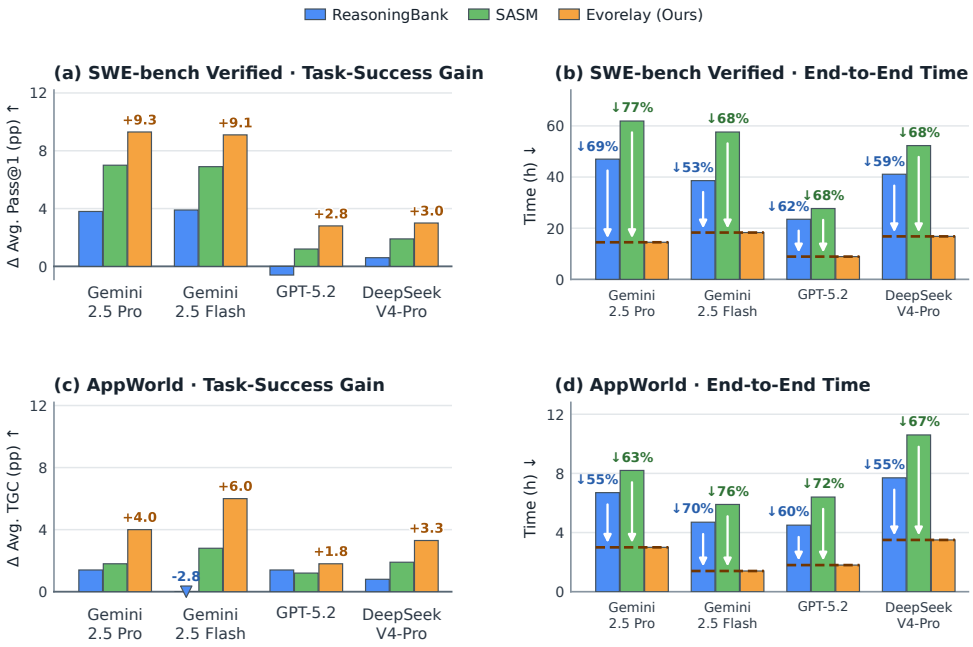


Figure 9: **Full per-backbone task-success gains and end-to-end runtime.** Panels (a,c) show Avg. Pass@1 and Avg. TGC gains (pp) over the same-backbone Vanilla Agent, calculated from the one-decimal averages displayed in Table 1; the Gemini 2.5 Flash ReasoningBank AppWorld gain (-2.8 pp) is labeled with a downward marker at zero; panels (b,d) show end-to-end task-stream time, including retrieval and memory-update costs. Colored labels show Evorelay's time reductions relative to each memory baseline; white arrows end at the dashed Evorelay runtime level.

D Additional Mechanism Analyses

D.1 Stage Structure and Experience Reuse

The stage-decomposition ablation isolates the effect of explicit stage segmentation itself, independent of experience retrieval and memory updates. This variant retains the same stage declarations and transitions as Evorelay while disabling memory retrieval and updates.

As shown in Table 2(a), stage segmentation alone provides only a modest improvement over the Vanilla Agent, whereas the full Evorelay framework yields a substantially larger gain. This indicates that the main performance improvement comes from accumulating and reusing stage-level experience rather than from stage segmentation itself.

D.2 Stage-Experience Representation

The representation ablation examines how a completed stage should be stored as reusable experience while keeping the stage boundaries, retrieval procedure, and other experimental settings fixed. We compare the raw stage trajectory, a single stage summary, and a stage record containing multiple atomic experiences.

As shown in Table 2(b), abstracting the raw trajectory into a stage summary improves transfer, while representing the stage as multiple atomic experiences provides a further gain. These results suggest that removing execution-specific details facilitates reuse, while separating distinct lessons within a stage helps preserve each lesson together with the conditions under which it is applicable.

D.3 Stage-Wise Behavioral Analysis

To characterize stage-specific behavior under Evorelay, we measure four observable behavioral indicators corresponding to Analyze, Reproduce, Edit, and Verify. For each backbone and method, let $\mathcal{T}$ denote all evaluated task trajectories. For an indicator q, we define its empirical rate over $\mathcal{T}$ as

$$\mathrm{Rate}(q) = \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \mathbb{I}[\text{trajectory } i \text{ satisfies the operational criterion for } q]. \tag{14}$$

Each indicator is determined from directly observable execution events in the trajectory. Reference patches are used only for this post-hoc behavioral analysis and are never exposed to the acting agent, retrieval module, or memory extraction process.

Table 9: **Stage-wise behavioral indicators on SWE-bench Verified with Gemini 2.5 Pro and DeepSeek V4-Pro.** Rates are percentages of evaluated trajectories; Δ denotes the absolute gain in percentage points.

Stage	Behavioral indicator	Observable criterion	Gemini 2.5 Pro			DeepSeek V4-Pro		
			Vanilla	Evorelay	Δ	Vanilla	Evorelay	Δ
Analyze	Target localization	Reference-modified file located before the first source edit	74.6	**94.2**	**+19.6**	96.0	**98.4**	**+2.4**
Reproduce	Failure reproduction	Executable reproducer, diagnostic, or test run before the first source edit	14.4	**43.2**	**+28.8**	83.4	**93.4**	**+10.0**
Edit	Patch targeting	Reference-modified file included in the final patch	81.4	**85.8**	**+4.4**	90.8	**92.0**	**+1.2**
Verify	Solution verification	At least one test-like or verification command completed successfully after editing	57.6	**77.6**	**+20.0**	97.0	**97.8**	**+0.8**

Evorelay achieves higher rates on all four stage-wise behavioral indicators for both backbones. On Gemini 2.5 Pro, the gains range from 4.4 to 28.8 percentage points; on DeepSeek V4-Pro, they range from 0.8 to 10.0 percentage points. The improvements span pre-edit target localization and failure reproduction, patch targeting during editing, and post-edit solution verification. These results indicate that the effects of stage-aligned experience reuse extend across the major functional stages of task execution rather than being concentrated only on the final code edit.

D.4 Global-Task Ranking of Stage-Selected Records

For each nonempty stage retrieval, we rank all earlier tasks by global task similarity and record the rank of the task that supplies Evorelay's selected stage record. Task-level coverage at K is

$$\mathrm{Coverage@K} = \frac{1}{|\mathcal{Q}|} \sum_{(i,j) \in \mathcal{Q}} \mathbb{I}\Big[\mathrm{rank}^{\mathrm{task}}_{i,j}\big(\mathrm{source}(m^*_{i,j})\big) \le K\Big], \tag{15}$$

where $\mathcal{Q}$ is the set of nonempty stage-retrieval events.

Table 10: **Global-task ranking of stage-selected records on SWE-bench Verified with DeepSeek V4-Pro.** For each stage entry, we rank all earlier tasks by global task similarity and locate the source task of Evorelay's top-1 stage record.

Stage	Task cov.@1	Task cov.@5	Med. rank
Analyze	30.7%	64.3%	3
Reproduce	17.6%	43.1%	8
Edit	19.4%	42.1%	8
Verify	16.7%	40.6%	9
Overall	**21.1%**	**47.5%**	**6**

Table 10 shows that only 21.1% of selected stage records originate from the top-ranked historical task under global task similarity, and 47.5% originate from its top five. The median source-task rank is 6. This analysis quantifies how often stage-level retrieval selects historical sources that would not be prioritized by global similarity, highlighting the difference between global task matching and local stage-specific experience needs.

D.5 Long-Range and Cross-Repository Reuse

For a target task at stream position i that retrieves a record from source position r, the task distance is

$$\delta = i - r. \tag{16}$$

Table 11: **Temporal and repository scope of retrieved experience on SWE-bench Verified with DeepSeek V4-Pro.** Distance is the difference between target and source positions in the fixed task stream.

Stage	Events	Med. dist.	$\delta \geq 50$	$\delta \geq 100$	Cross-repo.	Max dist.
Analyze	499	82.0	65.3%	44.7%	20.8%	473
Reproduce	499	91.0	66.3%	47.7%	34.5%	484
Edit	499	94.0	68.3%	48.1%	32.7%	492
Verify	498	89.0	68.1%	46.0%	36.7%	463
Overall	**1,995**	**90.0**	**67.0%**	**46.6%**	**31.2%**	**492**

Across 1,995 nonempty retrievals, the median source–target distance is 90 tasks. A total of 67.0% span at least 50 tasks, 46.6% span at least 100, and the maximum distance is 492. Source and target belong to different repositories in 31.2% of retrievals, reaching 32.7–36.7% in Reproduce, Edit, and Verify.

These results show that stage-level experience is reused over substantial temporal distances and across repository boundaries, rather than being limited to nearby or same-repository tasks.

D.6 Diversity of Reused Experience

The 1,995 nonempty retrievals select 799 distinct stage records. After sorting records by retrieval frequency, the 100 most frequently selected records account for 36.4% of all retrievals. At the same time, 353 records are selected exactly once, representing 44.2% of all selected records.

The resulting distribution contains both a set of frequently reused records and a broad tail of records serving less common stage objectives.

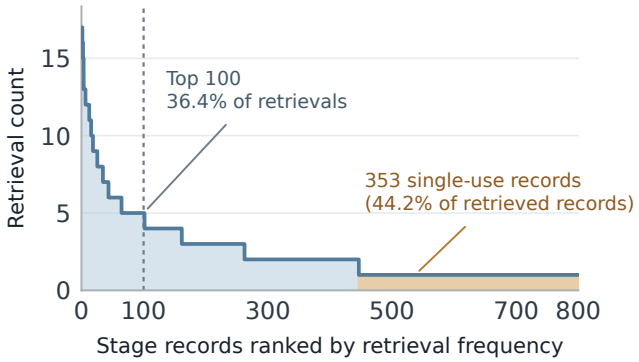


Figure 10: **Retrieval frequency of stage records.** The 799 distinct records selected in 1,995 nonempty retrieval events contain both repeatedly reused experience and a broad long tail.

D.7 Inference Cost

We measure average token consumption per task on SWE-bench Verified with DeepSeek V4-Pro. Action generation includes model calls used for agent–environment interaction; for memory-based methods, experience learning includes model calls used for experience judging or extraction, as applicable.

Table 12: **Average token consumption per task on SWE-bench Verified with DeepSeek V4-Pro.** Values are reported in thousands of tokens; – denotes no experience-learning calls.

Method	Action generation	Experience learning	Total
Vanilla Agent	1,254	–	1,254
ReasoningBank	1,242	54	1,296
SASM	1,336	22	1,358
Evorelay	1,282	37	1,319

Evorelay consumes 1.319M tokens per task, 1.8% more than ReasoningBank and 2.9% fewer than SASM. Its experience learning also uses fewer tokens than ReasoningBank's separate judging and extraction procedure. Evorelay therefore achieves its task-success and runtime gains with total token consumption comparable to the memory baselines.

We also report the average number of action steps per task for the same backbone and benchmark setting.

Table 13: **Average action steps per task on SWE-bench Verified with DeepSeek V4-Pro.**

Method	Avg. action steps
Vanilla Agent	41.96
ReasoningBank	42.21
SASM	43.90
Evorelay	42.80

D.8 Performance Across Interaction Complexity

We use the Vanilla Agent's number of environment actions as a fixed complexity proxy and partition the 500 SWE-bench Verified tasks into Easy ($\leq$ 27 actions, $n = 177$), Medium (28–47, $n = 156$), and Hard ($\geq$ 48, $n = 167$). The partition is shared by all methods, so a method's own execution length cannot change a task's group.

As shown in Figure 7, Evorelay improves Pass@1 over the Vanilla Agent by 1.1, 0.6, and 7.2 percentage points on Easy, Medium, and Hard tasks, respectively. The largest gain occurs on Hard tasks, suggesting that stage-aligned experience may be particularly useful in longer interaction trajectories with more intermediate decisions and state changes.

E Prompts and Qualitative Examples

E.1 Stage Declaration Prompt

The following excerpt specifies the stage-declaration format used by the acting agent on SWE-bench Verified.

Stage declaration (acting agent)

```
Before acting in a new stage, please output:
<stage>
{"domain":"<category>",
 "objective":{
   "description":"<current technical objective>",
   "keywords":["<2-5 retrieval terms>"]
 }}
</stage>

Keep the current stage across routine commands,
retries, and additional observations. Declare a new
stage only when the functional objective changes.
```

E.2 Atomic-Experience Extraction Prompt

Atomic-experience extraction

```
Extract reusable atomic experiences from one
completed functional stage. Use only actions that
were executed and observations that were obtained
in that stage.

Before extraction, assess whether the stage achieved
its declared objective using the stage trajectory.
If successful, extract effective strategies, why
they worked, and when to reuse them.
If unsuccessful, extract supported failure causes,
mistaken assumptions, and actions to avoid.

Retain an item only if it is supported by trajectory
evidence, transferable, recognizable in a future
situation, able to change a future decision, and
non-redundant with the other items.
Return multiple items when the stage supports
substantively different lessons. Return an empty
list when no item qualifies.

Use title for the transferable technical method.
In when_to_apply, state observable conditions
that would trigger reuse, not a past-task summary.
In experience, state what to do or avoid, why it
matters, and what a future agent should observe
to assess whether it worked. Exclude raw commands,
local paths, and unsupported claims.

Output item fields (JSON excerpt):
{"memory_items":[{
  "title":"...",
  "when_to_apply":"...",
  "experience":"..."
}]}
```

E.3 SWE-bench Execution Prompts

We use the following base system and task prompts for the mini-SWE-agent scaffold on SWE-bench Verified. The placeholder {{task}} is replaced with the current task description at runtime.

SWE-bench execution: system prompt

```
You are a helpful assistant that can interact with a computer shell
    to solve programming tasks.
```

SWE-bench execution: task prompt

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

<instructions>
```

```
# Task Instructions

## Overview

You're a software engineer interacting continuously with a computer by submitting commands.
You'll be helping implement necessary changes to meet requirements in the PR description.
Your task is specifically to make changes to non-test files in the current directory in order to fix the issue described in the PR description in a way that is general and consistent with the codebase.
<IMPORTANT>This is an interactive process where you will think and issue AT LEAST ONE command, see the result, then think and issue your next command(s).</IMPORTANT>

For each response:

1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
2. Provide one or more bash tool calls to execute

## Important Boundaries

- MODIFY: Regular source code files in /testbed (this is the working directory for all your subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg, etc.)

## Recommended Workflow

1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

## Command Execution Rules

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include reasoning text explaining what you're doing
- Your response MUST include AT LEAST ONE bash tool call. You can make MULTIPLE tool calls in a single response when the commands are independent (e.g., searching multiple files, reading different parts of the codebase).
- Directory or environment variable changes are not persistent. Every action is executed in a new subshell.
```

```
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write/load environment variables from files

Example of a CORRECT response:
<example_response>
I need to understand the Builder-related code. Let me find relevant files and check the project structure.

[Makes multiple bash tool calls: {"command": "ls -la"}, {"command": "find src -name '*.java' | grep -i builder"}, {"command": "cat README.md | head -50"}]
</example_response>

## Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require user input
- You can use bash commands or invoke any tool that is available in the environment
- You can also create new tools or scripts to help you with the task
- If a tool isn't available, you can also install it

## Submission

When you've completed your work, you MUST submit your changes as a git patch.
Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
Run `git diff -- path/to/file1 path/to/file2 > patch.txt` listing only the source files you modified.
Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source files you modified to fix the issue.
Do not submit file creations or changes to any of the following files:

- test and reproduction files
- helper scripts, tests, or tools that you created
- installation, build, packaging, configuration, or setup scripts unless they are directly part of the issue you were fixing (you can assume that the environment is already set up for your client)
- binary or compiled files
</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended changes and headers show `--- a/` and `+++ b/` paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not submit.
```

```
101
102 <CRITICAL>
103 - Creating/viewing the patch and submitting it MUST be separate
        commands (not combined with &&).
104 - If you modify patch.txt after verifying, you SHOULD verify again
        before submitting.
105 - You CANNOT continue working (reading, editing, testing) in any
        way on this task after submitting.
106 </CRITICAL>
107 </instructions>
```

E.4 Representative Stage Record

Figure 11 shows an atomic experience extracted from an Analyze stage. It separates a recognizable symptom from a diagnostic strategy: when a very small decimal is formatted in scientific notation despite a fixed number of decimal places, inspect the formatter's branch condition and probe adjacent inputs around the suspected cutoff. The item is not a summary of the original commands. It states when the lesson applies and how it should change a future analysis.

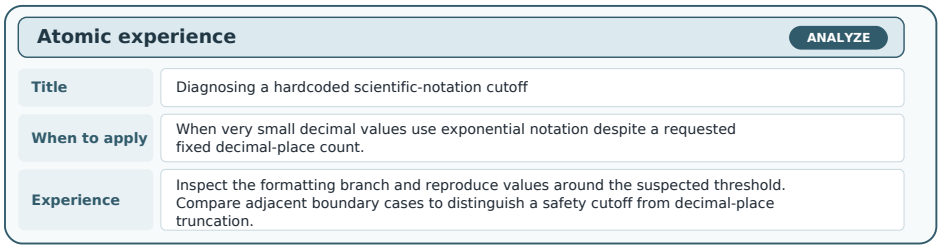


Figure 11: **Representative atomic experience from the DeepSeek V4-Pro memory bank.** The item separates its transferable lesson, applicability condition, and actionable diagnostic guidance.

E.5 Extended Analysis of the SymPy Case

Figure 8 contrasts the retrieved guidance for `sympy__sympy-23262`. ReasoningBank selects memories from `sympy__sympy-17655` about reflected multiplication and operator priority. Its patch modifies printer dispatch, but the tuple is assembled by `_recursive_to_string` in `sympy/utilities/lambdify.py`. The stage-aligned Analyze lesson explains why joining one element omits the required comma; the Edit lesson recommends a singleton special case. The successful patch adds that case without changing the shared list-handling path.