HOW SHOULD EXPERIENCE ACT? ADAPTIVE EXPERIENCE BINDING FOR AGENT SELF-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents are increasingly used to handle complex real-world tasks. To reduce the effort of manually designing and optimizing agents, recent work explored *agent self-evolution*, enabling agents to improve themselves toward target tasks automatically. Existing approaches either iteratively search for optimal agent configurations or accumulate and reuse experience to guide subsequent behavior. While the former often requires repeated rollout-and-evaluation cycles, the latter provides a more direct mechanism for preserving and exploiting execution experience. However, existing methods focus on what experiences to learn and retrieve, but leave a critical question largely under-explored: *how should learned experience act during execution?* Our measurement studies show that different ways of applying the same experience can substantially influence both agent behavior and end-to-end task performance. To address this gap, we propose **Adaptive Experience Binding (`AEB`)**, a training-free framework that explicitly regulates when and how learned experience influences agent behavior. `AEB` represents experience as structured and revisable Binding Experiences, learns and updates them from grounded execution evidence, and dynamically determines the appropriate form and strength of their influence according to the current execution state. Extensive experiments on complex agentic benchmarks demonstrate consistent improvements over representative baselines, with additional analysis experiments further supporting the effectiveness of the proposed design.

1 INTRODUCTION

Large Language Models (LLMs) are moving beyond conversation (Driess et al., 2023). With strong capabilities in planning, reasoning, and multimodal comprehension (Xi et al., 2025; Yao et al., 2022b; Xie et al., 2024), they now power agentic systems that tackle complex tasks across diverse domains, including agentic coding (Chen et al., 2021; OpenAI, 2025) and open-world gaming (Tan et al., 2025; Wang et al., 2023). Yet building these agents remains a craft. Prompts, workflows, tools, and inter-agent coordination all require manual effort and become a major bottleneck as the demand for task-specific agents grows (Wang et al., 2025a; Chen et al., 2024).

To tackle this problem, recent work explored agent self-evolution (Qian et al., 2024b; Zhou et al., 2024b; Gao et al., 2025), where agents improve themselves using past execution experience (recorded as trajectories and feedback (Yang et al., 2023; Li et al., 2026; Zhang et al., 2025a)). Existing approaches follow two paradigms. *Strategy-centric* methods such as AgentSquare (Shang et al., 2025) search the agent design space for configurations that suit the target task. These methods typically demand repeated rollout-and-evaluation iterations, guided by coarse task-level signals (*e.g.*, final task accuracy) (Li et al., 2026). *Memory-based* methods like ReasoningBank (Ouyang et al., 2025) distill prior interactions into reusable experience to directly guide subsequent execution.

Upon extracting and retrieving relevant experiences, these methods typically append them directly to the instruction prompt, leaving the agents to determine whether and how to utilize them. However, they neglect a crucial consideration: how a retrieved experience should subsequently be used to guide and affect agent behavior. Ideally, an agent self-evolution system should decide when, in what

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.

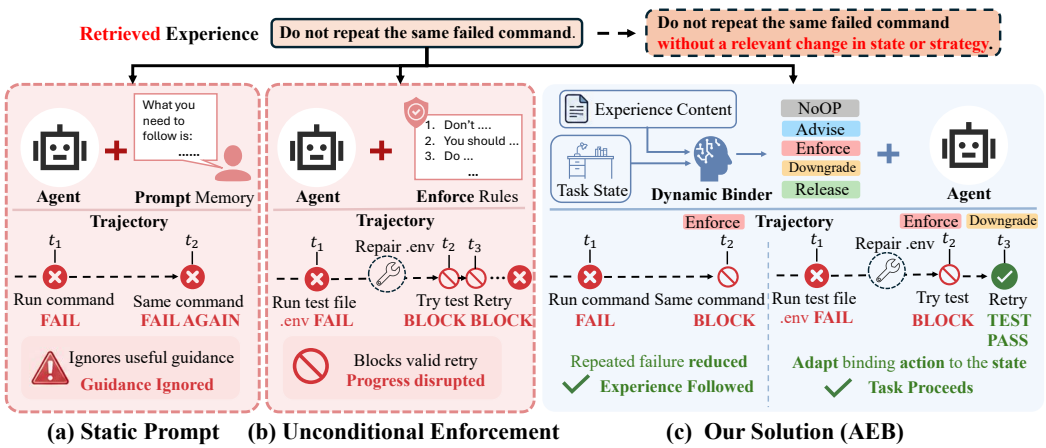


Figure 1: Why experience binding matters. Static prompt may ignore useful experience, whereas strict enforcement may apply incomplete experience or block valid actions. AEB instead dynamically determines the experience influence, balancing behavioral compliance and execution progress.

form, and to what extent a retrieved experience should influence subsequent agent execution. We term this runtime mechanism ***experience binding***.

To investigate whether *experience binding* matters, we conduct measurement studies (§ 3.1) on two agentic benchmarks. We fix the available experience and vary only how it is applied. We compare three common experience-adoption strategies and a rule-based strategy. In particular, *Static Prompt* appends experience as text and leaves adoption entirely to the agent. *Always Advice* repeatedly urges the agent to reconsider unmet experience. *Unconditional Enforcement* blocks any action that violates a retrieved experience and forces the agent to adopt it. The results expose two complementary failures (fig. 1 (a) & (b)). The first two strategies *under-bind* (*i.e.*, overlooking past experience): the agent often ignores relevant experience at the very moments when it could help. The third *over-binds* (*i.e.*, ruminating on the past): incomplete or only locally applicable experience often triggers endless debugging and stalls progress. Strikingly, even a simple experience-adoption rule (*e.g.*, dynamically considering the past experience influence and the current tasks' states during the runtime) can make substantial improvements (§ 3.1). These findings underscore that the runtime integration of *experience binding* can play a critical role in agent evolution.

To fill this gap, we propose **Adaptive Experience Binding (AEB)**, a memory-based agent self-evolution framework with an explicit runtime experience-binding mechanism. Rather than merely feeding agents with experiences, we augment each experience with a corresponding adoption strategy that explicitly regulates when and how to leverage such experience during execution and self-evolution. Intuitively, AEB acts like an experienced driving instructor, staying quiet when everything is going well, offering guidance when the car begins to drift, stepping in only when a collision is imminent, and gradually backing off as the student becomes a more confident and capable driver. We realize this through three components (fig. 1 (c)). We first define a **Binding Experience Paradigm** that summarizes each experience along with the corresponding structured, revisable adoption strategy. The **Binding Experience Learner** (§ 4.3) distills and stores execution trajectories into practical experiences following the paradigm. The **Dynamic Experience Binder** (§ 4.4) weighs each retrieved experience against the current state, proposed action, and recent history to decide whether it should remain inactive, advise, enforce, be downgraded, or be released. The binding process is used as fresh evidence to further update the experience store, sharpening the adoption strategies as experience accumulates. AEB thus moves agent self-evolution beyond learning what to remember, toward learning how stored experiences should guide decision-making and behavior. In summary, we make the following contributions:

- We identify *experience binding* as a missing yet critical dimension of memory-based agent self-evolution, and empirically reveal the complementary risks of under-binding and over-binding.
- We propose **Adaptive Experience Binding (AEB)** that learns structured Binding Experiences from grounded execution evidence and adaptively regulates their influence on agent behavior.
- Extensive experiments on two complex agentic benchmarks demonstrate significant improvements, with additional analysis experiments further supporting the efficacy of our key designs.

2 Related Work

LLM-Based Agents. LLM-based agents augment LLMs with structured reasoning and environmental interaction capabilities, enabling applications such as autonomous software development (Wang et al., 2025b; Hong et al., 2024; Qian et al., 2024a), tool-mediated user services (Yao et al., 2025; Schick et al., 2023; Qin et al., 2024), long-horizon task execution (Wu et al., 2024; Liu et al., 2024; Yao et al., 2022a), and scientific discovery (Su et al., 2025; Boiko et al., 2023). Despite their remarkable problem-solving capabilities, these systems often rely on task-specific workflows, limiting their adaptability and scalability (Nielsen et al., 2026; Zhu et al., 2025).

Agent Self-Evolution. In real-world applications, agents need to autonomously and continuously improve their behavior, strategies, or configurations in response to dynamic and ever-changing environments. Early efforts toward agent self-evolution mainly optimize isolated components, such as prompts (Yang et al., 2023), agent profiles (Yuan et al., 2025), and inter-agent connectivity (Zhuge et al., 2024). While reducing manual effort, such optimization limits the scope of agent adaptation.

Recent self-evolution methods can be broadly grouped into two types. ❶ *Strategy-centric* methods (Zhang et al., 2025a; Li et al., 2026; Zhang et al., 2025b) search for effective strategies or configurations: ADAS (Hu et al., 2024) explores agent designs in code space, while AgentSquare (Shang et al., 2025) evolves and recombines planning, reasoning, memory, and tool-use modules. They extend self-evolution to whole-agent strategies, yet execution feedback mainly guides subsequent search rather than directly shaping agent behavior, so improvement still rests on repeatedly generating and evaluating new configurations. ❷ *Memory-based* methods (Shinn et al., 2023; Zhao et al., 2024; Xu et al., 2025; Yu et al., 2026; Zhang et al., 2026) instead accumulate experience from past interactions to guide subsequent execution directly: AWM (Wang et al., 2024) abstracts successful trajectories into reusable workflows, and ReasoningBank (Ouyang et al., 2025) distills reasoning strategies from both successful and failed trajectories. Yet these advances primarily refine *what* experience is extracted and retrieved. Recent work further shows that inaccurate experiences can strongly alter agent behavior and propagate errors (Xiong et al., 2026), yet *how* such influence should be regulated remains largely implicit. `AEB` addresses this missing dimension via novel *experience binding*, explicitly regulating when and how learned experience acts at runtime.

3 Measurement Study & Problem Definition

3.1 Measurement Study

Study Goal and Controlled Setup. Our central hypothesis is that the effect of experience depends not only on *what* experience is available, but also on *how* it is applied during execution. To test this hypothesis, we conduct two complementary studies that fix the available experiences and vary only how they are adopted at runtime. We compare four experience-binding strategies. ❶ *Static Prompt* provides experience only in the initial prompt; ❷ *Always Advice* provides advisory feedback whenever relevant experience remains unsatisfied; ❸ *Unconditional Enforcement* blocks actions that violate applicable experience; and ❹ *Rule-based Dynamic Binding* dynamically selects among no intervention, advice, enforcement, and downgrade according to the current execution state. To balance experimental cost and runtime while maintaining a standardized and *controlled* setup, both studies use `GPT-4o-mini` (OpenAI, 2024b) as the execution model. To ensure fair comparisons, we keep the reusable experiences fixed during testing and *only vary the mechanism* by which they are integrated into the agent. Full definitions and implementation details are provided in § A.2.

Behavior-Level experiment on SWE-Bench-Verified. This benchmark aims to evaluate agents' ability to analyze code issues, navigate the codebase, and produce a functional patch. During execution, a task reaches *finalization* when the agent decides to submit its current solution. Note that reaching finalization does not mean the solution is correct (Jimenez et al., 2024). For this study, we manually specify three fixed experiences concerning *final-diff inspection*, *code validation*, and *avoiding repeated command failures*, and evaluate how different binding strategies affect compliance with these experiences. Detailed experience definitions and evaluation criteria are provided in § A.2.1 and § A.2.3. As shown in Tab. 1, Static Prompt finalizes on 70% of tasks but only verifies 5%, indicating that the three experiences are largely ignored. Always Advice raises verified finalization to 15%. Unconditional Enforcement fully eliminates unverified finalization, yet only 15% executions finalize, while 75% are terminated after exhausting the enforcement budget. These results

Table 1. Behavior-level effects of experience binding on the SWE-Bench-Verified benchmark.

Binding	Final. ↑	Verif. ↑	Unverif. ↓	Limit ↓
Static Prompt	70%	5%	65%	0%
Always Advice	55%	15%	40%	0%
Unconditional Enforce.	15%	15%	**0%**	75%
Rule-based Dynamic	**80%**	**45%**	35%	**0%**

Verif.: applicable pre-finalization requirements are satisfied;
Limit: execution terminates after the maximum number of allowed enforcement actions is reached.

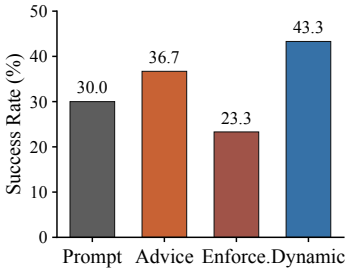


Figure 2. End-to-end experiment on WebArena.

reveal two distinct failure modes. Static Prompt and Always Advice *under-bind*: relevant experience is available yet barely shapes agent behavior when needed. Unconditional Enforcement *over-binds*: experience exerts excessive influence and obstructs valid progress. Rule-based Dynamic Binding, by contrast, finalizes 80% of tasks, with 45% verified, showing the efficacy of *dynamic binding*.

End-to-End experiment on WebArena. We next test whether these differences affect end-to-end task performance. We conduct a second study on WebArena to evaluate agents' ability in navigating and interacting with diverse websites. We first randomly select 100 shopping tasks from WebArena and use ReasoningBank to accumulate reusable experiences. Then, we evaluate the agent on another set of 30 shopping tasks. For each evaluation task, the retrieved experience is determined in advance and shared across binding conditions, so the comparison varies only how the same experience is applied during execution. Full settings of this study are detailed in § A.2.4. As shown in Fig. 2, Static Prompt succeeds on 9/30 tasks, Always Advice on 11/30, and Unconditional Enforcement on 7/30, while Rule-based Dynamic Binding achieves the highest success rate of 13/30. These results show that the benefit of runtime experience depends on how it is applied, with Rule-based Dynamic Binding performing best by adapting to the current execution state.

Takeaway. Experience binding matters, and fixed strategies are sub-optimal. *Static Prompt* and *Always Advice* can leave relevant experience ineffective, whereas *Unconditional Enforcement* can over-constrain execution and impede further progress.

3.2 Problem Definition

Memory-based Agent Self-Evolution. Following previous works (Wang et al., 2024; Ouyang et al., 2025), we consider an agent with policy $\pi_{\mathcal{L}}(\cdot \mid \mathcal{M}, \mathcal{A})$, where $\mathcal{L}$ is the backbone LLM, $\mathcal{M}$ is the experience store, and $\mathcal{A}$ is the action space. During execution, the agent sequentially tackles a series of tasks $\mathcal{Q} = \{q_1, \ldots, q_N\}$. Before q_i, the agent accumulates experiences into $\mathcal{M}_{i-1}$ from previous tasks. Executing q_i produces an interaction trajectory τ_i, after which the experience store is updated:

$$\mathcal{M}_i = \mathcal{U}(\mathcal{M}_{i-1}, \tau_i), \qquad \mathcal{M}_0 = \emptyset \tag{1}$$

where $\mathcal{U}$ denotes the experience learning process. Throughout this process, the backbone LLM remains unchanged, and self-evolution occurs through the continual accumulation and reuse of experiences. At execution step t of a current task q_i, a retrieval mechanism $\mathcal{R}$ selects potentially relevant experience $E_{i,t}$ from the current store $\mathcal{M}_{i-1}$ based on the interaction history $h_{i,t}$:

$$E_{i,t} = \mathcal{R}(\mathcal{M}_{i-1}, q_i, h_{i,t}), \tag{2}$$

Experience Binding. Experience learning determines what experience should be stored, while retrieval determines which stored experiences are relevant to the current execution context. However, neither specifies how the retrieved experience should influence agent execution. We formalize this missing process as *experience binding*. At execution step t of task q_i, given the retrieved experience $E_{i,t}$, the interaction history $h_{i,t}$, and the agent-proposed action $\tilde{a}_{i,t}$, a binding policy β determines

$$b_{i,t} = \beta(E_{i,t}, q_i, h_{i,t}, \tilde{a}_{i,t}), \qquad b_{i,t} \in \mathcal{B}, \tag{3}$$

where $b_{i,t} \in \mathcal{B}$ specifies whether, in what form, and to what extent the retrieved experience influences the current execution. For example, an experience $e \in E_{i,t}$ may be provided as advisory feedback when its guidance is relevant, or enforced by blocking a proposed action when the action may violate its requirement. The goal of experience binding is to ensure that relevant experience exerts sufficient influence when needed, while avoiding excessive intervention when it is unnecessary or potentially obstructive. Effective experience use thus requires state-dependent binding, motivating us to jointly learn structured experience and adapt its influence during execution.

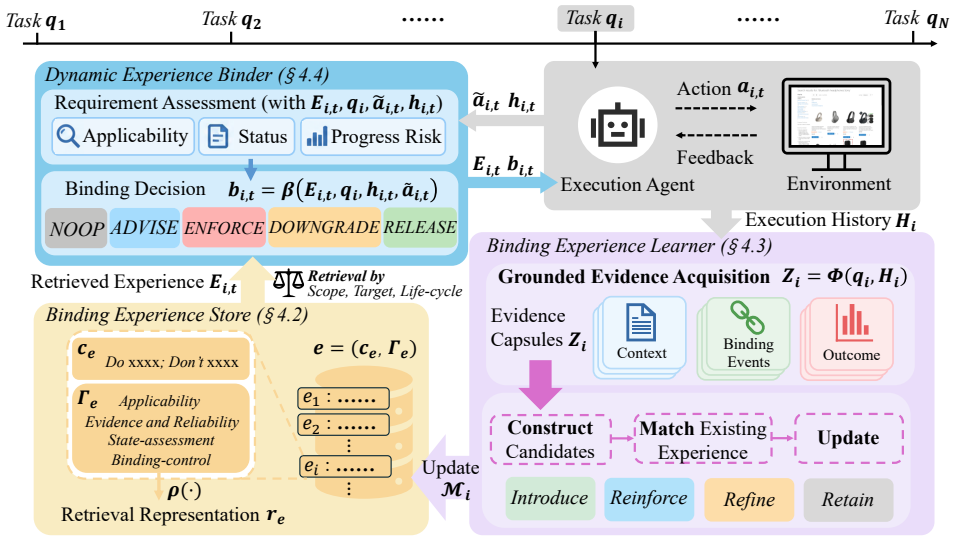


Figure 3: AEB forms a continual self-evolution loop that learns structured Binding Experiences from execution evidence and adaptively regulates how they influence subsequent agent decisions.

To this end, we propose **Adaptive Experience Binding (AEB)**, a training-free framework for memory-based agent self-evolution that jointly learns structured and revisable experiences from execution evidence and adaptively regulates how they influence subsequent agent behavior.

4 METHOD

4.1 OVERVIEW

Fig. 3 illustrates AEB, where experience binding is jointly realized by three tightly coupled modules. The *Binding Experience Paradigm* (§ 4.2) defines structured and revisable specifications describing when and how an experience may influence execution; the *Binding Experience Learner* (§ 4.3) constructs and updates them from grounded execution evidence; and the *Dynamic Experience Binder* (§ 4.4) applies them to retrieved experiences, selecting the form and strength of influence according to the current state and proposed action. Thus, the Binder makes runtime binding decisions based on structured information defined by the Paradigm and continually learned and updated by the Learner, allowing binding behavior to evolve with accumulated execution experience.

4.2 BINDING EXPERIENCE PARADIGM

We propose a *Binding Experience Paradigm* that augments the original experience with a structured, reusable binding specification (*i.e.*, the adoption strategy), which explicitly states how the experience should be used at runtime. Such a structured design explicitly regulates experience adoption rules, enabling more selective retrieval, state-dependent utilization, and dynamic revision.

Paradigm Definition. AEB defines each *Binding Experience* as a pair:

$$e = (c_e, \Gamma_e), \tag{4}$$

where c_e is the experience content containing reusable execution guidance and Γ_e is a structured binding specification that determines how c_e should be used at runtime, containing four types of information: *Applicability* specifies the conditions under which the experience is deemed relevant to the current context; *State assessment* defines the rule that determines whether the experience has already been satisfied based on the agent's current execution state; *Binding control* specifies which forms of intervention are permitted under different conditions; Finally, *Evidence and reliability* tracks the historical utilization of the experience, constraining the extent to which it influences agent execution. AEB also extracts coupled retrieval regulations from Γ_e corresponding to e:

$$r_e = \rho(\Gamma_e), \tag{5}$$

where ρ focuses on information about the experience scope and target, relevant execution phase, runtime signals, and accumulated reliability. Because r_e is derived directly from Γ_e, it is updated

automatically whenever the Binding Experience is revised. Note that c_e, Γ_e, and r_e are all learned and constructed by the Binding Experience Learner (§ 4.3) and can be continuously updated. A complete structured experience template is provided in § A.3.

Experience Ranking, Retrieval & Assessment. During runtime, AEB accumulates the structured experiences in a store for dynamic retrieval. At each decision point, we first filter out irrelevant experiences utilizing the structured retrieval regulations r_e. An experience is removed if the current task falls outside the scope of the task target or benchmark, or is not eligible for retrieval. We then grade the remaining experiences using a deterministic structured score:

$$s_{\text{ret}}(e; i, t) = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} s_f(r_e, q_i, h_{i,t}), \tag{6}$$

where $\mathcal{F}$ denotes a set of factors (detailed in § A.3) that assess the relevance of experience e to the current context, and $s_f(\cdot)$ is a scoring function that evaluates the compatibility between r_e and the current context with respect to f, producing a normalized score in $[0, 1]$. q_i is the current task context (*e.g.*, the task target and benchmark) and $h_{i,t}$ is the interaction history. We then rank the experiences based on their scores and further select experiences that are highly relevant to the current context based on Γ_e, q_i, and $h_{i,t}$. For each experience candidate, AEB checks its deterministic condition and assesses its applicability and satisfaction status. An experience is *applicable* when the current context satisfies the conditions under which its guidance is intended to apply, and it is *satisfied* when its requirement on agent behavior has already been fulfilled. For example, the repeated-failure experience becomes applicable once the same action fails repeatedly, and is satisfied once the agent changes its execution strategy. The resulting candidate set $E_{i,t}$, together with their assessed applicability and satisfaction status, is passed to the Dynamic Experience Binder (§ 4.4). § A.3 details the experience representation, retrieval score, and applicability assessment.

4.3 Binding Experience Learner

After each task, AEB distills agent execution trajectories into reusable experience following the structured paradigm (§ 4.2). However, a completed execution trajectory may mix useful behavioral patterns with failed attempts or incidental actions. Extracting experiences directly from a single execution trajectory to serve as persistent guidance across diverse tasks risks overfitting to task-specific patterns, thereby biasing subsequent execution and degrading overall generalizability. To avoid this, our agent-based *Binding Experience Learner* extracts grounded evidence from each completed execution, accumulates the evidence in a buffer, and dynamically determines whether new experiences should be created. Such an accumulation-triggered paradigm prevents overfitting to isolated trajectories, ensuring that the learned experiences remain generalizable across diverse tasks.

Grounded Evidence Acquisition. During task q_i, we collect the task execution history H_i when the agent operates on the previous experience store $\mathcal{M}_{i-1}$. H_i contains the agent's proposed and executed actions, environment feedback, runtime failures, and *Binding Events* produced by the Dynamic Experience Binder (§ 4.4). Following ReasoningBank, the learner uses an LLM-based judge to assign a post-task success/failure label from H_i. This label serves as a learning signal for experience construction and update, does not access benchmark ground-truth evaluation, and is not used as the reported task outcome. This judge uses the same foundation model as the corresponding experience learner. AEB then extracts a collection of grounded *Evidence Capsules* from H_i:

$$Z_i = \Phi(q_i, H_i). \tag{7}$$

The extraction function, $\Phi(\cdot)$, first partitions H_i into fine-grained salient events such as failures, recoveries, validations, and experience interventions. The learner then examines each event to extract structured descriptions and intermediate execution results, packs them into an event-specific Evidence Capsule, and accumulates the capsules in a buffer for further processing.

Evidence-Guided Experience Accumulation and Consolidation. Based on the accumulated capsule buffer, AEB constructs candidate Binding Experiences and matches them against the current store using their semantic identity and structured information. Each candidate is then processed through one of four update operations: *Introduce*, *Reinforce*, *Refine*, or *Retain*, depending on whether it represents a new experience, provides additional support, requires revision of the binding specification, or does not justify a store mutation. We summarize this post-task update as

$$\mathcal{M}_i = \Psi\big(\mathcal{M}_{i-1}, Z_i\big), \tag{8}$$

where Ψ is the experience learner that integrates agent-based evidence interpretation with the structured update rules described above to update the experience store. All store updates are committed only after task q_i completes, so the resulting store $\mathcal{M}_i$ becomes available starting from the next task. Detailed matching, update, and activation rules are provided in § A.4.

4.4 DYNAMIC EXPERIENCE BINDER

After retrieving a Binding Experience from the store, the Dynamic Experience Binder determines how it influences the agent's subsequent decision based on its detailed binding specification Γ_e (§ 4.2), considering its current applicability and satisfaction status, the agent-proposed action, accumulated experience support, and recent execution and binding history.

State-Dependent Binding. Following the formulation in § 3.2, let $E_{i,t}$ denote the experiences retrieved at decision step t, together with their applicability and satisfaction status from Γ_e. Given the current task q_i, interaction history $h_{i,t}$, and agent-proposed action $\tilde{a}_{i,t}$, the Binder determines

$$\begin{aligned} &b_{i,t} = \beta\big(E_{i,t}, q_i, h_{i,t}, \tilde{a}_{i,t}\big), \qquad b_{i,t} \in \mathcal{B} \\ &\mathcal{B} = \{\text{NOOP}, \text{ADVISE}, \text{ENFORCE}, \text{DOWNGRADE}, \text{RELEASE}\} \end{aligned} \tag{9}$$

where $b_{i,t}$ is the binding decision, including five modes specifying experiences' influences:

- NOOP: The retrieved experience does not intervene in the current decision.
- ADVISE: The experience is used as advisory guidance while the agent freely chooses next action.
- ENFORCE: The proposed action is blocked or redirected according to the experience requirement.
- DOWNGRADE: A previously strong intervention is relaxed to a weaker form, typically ADVISE.
- RELEASE: The experience stops influencing subsequent decisions in the current task.

Requirement Check & Intervention. Next, the Binder checks if the retrieved experience remains applicable and if its requirement has been satisfied. If an existing binding is RELEASEd and no binding is active, NOOP is applied. Missing execution signals or unresolved applicability likewise result in NOOP, preventing intervention when the current state provides insufficient evidence. If the experience has previously exerted strong control, repeated blocking, lack of useful progress, ambiguity, or pressure on the remaining intervention budget can instead trigger DOWNGRADE. After passing the check, ENFORCE is adopted if the proposed action meets the violation condition specified in Γ_e, the experience has accumulated the evidence and reliability required for enforcement, no relevant ambiguity or contradiction is present, and the constraint can be safely executed via the available action interface. The Binder applies ADVISE when the above enforcement conditions are unmet, even if an experience is applicable; otherwise, it applies NOOP. In this way, the same experience can exert distinct forms and strengths of influence as execution progresses rather than operating in a fixed intervention mode. The resulting binding decision is then applied to the proposed action $\tilde{a}_{i,t}$ before execution. Detailed learner-side prompts and runtime advice packets are provided in § A.5.

Binding Feedback. Each binding decision produces a *Binding Event* that records the involved experience, binding mode, the proposed action, the action ultimately executed, and the subsequent environment feedback. These events are incorporated into the task execution history H_i, which is further used by the Binding Experience Learner (§ 4.3) for experience store update. This closes the self-evolution loop: accumulated evidence updates where and how an experience may act, while subsequent binding decisions generate new grounded evidence for future updates.

5 EXPERIMENTS

5.1 EXPERIMENT SETTINGS

Following prior memory-based agent self-evolution studies, we evaluate `AEB` on the **WebArena** 684 tasks subset adopted from ReasoningBank, and **SWE-Bench-Verified**, containing 500 human-validated coding issues. We compare against two representative memory-based methods, **AWM** and **ReasoningBank**, together with a **NO-MEMORY** reference (§ A.6.2). Both memory baselines are evaluated on WebArena; following ReasoningBank, we exclude AWM on SWE-Bench-Verified because mini-SWE-agent's open-ended Bash action space does not readily support the fixed reusable workflows required by AWM. We conduct experiments with `GPT-4o-2024-08-06` (OpenAI, 2024a) and `DeepSeek-V4-Flash-2026-07-31` (DeepSeek-AI, 2026), using the corresponding backbone model for task execution and AEB's post-task experience-learning operations within

Table 2: Experiment results of AEB and baselines on WebArena. Success rate (SR↑) and number of steps (Step↓) are reported on five subsets with two backbone LLMs. Best are **Bold**.

Models	Shopping (187)		Admin (182)		GitLab (180)		Reddit (106)		Multi-site (29)		Overall (684)	
	SR(%)↑	*Step*↓	*SR(%)*↑	*Step*↓	*SR(%)*↑	*Step*↓	*SR(%)*↑	*Step*↓	*SR(%)*↑	*Step*↓	*SR(%)*↑	*Step*↓
GPT-4o												
NO MEMORY	35.83	9.93	36.26	12.36	28.89	16.14	32.08	10.84	0	**13.86**	32.02	12.52
AWM	32.62	**9.89**	34.62	12.75	30.55	14.66	37.74	10.32	0	15.59	32.02	12.21
REASONINGBANK	36.90	11.72	37.91	13.15	31.67	14.86	47.17	**7.67**	0	17.03	35.82	12.53
AEB(OURS)	**38.50**	10.58	**40.11**	**10.59**	**35.00**	**12.89**	**51.89**	8.92	**3.45**	16.10	**38.45**	**11.17**
DeepSeek-V4-Flash												
No Memory	45.99	10.11	53.30	12.99	46.11	13.39	38.68	11.42	3.45	24.93	45.03	12.57
AWM	36.90	**9.58**	43.96	10.96	44.44	10.34	50.94	10.95	0	**16.34**	41.37	10.65
REASONINGBANK	47.06	11.10	54.95	**10.27**	51.11	10.46	43.40	9.19	10.34	21.24	48.10	10.84
AEB(OURS)	**52.95**	11.53	**58.24**	10.94	**53.33**	**9.07**	**65.09**	**7.90**	**13.79**	20.14	**54.68**	**10.53**

Table 3: Results of AEB and baselines on SWE-Bench-Verified. Best are **Bold**.

Methods	Resolve Rate	Steps
GPT-4o		
NO MEMORY	26.2%	47.2
REASONINGBANK	28.4%	45.6
AEB(OURS)	**35.8%**	**38.2**
DeepSeek-V4-Flash		
NO MEMORY	63.4%	74.0
REASONINGBANK	61.2%	71.9
AEB(OURS)	**83.8%**	**31.7**

Table 4: Experience Impact Profile results of AEB and baselines using DeepSeek-V4-Flash on WebArena and SWE-Bench-Verified. Best are **Bold**.

Methods	Rescue ↑	Deterioration ↓	Preservation ↑
WebArena			
AWM	67	92	216
REASONINGBANK	78	57	251
AEB(OURS)	**101**	**35**	**273**
SWE-Bench-Verified			
REASONINGBANK	48	59	258
AEB(OURS)	**118**	**16**	**301**

each setting; model-selection details are provided in § A.8. On WebArena, all methods use the same BrowserGym environment and AgentLab execution framework (Chezelles et al., 2024) with the official evaluator (Zhou et al., 2024a); on SWE-Bench-Verified, we follow ReasoningBank and use the same mini-SWE-agent scaffold (Yang et al., 2024). We report two primary metrics: task success (success rate on WebArena and resolve rate on SWE-Bench-Verified) and the average number of execution steps per task. All methods follow the same self-evolution protocol in which each task can access only experience accumulated from previous tasks. We report details of our experiment settings and baseline implementation in § A.6.

5.2 OVERALL PERFORMANCE

Overall Task Performance. Tab. 2 and Tab. 3 compare AEB with the main baselines on WebArena and SWE-Bench-Verified, respectively. On WebArena, AEB consistently outperforms baselines under both backbone models. With GPT-4o, AEB reaches 38.45% overall success rate, surpassing 35.82% of ReasoningBank and 32.02% of AWM and NO-MEMORY, while reducing the average execution length to 11.17 steps. With DeepSeek-V4-Flash, AEB further achieves 54.68% success rate and 10.53 average steps, outperforming ReasoningBank by 6.58 percentage points. Notably, AEB obtains the highest success rate on every WebArena subset under both models, showing consistent benefits across heterogeneous web-interaction tasks.

In comparison, existing memory-based methods provide less stable gains. ReasoningBank moderately improves over NO-MEMORY, whereas AWM matches or even underperforms NO-MEMORY in several settings. This is consistent with our measurement study in § 3.1: learning reusable experience alone does not ensure effective adoption, as useful experience may be ignored while valid or irrelevant experience may interfere with execution. By explicitly regulating when and how learned experience should act, AEB turns experience reuse into a more reliable source of improvement rather than an uncontrolled source of behavioral bias. The advantage becomes more obvious on SWE-Bench-Verified when the agentic task becomes complex and long-horizon. With GPT-4o, AEB improves the resolve rate from 28.4% with ReasoningBank to 35.8%, while reducing average steps from 45.6 to 38.2. With DeepSeek-V4-Flash, AEB reaches 83.8% resolve rate, a 22.6 percentage-point absolute improvement over ReasoningBank, while requiring only 31.7 steps on av-

erage, 44.1% of ReasoningBank's average execution length. The larger gains on this longer-horizon benchmark suggest that regulating experience becomes increasingly important, as longer execution introduces more intermediate decisions and therefore more opportunities for inappropriate experience use, making it challenging for agents to make intermediate decisions without our dynamic binding mechanism. The consistent gains across two distinct model families further indicate that the benefit of AEB is not specific to a particular LLM backbone.

Experience Impact Profile. To examine how memory changes task outcomes, we define an Experience Impact Profile (EIP) with three categories: *Rescue* (memory turns a failure into success), *Deterioration* (memory turns a success into failure), and *Preservation* (both succeed). As shown in Tab. 4, AEB achieves 101 rescues with only 35 deteriorations on WebArena, compared with 78/57 for ReasoningBank and 67/92 for AWM. On SWE-Bench-Verified, AEB achieves 118/16 compared with 48/59 for ReasoningBank, while also preserving more NO-MEMORY successes. These results show that effective memory use requires both rescuing failed executions and avoiding disruption to successful ones, supporting our key motivation that experience influence should be selectively bound to both experience itself and the execution state. Further efficiency analysis is provided in A.7.

5.3 BINDING STRATEGY ANALYSIS

To examine the role of the *Dynamic Experience Binder* in self-evolution, we replace it with fixed binding strategies while keeping the remaining framework and experimental setting unchanged. Specifically, whenever the Binder is invoked, the fixed variants always apply the corresponding binding mode, whereas AEB dynamically determines the appropriate intervention according to the current execution state. Each variant independently runs the complete self-evolution process, so differences in runtime behavior can further affect the experiences learned from subsequent trajectories.

Table 5: Performance of different experience-binding strategies with DeepSeek-V4-Flash on SWE-Bench-Verified.

Method	Resolve Rate	Steps
No-Mem	63.4%	74.0
Advise	72.6%	37.9
Enforce	19.2%	132.4
Full AEB	**83.8%**	**31.7**

As shown in Tab. 3 and Tab. 5, the fixed ADVISE variant already achieves a 72.6% resolve rate, outperforming both NO-MEMORY (63.4%) and ReasoningBank (61.2%). This variant retains the structured Binding Experience Paradigm and Learner while restricting runtime influence to advisory feedback, thereby providing strong benefits even without adaptive control over binding strength. More importantly, unlike prompt-based memory methods that expose retrieved experience to the agent from the beginning of execution, ADVISE delivers guidance only when the corresponding experience becomes relevant. This further highlights that experience binding concerns not only *how* strongly experience should influence behavior, but also *when* such influence should occur. Nevertheless, Full AEB substantially outperforms fixed ADVISE, reaching 83.8% resolve rate with only 31.7 steps on average. Advisory feedback can still be too weak when stronger intervention is required, while a fixed advisory policy cannot adaptively relax or release experience influence when it becomes unnecessary. In contrast, AEB dynamically adjusts the binding mode according to the current execution state. The opposite extreme, fixed ENFORCE, performs poorly, achieving only 19.2% resolve rate and requiring 132.4 steps on average. Inspection of its execution trajectories shows frequent rigid interventions and conflicts with otherwise valid progress, causing repeated blocking and prolonged execution. These results reinforce our central finding: effective experience use requires both timely intervention and adaptive control over the form and strength of that intervention.

6 CONCLUSION & DISCUSSION

We study a critical yet overlooked question in memory-based agent self-evolution: how should learned experience influence execution? Our measurement study shows that different experience adoption strategies can lead to substantially different behaviors and outcomes. Motivated by this, we propose AEB that equips experience with structured, revisable binding information and adaptively regulates its runtime influence. Experiments show consistent performance gains with lower execution costs. More broadly, our results suggest that effective agent memory depends not only on what experiences are learned, but also on how they are integrated into execution. As models and agent workflows continue to advance, how to better leverage learned experience while minimizing the negative interference of inappropriate experience use will remain a promising future direction.

AI USE STATEMENT

We used generative AI tools for language editing and polishing of the manuscript, as well as for assisting with the collection of potentially relevant references. Generative AI was not used to formulate the research ideas, design the methodology or experiments, implement the experimental code, conduct the experiments, or produce the initial manuscript drafts; these components were developed independently by the authors. We additionally used generative AI tools to assist with literature organization and the collection of potentially relevant references, with the aim of ensuring that our claims and experimental comparisons were considered against recent related work. All cited references were manually verified and read by the authors, and all baseline implementations and experimental comparisons were independently verified and conducted by the authors. We reviewed all AI-assisted content and take full responsibility for the final manuscript, including its claims, citations, experimental results, and conclusions.

REPRODUCIBILITY STATEMENT

We have made substantial efforts to support reproducibility of this work. Sections 3 and 4 provide detailed descriptions of the problem formulation, the design of AEB, and the corresponding mathematical abstractions used to formalize the proposed experience-learning and binding mechanisms. The experimental section reports the benchmark settings, compared methods, evaluation protocols, and model configurations used in our study. Additional implementation details, including the measurement-study setup, Binding Experience representation and retrieval, and experience construction and update procedures, are provided in the appendix. Together, these descriptions are intended to enable independent reproduction of both the proposed method and the experimental protocol. As this work is still under review and has not yet been publicly released, we defer the public release of the implementation until acceptance. Upon acceptance, we plan to release the complete code and necessary configuration files to facilitate reproduction and further research.

REFERENCES

Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992), 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. *Conference on Neural Information Processing Systems*, 2024.

De Chezelles, Thibault Le Sellier, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F Xu, Siva Reddy, Quentin Cappart, et al. The browsergym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv e-prints*, pp. arXiv–2507, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

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? In *International Conference on Learning Representations*, volume 2024, 2024.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, and Fengli Xu. Agentswift: Efficient llm agent design via value-guided hierarchical search. In *Association for the Advancement of Artificial Intelligence*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, 2024.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. In *The Fourteenth International Conference on Learning Representations*, 2026.

OpenAI. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024a.

OpenAI. Gpt-4o mini: Advancing cost-efficient intelligence, 2024b. URL `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`.

OpenAI. Introducing upgrades to Codex. `https://openai.com/index/introducing-upgrades-to-codex/`, September 2025.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *The Annual Meeting of the Association for Computational Linguistics*, 2024a.

Cheng Qian, Shihao Liang, Yujia Qin, Yining Ye, Xin Cong, Yankai Lin, Yesai Wu, Zhiyuan Liu, and Maosong Sun. Investigate-consolidate-exploit: A general strategy for inter-task agent self-evolution. *arXiv preprint arXiv:2401.13996*, 2024b.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Conference on Neural Information Processing Systems*, 2023.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic LLM agent search in modular design space. In *International Conference on Learning Representations*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Conference on Neural Information Processing Systems*, 2023.

Haoyang Su, Renqi Chen, Shixiang Tang, Zhenfei Yin, Xinzhe Zheng, Jinzhe Li, Biqing Qi, Qi Wu, Hui Li, Wanli Ouyang, et al. Many heads are better than one: Improved scientific idea generation by a llm-based multi-agent system. In *The Annual Meeting of the Association for Computational Linguistics*, 2025.

Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, et al. Lumine: An open recipe for building generalist agents in 3d open worlds. *arXiv preprint arXiv:2511.08892*, 2025.

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.

Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. Megaagent: A large-scale autonomous llm-based multi-agent system without predefined sops. In *The Annual Meeting of the Association for Computational Linguistics*, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2025b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 2025.

Junlin Xie, Zhihong Chen, Ruifei Zhang, Xiang Wan, and Guanbin Li. Large multimodal agents: A survey. *arXiv preprint arXiv:2402.15116*, 2024.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior. In *The Annual Meeting of the Association for Computational Linguistics*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Conference on Neural Information Processing Systems*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2023.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Conference on Neural Information Processing Systems*, 2024.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Conference on Neural Information Processing Systems*, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2022b.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. In *The Annual Meeting of the Association for Computational Linguistics*, 2026.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *The Annual Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, 2025.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025a.

Jiaquan Zhang, Chaoning Zhang, Shuxu Chen, Zhenzhen Huang, Pengcheng Zheng, Zhicheng Wang, Ping Guo, Fan Mo, Sung-Ho Bae, Jie Zou, et al. Lightweight llm agent memory with small language models. In *The Annual Meeting of the Association for Computational Linguistics*, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Association for the Advancement of Artificial Intelligence*, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024a.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, et al. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024b.

Runchuan Zhu, Bowen Jiang, Lingrui Mei, Fangkai Yang, Lu Wang, Haoxiang Gao, Fengshuo Bai, Pu Zhao, Qingwei Lin, Saravan Rajmohan, et al. Adaptflow: Adaptive workflow optimization via meta-learning. *arXiv preprint arXiv:2508.08053*, 2025.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *International Conference on Machine Learning*, 2024.

A Appendix

A.1 Notation Summary

Table 6: Summary of the main notation used in this paper.

Notation	Description
L	Backbone LLM of the execution agent.
$\pi_L(\cdot \mid \mathcal{M}, \mathcal{A})$	Agent policy parameterized by L, experience store $\mathcal{M}$, and action space $\mathcal{A}$.
$\mathcal{Q} = \{q_1, \ldots, q_N\}$	Sequential task stream; q_i denotes the i-th task.
$\mathcal{M}_i$	Experience store after completing task q_i, with $\mathcal{M}_0 = \emptyset$.
τ_i	Interaction trajectory produced when executing task q_i.
$\mathcal{U}$	Experience-store update process.
$h_{i,t}$	Interaction history of task q_i up to execution step t.
H_i	Complete execution history of task q_i.
$\mathcal{R}$	Experience retrieval mechanism.
$E_{i,t}$	Retrieved candidate experiences at step t of task q_i.
$\tilde{a}_{i,t}$	Action proposed by the execution agent before experience binding.
$a_{i,t}$	Action executed after applying the binding decision.
$b_{i,t}$	Binding decision at step t of task q_i.
$\mathcal{B}$	Binding-mode space: {NoOp, Advise, Enforce, Downgrade, Release}.
β	Dynamic experience-binding policy that determines $b_{i,t}$.
$e = (c_e, \Gamma_e)$	A Binding Experience consisting of experience content and its binding specification.
c_e	Reusable execution guidance contained in experience e.
Γ_e	Structured binding specification of experience e.
r_e	Structured retrieval representation derived from Γ_e.
$\rho(\cdot)$	Mapping from a binding specification Γ_e to retrieval representation r_e.
$s_{\mathrm{ret}}(e; i, t)$	Structured retrieval score of experience e for the current execution context.
$\mathcal{F}$	Set of factors used to compute the structured retrieval score.
$s_f(\cdot)$	Normalized compatibility score for retrieval factor $f \in \mathcal{F}$.
Z_i	Grounded Evidence Capsules extracted from task q_i.
$\Phi(\cdot)$	Grounded evidence acquisition process that constructs Z_i from (q_i, H_i).
$\Psi(\cdot)$	Binding Experience Learner that updates the experience store from grounded evidence.

A.2 Additional Details of the Measurement Study

A.2.1 Measurement Study Setup

The measurement study in § 3.1 is designed to empirically test our hypothesis that the effect of experience depends not only on *what* experience is available, but also on *how* it is applied during execution. To examine this hypothesis in a controlled setting, all binding conditions within each benchmark are evaluated on the same task set with the same base executor, execution model, environment configuration, and available experience, while varying only the runtime mechanism through which that experience influences agent behavior. All four conditions use `GPT-4o-mini` as the execution model. By controlling these factors, the study isolates the effect of experience binding from differences in experience learning or retrieval, and is therefore distinct from the full self-evolution setting evaluated in our main experiments.

Experience Specification on SWE-Bench-Verified. For the SWE-Bench-Verified study, we manually define three auditable experiences before running the binding comparison following Reasoningbank's memory design (Title, Description, Content) (Ouyang et al., 2025). These experiences are fixed throughout the study and capture three common execution requirements observed in software engineering tasks:

1. **Inspect Final Diff Before Submission** (`inspect_final_diff`). This experience becomes applicable once source files have been modified. It requires the agent to inspect the final code diff before terminating the task. The experience is considered satisfied if a `git diff`-type inspection is executed after the latest source modification, and violated if the agent proposes finalization after modifying source files without such an inspection.
2. **Validate Modified Code Before Submission** (`validate_modified_code`). This experience also becomes applicable after source files have been modified. It requires the agent to run an available test or validation command before finalization. It is considered satisfied once a validation command has been executed after the latest source modification, regardless of whether the validation itself succeeds. A violation occurs when the agent proposes finalization after modifying source files without having executed any validation.
3. **Avoid Repeating Failed Commands** (`avoid_repeating_failed_commands`). This experience becomes applicable when the same normalized command has already failed at least twice. It requires the agent to change its strategy rather than issue the identical failed command again. The experience is satisfied once the proposed command no longer repeats the previously failed command, and violated when the agent proposes the same failed command again without a relevant change in strategy or execution state.

Experience Preparation on WebArena. For the WebArena study, we use ReasoningBank to accumulate reusable experience from 100 Shopping tasks before the binding comparison. We then evaluate on a disjoint set of 30 Shopping tasks. Together, the 100 experience-accumulation tasks and 30 evaluation tasks cover a substantial portion of the 187 Shopping tasks in WebArena. For each evaluation task, experience retrieval is performed in advance, and the resulting experience is reused across all four binding conditions. This allows the study to compare different runtime applications of the same retrieved experience without introducing condition-specific differences in retrieval.

A.2.2 Binding Mechanisms

Under the controlled setup described above, we compare four binding mechanisms: *Static Prompt*, *Always Advice*, *Unconditional Enforcement*, and *Rule-based Dynamic Binding*. Below, we provide their exact runtime behavior used in the measurement study.

Static Prompt. The available experiences are appended to the agent's initial instruction as textual guidance before execution begins. Afterward, no additional experience feedback or action-level control is introduced during execution. The execution agent therefore decides on its own whether and how to follow the provided experience.

Always Advice. Always Advice uses a fixed advisory strategy whenever an available experience remains unsatisfied. When advice is triggered, the current decision is interrupted and the corresponding experience is returned to the agent as textual feedback, prompting it to reconsider how to proceed. The agent then generates a new action without being required to follow a particular action prescribed by the experience. Advice is issued when the experience first becomes applicable and may be repeated after a short cooldown while the requirement remains unsatisfied.

Unconditional Enforcement. When a proposed action is detected to violate an applicable experience, the action is prevented from reaching the environment and the corresponding experience is returned as enforcement feedback. This enforcement is applied whenever a violation is detected; otherwise, the proposed action proceeds without intervention.

Rule-based Dynamic Binding. Rule-based Dynamic Binding uses a simple predefined rule to determine how an applicable experience should influence the current decision. It introduces no additional LLM or learned policy. Instead, the rule considers whether the experience is currently applicable, whether its requirement has already been satisfied, whether the proposed action triggers its enforcement condition, and whether the same experience has repeatedly blocked previous proposals. Based on these signals, it selects among four binding behaviors:

1. **NOOP.** If no relevant experience is applicable, or the relevant experience is inactive or already satisfied, no binding action is applied and the proposed action proceeds without intervention.
2. **ADVISE.** If an applicable experience remains unsatisfied but the current proposal does not trigger its enforcement condition, the corresponding experience is returned as advisory feedback, prompting the agent to reconsider how to proceed.
3. **ENFORCE.** If the current proposal triggers the enforcement condition of an applicable experience, the proposed action is blocked from reaching the environment and the corresponding experience is returned as enforcement feedback.
4. **DOWNGRADE.** If the same experience has already caused repeated enforcement, a subsequent ENFORCE decision is relaxed to ADVISE. This prevents the same experience from repeatedly blocking execution while still keeping its guidance available to the agent.

When multiple experiences suggest different binding behaviors at the same step, stronger binding takes priority, with ENFORCE taking precedence over ADVISE.

Intervention Safeguards. To prevent indefinite blocking, we impose a finite enforcement budget on the binding controller. Executions that exhaust this budget are reported as *Limit* in table 1; the exact thresholds are given in § A.2.3.

A.2.3 SWE-BENCH-VERIFIED STUDY SETTINGS

Execution Settings. The SWE-Bench-Verified study is intended to analyze how different binding mechanisms affect agent behavior rather than to estimate overall benchmark performance. Following ReasoningBank (Ouyang et al., 2025), all conditions use the same mini-SWE-agent execution scaffold (Yang et al., 2024), the SWE-Bench-Verified environment, and `GPT-4o-mini` (OpenAI, 2024b) as the execution model with temperature set to 0. Each task is allowed at most 250 execution steps with a per-task cost budget of $3.0. For runtime binding mechanisms, we additionally introduce safeguards against repeated intervention. At most 10 advisory feedback events are delivered within one execution, and a single experience can trigger at most 2 enforcement blocks. To prevent enforcement from indefinitely blocking execution, a run is terminated after 5 blocked actions in total or 3 blocked final-submission attempts. These thresholds are fixed across all tasks and are not tuned on task outcomes.

Evaluation Metrics. Because the purpose of this study is to measure how binding changes execution behavior, we focus on process-level outcomes rather than SWE-Bench-Verified task success. *Finalization* records whether the agent reaches a final submission during execution. Among finalized executions, we distinguish *verified* and *unverified* finalization according to whether the applicable

Table 7: Additional diagnostics results for the WebArena measurement study under different binding mechanisms.

Binding Mechanism	Success	SR	Avg. Steps	30-step Timeouts	Trigger Rate
Static Prompt	9/30	30.0%	16.53	13	–
Always Advice	11/30	36.7%	16.00	14	5.0%
Unconditional Enforcement	7/30	23.3%	16.60	12	75.1%
Rule-based Dynamic Binding	**13/30**	**43.3%**	**13.73**	**9**	10.7%

pre-finalization requirements have been satisfied. Specifically, after source code has been modified, a finalization is verified only if the agent has both inspected the final diff and executed a validation command after the latest modification. Otherwise, the finalization is marked as unverified. This verification concerns compliance with the prescribed execution requirements and should not be interpreted as correctness of the resulting patch. *Limit* specifically denotes executions terminated by the binding safeguard after repeated enforcement exhausts the allowed blocking budget. Runs that do not finalize because of the base execution budget or other execution failures are treated as non-finalized but are not counted as *Limit*. The repeated-failed-command experience is analyzed through the agent's runtime behavior but is not itself part of the verified-finalization criterion.

A.2.4 WebArena End-to-End Study Settings

Experience Accumulation and Evaluation Split. The WebArena study uses tasks from the Shopping domain and examines whether the behavior-level differences observed on SWE-Bench-Verified translate into end-to-end task success. We first run ReasoningBank on 100 Shopping tasks to accumulate reusable experience. The resulting experience store is then fixed before the binding comparison. We use a disjoint set of 30 Shopping tasks for evaluation, ensuring that no evaluation task appears in the 100-task experience-accumulation set. For each evaluation task, we use the ReasoningBank retrieval mechanism to retrieve relevant experience once before running the four binding conditions and reuse the retrieved experience across all conditions. No new experience is learned from the 30 evaluation tasks during the measurement study. Therefore, differences in task outcomes reflect how the same retrieved experience is applied during execution rather than differences in experience learning or retrieval.

Controlled Binding Evaluation. All four conditions start from the same task instruction and environment state and use the same execution agent and model configuration. Runtime experience application follows the binding mechanisms defined in § A.2.2.

Evaluation Metric. We use the official WebArena task evaluator to determine end-to-end success. A task is counted as successful only when the evaluator judges that the requested goal has been completed. The primary metric reported in fig. 2 is therefore the success rate over the same 30 tasks for each binding condition. In addition, we record execution length and runtime binding events for diagnostic analysis in table 7, while these auxiliary statistics do not affect the success judgment. 30-step Timeouts denotes the number of executions that reach the maximum 30-step execution horizon. Trigger Rate denotes the fraction of execution steps at which the runtime binding mechanism is triggered; it is not applicable to Static Prompt, which introduces no runtime intervention.

A.3 Realization Details of Binding Experiences and Structured Retrieval

Binding Experience Realization. The representation $e = (c_e, \Gamma_e)$ introduced in § 4.2 is a functional abstraction of the structured experience record used in our implementation. The content component c_e captures the behavioral knowledge to be reused. It is instantiated by the reusable guidance and its associated behavioral requirement, together with an advice template used when the experience is presented to the agent as guidance.

The binding specification Γ_e contains the structured information required to determine when this guidance may be relevant, how its current status can be assessed, and what forms of intervention may be permitted. We organize this information into four functional groups. *Applicability information* describes where and when the experience may apply, including its benchmark scope, target

scope, relevant execution phase, and trigger conditions. *State-assessment information* specifies the runtime signals used to evaluate the experience, together with conditions for determining whether its guidance has been satisfied or whether a proposed behavior would violate it. *Binding-control information* specifies the admissible forms of intervention and the conditions under which influence may be strengthened, relaxed, or released. Finally, *evidence and reliability information* summarizes the accumulated support for the experience, including confidence, usage and utility statistics, control-related statistics, and lifecycle state.

Concretely, these groups are instantiated by fields including benchmark and target scope, execution-phase information, trigger conditions, observable features, satisfaction and violation conditions, recovery and binding policies, confidence, evidence-support and utility statistics, control-related statistics, and lifecycle state. The implementation additionally maintains provenance, version, and lineage information for auditing and store updates. These auxiliary fields are omitted from the main-text abstraction because they do not directly participate in the runtime binding formulation.

Illustrative Example. Table 8 provides a concrete instantiation of the Binding Experience paradigm. Consider an experience learned from repeated execution failures: after the same action repeatedly fails, the agent should change its execution strategy rather than retrying the identical action. Beyond this reusable guidance, the binding specification records the runtime signals that make the experience applicable, the conditions used to determine satisfaction or violation, the interventions that may be applied, and the accumulated evidence governing the strength of its influence. Consequently, the same experience can be retrieved, assessed, strengthened, relaxed, or released according to the current execution state rather than being injected as static text.

Structured Retrieval. The retrieval representation $r_e = \rho(\Gamma_e)$ selects the subset of structured attributes used to locate potentially relevant experiences during execution. Retrieval is performed at each decision point over the immutable experience store available to the current task. Experiences that are incompatible with the current benchmark, target, or lifecycle state are first removed.

The remaining experiences are ranked using the deterministic structured score defined in § 4.2. The current implementation uses nine context-dependent factors organized into four groups, together with one fixed neutral term. *Context relevance* covers benchmark scope, target compatibility, and execution-phase relevance. *Signal availability* captures coarse indicators of whether the trigger features and observable signals required by the experience are present in the current execution context. *Experience reliability* incorporates accumulated confidence, utility reliability, and prior successful use. *Control compatibility* reflects the estimated cost of applying stronger intervention. The remaining semantic-identity term is fixed to a neutral value in the current implementation and therefore does not affect candidate ordering. All factors are normalized to $[0, 1]$ and equally weighted. Runtime retrieval is deterministic and invokes neither an embedding model nor an LLM, and the human-readable guidance itself is not used for semantic-similarity matching.

The highest-scoring experiences form a bounded candidate set. Retrieval scores are used only to prioritize which experiences should be examined further and do not determine whether an experience currently applies or should influence the agent. For each retrieved candidate, `AEB` evaluates its declared applicability conditions against the current execution history and records whether the guidance is already satisfied. Proposal-dependent violation conditions are evaluated later by the Dynamic Experience Binder using the current proposed action $\tilde{a}_{i,t}$. The resulting candidates constitute $E_{i,t}$, with their assessed applicability and satisfaction status provided to the Dynamic Experience Binder for the state-dependent decision described in § 4.4.

A.4 Details of Binding Experience Construction and Update

Candidate Construction and Matching. Given the grounded Evidence Capsules Z_i extracted from task q_i, `AEB` first groups related capsules according to their benchmark and target scope and their observed violation or satisfaction pattern. For each group, AEB constructs a candidate Binding Experience using a fixed structured template, containing reusable guidance c_e and the corresponding binding specification Γ_e. The learner LLM then reviews each candidate against its grounded evidence and retains only sufficiently supported candidates. The candidate templates are not rewritten by the LLM. Here, the fixed template specifies only the structure of a Binding Experience rather than a predefined library of experience content; the concrete guidance and its associated binding infor-

Table 8: Illustrative Binding Experience for avoiding repeated failed actions. The example shows how reusable guidance is paired with structured information that governs its retrieval, assessment, and runtime influence.

Field	Illustrative Value
Experience Content c_e	
Reusable guidance	After the same action repeatedly fails, change the execution strategy rather than issuing the identical action again.
Advice template	The previous action has already failed repeatedly. Consider changing the current strategy before retrying.
Applicability Information	
Scope / Target	Tasks involving iterative environment interaction and executable actions.
Trigger condition	The same normalized action has failed repeatedly within the recent execution history.
Execution phase	Active execution after an unsuccessful action attempt.
State-Assessment Information	
Observable signals	Recent action identity, execution result, and failure history.
Satisfaction condition	The agent changes the action or adopts a meaningfully different execution strategy.
Violation condition	The agent proposes the same failed action again without a relevant change in execution state or strategy.
Binding-Control Information	
Permitted influence	ADVISE, ENFORCE, DOWNGRADE, and RELEASE, subject to the current evidence and execution state.
Recovery behavior	Relax stronger intervention when repeated blocking no longer produces useful progress.
Evidence and Reliability Information	
Evidence support	References to prior execution episodes exhibiting repeated failure, strategy changes, and subsequent outcomes.
Lifecycle / reliability	Updated according to accumulated support, contradictory evidence, and observed utility.

mation are instantiated from the accumulated execution evidence. Before any store update, selected candidates are further checked for structural consistency, valid evidence references, and provenance.

Each selected candidate is then matched against the Binding Experiences in $\mathcal{M}_{i-1}$. Matching first checks for an exact match of the semantic-identity key, which is constructed from the benchmark, target identities, and normalized behavioral requirement. Otherwise, AEB computes deterministic overlap over the candidate and stored experience in terms of their behavioral requirement, scope, applicability conditions, observables, and binding-related information. The overlap between two structured fields is computed from their normalized token sets using Jaccard similarity.

The resulting match is determined by fixed thresholds. A candidate supports an existing experience when its behavioral-requirement overlap and scope overlap are both at least 0.65. When the behavioral-requirement overlap is at least 0.65 but the scope overlap is below 0.45, a strict inclusion relation between their target scopes is used to identify potential generalization or specialization. A contradiction with behavioral-requirement overlap of at least 0.35 is treated as conflicting. Otherwise, a behavioral-requirement overlap of at least 0.35 indicates a related but distinct experience; if the maximum overlap over scope, applicability conditions, and observables is at least 0.5, the match is treated as ambiguous; all remaining cases are treated as having no compatible match.

Experience Update Decisions. Based on the resulting match and the accumulated evidence associated with the experience, AEB constructs the corresponding store mutation. For clarity, we group these mutation decisions into four high-level outcomes, summarized in Table 9.

- **Introduce.** If no compatible existing experience is found and the selected candidate passes the grounding and mutation checks, a new Binding Experience is created. A newly introduced experience enters the CANDIDATE lifecycle state and is not available for runtime retrieval until sufficient independent support accumulates.

Table 9: High-level outcomes of evidence-guided Binding Experience updates.

Outcome	Persistent Experience	Effect of New Evidence
Introduce	Create a new candidate $e = (c_e, \Gamma_e)$	Initialize evidence support and reliability
Reinforce	Preserve c_e, scope, and binding conditions	Add support and update reliability statistics
Refine	Preserve c_e when possible; revise relevant parts of Γ_e	Update scope, control information, or lifecycle
Retain	No Binding Experience mutation	Archive the grounded evidence

- **Reinforce.** If the new evidence supports an existing experience without requiring changes to where or how it applies, AEB adds the new evidence support and updates its reliability statistics while preserving its core guidance, scope, and binding conditions.
- **Refine.** If later evidence indicates that an existing experience should apply more narrowly or broadly, or that its runtime influence should be weakened, AEB revises the corresponding scope, reliability, binding-control, or lifecycle information while preserving the core guidance whenever possible. In the current implementation, refinement may specialize or generalize the target scope, update reliability and negative-evidence statistics, adjust escalation information or the preferred binding action, and change the lifecycle state.
- **Retain.** If no candidate is selected or no mutation is accepted, the grounded capsules remain archived while the content of the Binding Experience store is left unchanged.

Conservative Activation and Update. A single task may introduce a new Binding Experience, but the experience initially remains in the CANDIDATE state and cannot affect runtime execution. In our implementation, activation requires supporting evidence from at least two distinct tasks. Becoming active does not by itself permit ENFORCE. An experience can trigger ENFORCE only after accumulating a support count of at least six and reaching an applicability confidence of at least 0.8, while the current execution must also exhibit a clear violation, contain no relevant ambiguity or contradiction, provide an enforceable action constraint, and remain within the available control budget. These runtime conditions are further described in § 4.4.

Later evidence may increase or reduce the estimated reliability of an experience, refine its applicable scope or preferred binding behavior, or change its lifecycle state. Importantly, the accumulated support count itself is not decreased by negative evidence; instead, contradictory or unsuccessful evidence affects reliability, confidence, negative-evidence statistics, and lifecycle. All accepted mutations are applied only after task q_i completes, so the resulting experience store becomes available starting from the next task.

A.5 PROMPT TEMPLATES AND RUNTIME PROMPT AUGMENTATION

Apart from the trajectory-level self-judgment inherited from ReasoningBank (Ouyang et al., 2025), the current implementation of AEB introduces two additional bounded LLM-based semantic judgments for post-task experience learning. Experience retrieval, runtime applicability assessment, binding-action selection, multi-experience arbitration, and experience-store mutation are implemented deterministically and therefore do not involve separate LLM prompts. We provide the two AEB-specific learner-side prompt templates and the runtime advice packet below.

P.1: Grounded-Claim Elaboration

Role: You perform a bounded semantic judgment over a structurally identified evidence opportunity.

Input Data:
{TASK_CONTEXT}
{EPISODE_CONTEXT}
{FIXED_CLAIM_TYPE}
{EVIDENCE_EVENTS}

Task Description: Determine whether the provided execution evidence supports one grounded factual claim of the fixed claim type. The claim type and supporting execution events are fixed before the model is invoked.
If the evidence does not support such a claim, abstain. Otherwise, emit exactly one factual claim grounded only in the provided evidence. Do not judge its reusability, generalizability, future utility, persistent identity, or lifecycle operation.

Output Requirement: Return exactly one JSON object following the specified schema, containing either an abstention or one grounded factual claim with bounded confidence. Do not use Markdown fences.

P.2: Candidate Semantic Review

Role: You review whether a complete candidate Binding Experience is supported by its grounded evidence.

Input Data:
{TASK}
{SUPPORTING_EVIDENCE_CAPSULES}
{CANDIDATE_BINDING_EXPERIENCE}

Task Description: Determine whether the provided candidate Binding Experience is naturally supported by the grounded Evidence Capsules. The candidate has already been constructed by the system and must not be rewritten, expanded, or assigned a new identity. Do not fabricate additional evidence.

Output Requirement: Return exactly one of the following JSON objects:
{"decision": "ACCEPT"}
or
{"decision": "REJECT"}
Do not include additional fields or Markdown fences.

P.3: Runtime Advice Packet

Purpose: The Dynamic Experience Binder does not use a separate LLM to choose among NoOP, Advise, Enforce, Downgrade, and Release. These decisions are made deterministically from the structured runtime state. When Advise-level strength is selected, however, the corresponding experience guidance is appended to the ordinary execution-agent context through a dedicated advice packet.

Packet Template:

```
<AEB_ADVICE_PACKET>
  {
    "scope": "CURRENT_ACTION_ONLY",
    "binding_strength": {BINDING_STRATEGY},
    "experience": {RETRIEVED_EXPERIENCE},
    "guidance": {ADVICE_TEMPLATE_OR_REUSABLE_GUIDANCE}
  }
</AEB_ADVICE_PACKET>
```

Runtime Effect: The packet does not determine the binding action itself. It is inserted only after the Binder has selected Advise, and the execution model remains free to choose its subsequent action.

A.6 Detailed Experimental Settings

A.6.1 Benchmarks and Evaluation Protocols

WebArena. WebArena (Zhou et al., 2024a) is a realistic web-agent benchmark requiring multi-step interaction with functional websites. Following ReasoningBank (Ouyang et al., 2025), we evaluate

on the 684-task subset consisting of 187 Shopping, 182 Shopping Admin, 180 GitLab, 106 Reddit, and 29 Multi-site tasks. All compared methods use the same BrowserGym environment and AgentLab execution framework (Chezelles et al., 2024), together with the official WebArena evaluator. For WebArena, all methods process the 684 evaluation tasks in the same fixed order as a single continual stream. The task set, execution budget, task ordering, and environment configuration are kept identical across methods. Each method maintains only its own cross-task memory state, and task q_i can access only memory accumulated from preceding tasks.

SWE-Bench-Verified. SWE-Bench-Verified (Jimenez et al., 2024) contains 500 human-validated software-engineering issues drawn from real-world GitHub repositories. Following ReasoningBank (Ouyang et al., 2025), all compared methods use the same mini-SWE-agent execution scaffold (Yang et al., 2024) and the official test-based evaluation procedure. Following the standard mini-SWE-agent configuration adopted in this setting, each task is assigned the same $3.0 model-cost budget and 250 step limit for within-task agent execution across all compared methods. This execution budget is imposed by the mini-SWE-agent scaffold and is kept unchanged for NO-MEMORY, ReasoningBank, and AEB. For SWE-Bench-Verified, all methods process the 500 tasks in the same fixed order, without reordering tasks by repository. The execution environment, action interface, task budget, and evaluation procedure are shared across compared methods. As in WebArena, experience learned from task q_i is committed only after the task completes and becomes available starting from the subsequent task.

A.6.2 BASELINE IMPLEMENTATIONS

No-Memory. The NO-MEMORY reference preserves the standard execution capability of the underlying agent while removing persistent cross-task learned experience. It uses the same executor, environment, action space, task budget, execution model, and evaluation procedure as the corresponding memory-based methods. Specifically, NO-MEMORY performs no retrieval from previous tasks, injects no learned cross-task experience into the execution prompt, performs no post-task experience induction, and maintains no persistent learned memory across tasks. Importantly, the agent still retains the ordinary episode-level interaction history required for multi-step execution. Thus, NO-MEMORY refers specifically to the absence of persistent learned experience transferred across tasks.

Agent Workflow Memory (AWM). We reproduce AWM (Wang et al., 2024) based on its paper and publicly released implementation repository. Since the released AWM code does not directly support the standard WebArena evaluation protocol used in our experiments, we adapt its workflow-memory mechanism to our shared WebArena framework. Following the experimental setting of ReasoningBank, all compared methods are required to use the same BrowserGym environment, AgentLab execution framework, task set, action interface, task budget, model configuration, and official evaluator. We therefore adapt AWM to this common setting based on its original paper and codebase, while preserving its core workflow induction, retrieval, and reuse procedure.

ReasoningBank. We reproduce ReasoningBank (Ouyang et al., 2025) based on its publicly released implementation repository, with only minimal adaptations required to run it under the same execution and evaluation setting as the other methods.

A.6.3 EXECUTION ENVIRONMENT

All experiments are run on a local machine running Ubuntu 24.04.3 LTS, equipped with an Intel Core Ultra 7 265K CPU and 46 GiB of system memory. SWE-Bench-Verified tasks are executed locally in isolated Docker containers. WebArena services are deployed on AWS EC2 using the same instance configuration across all compared methods, with 1 TB of storage provisioned for the benchmark environment. The server configuration is kept fixed throughout the experiments.

LLM inference is accessed through the corresponding model-provider APIs and therefore does not rely on local GPU computation. Within each benchmark and model setting, all compared methods share the same execution infrastructure so that differences in task performance are not attributable to differences in local or server-side compute resources.

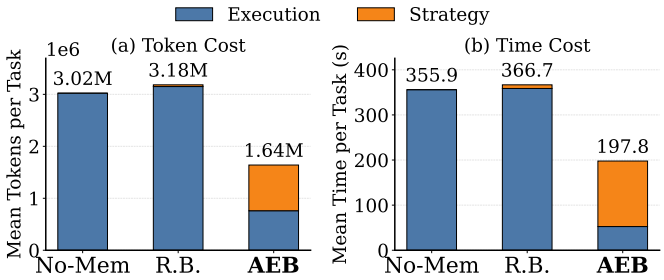


Figure 4: Comparisons of time and token costs on SWE-Bench-Verified across execution and strategy using `DeepSeek-V4-Flash` (R.B. means ReasoningBank).

A.7 Execution Efficiency

The fine-grained experience learning and novel runtime binding of `AEB` inevitably incur additional strategy-side computation, which may become more noticeable on complex, long-horizon tasks. We therefore further examine whether these additional mechanisms introduce substantial end-to-end cost by analyzing token and wall-clock time consumption on SWE-Bench-Verified with `DeepSeek-V4-Flash`. As shown in Fig. 4, `AEB` achieves substantially lower overall token and time costs than all baselines. Compared with ReasoningBank, `AEB` reduces the average token consumption from 3.18M to 1.64M and the average wall-clock time from 366.7 to 197.8 seconds. Interestingly, the cost breakdown shows that `AEB` incurs higher overhead at the strategy level, as expected from its more fine-grained experience learning and runtime binding. But more importantly, by applying experience more selectively and effectively, `AEB` guides the agent toward more efficient execution strategies, requiring substantially fewer interaction steps to complete tasks. Consequently, the additional strategy-side cost is more than compensated by the reduction in execution-side consumption, resulting in lower overall token and time costs.

A.8 Model Selection and Evaluation Consistency

Model Configuration. We evaluate `AEB` and all compared baseline methods with two execution models: `GPT-4o` (OpenAI, 2024a) and `DeepSeek-V4-Flash` (DeepSeek-AI, 2026). Within each model setting, the same backbone model is used for task execution and the post-task experience-learning process, including its learner-side semantic judgments and trajectory-level self-judgment. Specifically, we use `GPT-4o-2024-08-06` and `DeepSeek-V4-Flash-2026-07-31` in the corresponding experimental settings.

Why GPT-4o. We first select `GPT-4o` because it provides a strong and well-established reference point for evaluating LLM-based agents. GPT-family models, particularly models in the GPT-4 family, have been widely used in prior work on agent self-evolution, automatic agent construction, and experience-based agents, including AgentSquare (Shang et al., 2025), AFlow (Zhang et al., 2025b), MaAS (Zhang et al., 2025a), AgentSwift (Li et al., 2026), and AWM (Wang et al., 2024). Using GPT-4o therefore facilitates comparison with this line of work while providing a sufficiently capable model for the long-horizon interaction required by WebArena and SWE-Bench-Verified. In addition, both benchmarks have been extensively studied with GPT-4-class agents, making GPT-4o a natural reference model for evaluating whether improvements arise from the proposed experience-binding mechanism rather than from an unusually weak execution model.

Why DeepSeek-V4-Flash. We further evaluate a substantially newer model from a different model family to examine whether the effectiveness of `AEB` depends on GPT-specific behavior. We choose `DeepSeek-V4-Flash-2026-07-31` as a recent and efficient model from the widely adopted DeepSeek family. Compared with GPT-4o, it differs in model family, release period, and deployment characteristics, providing a complementary execution setting rather than a minor variation of the same model. Evaluating the same self-evolution and binding mechanisms under both models therefore allows us to test whether the benefits of `AEB` generalize across substantially different LLM backbones.

Learning-Time and Evaluation-Time Judging Explanation. We distinguish the judgment used for experience learning from the judgment used for benchmark evaluation. Following ReasoningBank (Ouyang et al., 2025), the post-task learner uses an LLM-based judge to assign a success/failure label to the observed execution trajectory. This label is only a proxy learning signal and is not treated as ground-truth task correctness. Within each model setting, the learner-side judge uses the same model as the corresponding experience learner.

Benchmark evaluation is performed independently through the corresponding benchmark harness and does not participate in experience learning. For evaluation components that require LLM-based judging, we keep the judging model fixed to `GPT-4o` across execution model settings to ensure a consistent scoring criterion. Benchmark-native deterministic or official evaluators are used whenever applicable. Consequently, benchmark evaluation outcomes are used only for final scoring and are never exposed to the learner.