AGENTEAES: EFFECT-AWARE EXPERIENCE SELECTION FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances have enabled large language models (LLMs) to act as agents capable of solving increasingly complex tasks in scientific research, office productivity, and software development. Prior work often improves agent performance at inference time by retrieving relevant experiences from past interactions. Although such experience-based augmentation offers a promising path toward continual adaptation, we identify a fundamental limitation of similarity-based retrieval in agentic settings. Highly similar experiences can degrade performance, even on tasks that the agent would otherwise solve successfully. This finding suggests that experience utilization is a decision problem rather than merely a retrieval problem. We propose effect-aware experience selection (AgentEAES), a framework that learns which experience to use from experience-augmented rollouts. AgentEAES trains a model to predict the sign of each task-experience pair's effect and selects the experience with the largest positive predicted effect. Experiments cover two agentic benchmarks with backbones of different strength, allowing us to evaluate both a strong backbone on a more challenging benchmark and a smaller model on an easier one. On AppWorld, with Qwen-3.7-Max, AgentEAES improves task success rates on the challenging domain by 13.3 percentage points. On WorkBench, even with the smaller Qwen3-30B-A3B backbone, AgentEAES still yields an improvement of up to 21.5 percentage points. The code is available at https://anonymous.4open.science/r/AgentEAES-BDBF.

1 INTRODUCTION

Large language model agents that iteratively plan, act, and observe in tool-augmented environments have demonstrated strong performance across a wide range of tasks (Yao et al., 2022; Shinn et al., 2023; Wang et al., 2024). A natural way to improve these agents is to provide relevant past experiences as in-context demonstrations (Luo et al., 2026). Existing approaches broadly follow three paradigms, including storage, reflection, and abstraction. They store trajectories for subsequent retrieval (Kagaya et al., 2024), refine experiences through self-feedback (Shinn et al., 2023; Ding, 2026), or transform experiences into reusable representations such as skills, strategies, and summaries of experiential knowledge (Cai et al., 2025; Fan et al., 2026; Zhang et al., 2026). Despite their practical appeal, these methods retain experiences solely as inference-time context. Their effectiveness is therefore constrained by the model's in-context learning capacity and may diminish as accumulated experiences lead to context collapse (Zhang et al., 2026). Nevertheless, context-based methods remain uniquely applicable to closed-source models without requiring parameter access, making them particularly important in agentic settings.

Most existing approaches retrieve experiences based on embedding similarity (Kagaya et al., 2024). Given a new task, the agent retrieves the top-k experience whose prompts are closest in the embedding space and incorporates them into its context. However, we identify a critical yet underexplored failure mode of this paradigm. Even highly relevant experiences can degrade performance on tasks that the agent would otherwise solve correctly. On the AppWorld benchmark (Trivedi et al., 2024), we empirically observe that top-k retrieval can hurt performance even when the retrieved experiences are highly similar to the current task. Appendix F provides a representative example. This finding challenges the common assumption that greater contextual relevance necessarily improves performance.

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.

We formulate experience utilization as an effect-aware decision problem rather than merely a retrieval problem, because applying an experience and re-executing the agent directly reveals whether it has a positive or negative effect on task performance. We first construct candidate experiences from informative historical agent trajectories. Because estimating the effect of each candidate requires an additional LLM rollout, exhaustive evaluation is prohibitively expensive. We therefore experimentally analyze the trade-off between the number of sampled experiences and final performance improvement. Based on these observations, we train an effect-aware model to select which historical experience to apply to a given task.

Our contributions are as follows. (i) We identify a failure mode of experience-augmented agents, showing that similarity-based retrieval can hurt performance on tasks that the agent would otherwise solve correctly. (ii) We reformulate experience utilization as an effect-aware decision problem, in which the agent selects a historical experience by its predicted effect rather than by similarity to the current task. (iii) We propose AgentEAES, a framework that learns an effect-aware model from sampled interventions to predict whether a candidate experience will be helpful or harmful. (iv) We design a sparse lexical representation for task-experience pairs and show, through the ablation study in section 5.4, that interaction features are critical, suggesting that the decision signal depends on the joint task-experience context.

Experiments on AppWorld and WorkBench further validate the effectiveness of AgentEAES. On AppWorld, with the strong Qwen-3.7-Max backbone, AgentEAES improves task success rates by up to 13.3 percentage points. On WorkBench, where the same strong backbone is nearly saturated (Styles & Miller, 2026), we instead evaluate using the smaller and earlier Qwen3-30B-A3B backbone, under this setting, AgentEAES also significantly outperforms retrieval-based baselines.

2 PROBLEM FORMULATION

Let $t \sim \mathcal{D}_{\mathcal{T}}$ be a task instance drawn from a distribution $\mathcal{D}_{\mathcal{T}}$ over the task space $\mathcal{T}$. Let $\mathcal{M}$ be a finite set of candidate historical experiences, each of which may be applied to t. We use $\varnothing$ to denote abstention, i.e., applying no historical experience. The action space is therefore $\mathcal{M} \cup \{\varnothing\}$.

Executing the agent on t under an action $m \in \mathcal{M} \cup \{\varnothing\}$ is stochastic, since the agent samples its plans and actions during decoding. We therefore model the outcome $Y_t(m)$ as a random variable whose randomness arises from execution stochasticity. We define the task-specific expected gain of applying m relative to abstention as

$$\tau(t, m) = \mathbb{E}[Y_t(m)] - \mathbb{E}[Y_t(\varnothing)] \,. \tag{1}$$

A policy is a mapping $\pi : \mathcal{T} \to \mathcal{M} \cup \{\varnothing\}$ that assigns each task either a historical experience or abstention. Let Π denote a class of such policies. Our objective is to find a policy $\pi \in \Pi$ that maximizes the expected gain over the task distribution

$$\pi^\star \in \arg\max_{\pi \in \Pi} \mathbb{E}_{t \sim \mathcal{D}_{\mathcal{T}}} \left[\tau\big(t, \pi(t)\big) \right] . \tag{2}$$

At inference time, decisions are made independently across tasks, with no shared resource or cross-task constraint. Thus, without loss of generality, Π can be taken as unrestricted, and the optimization in equation 2 decomposes pointwise over tasks. Hence, for each task t, an optimal action is any maximizer of $\tau(t, m)$ over $m \in \mathcal{M} \cup \{\varnothing\}$. Since $\tau(t, \varnothing) = 0$, one optimal policy can be written as

$$\pi^\star(t) = \begin{cases} m^\star, & \text{if } \max_{m \in \mathcal{M}} \tau(t, m) > 0, \\ \varnothing, & \text{otherwise,} \end{cases} \tag{3}$$

where $m^\star \in \arg\max_{m \in \mathcal{M}} \tau(t, m)$. In other words, the policy applies the historical experience with the largest positive gain and abstains whenever no candidate improves upon abstention.

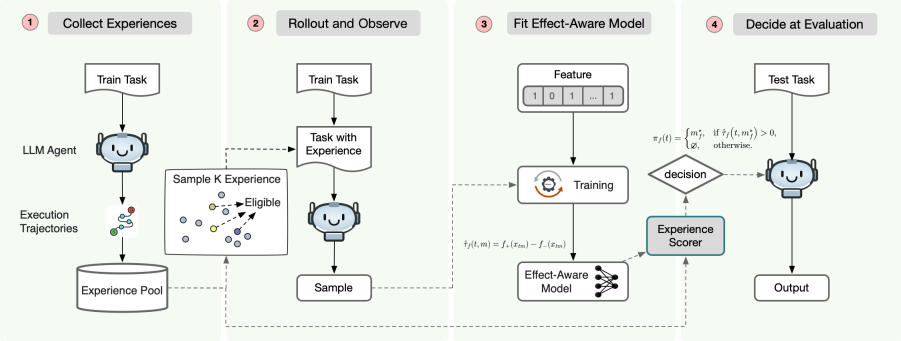

Figure 1: Overview of AgentEAES. Historical experiences are collected from the training set, sampled for re-execution feedback, and used to train an effect-aware model, which then guides test-time experience selection.

3 Method Overview

We begin with an overview of AgentEAES (Figure 1). On the training set, we first run an experience-free agent to collect a pool of historical experiences and record the abstention outcome $Y_t(\varnothing)$ for each task t. We then sample a small number of experiences per task, excluding experiences from the task itself, and re-execute the agent with each sampled experience inserted into the prompt. This produces the outcome $Y_t(m)$ and the observed gain $G_t(m) = Y_t(m) - Y_t(\varnothing)$. These observations are then used to train an effect model over task-experience pairs. At test time, the model scores candidate experiences from the training pool and retrieves one only if its predicted gain is positive.

4 Effect-Aware Model

In this section, we describe the design of the effect-aware model. Since the outcome in many agentic tasks can be naturally defined in terms of execution success, we assume binary outcomes, i.e., $Y_t(m) \in \{0, 1\}$. Our framework is not restricted to this setting, but the binary formulation is sufficient for the benchmarks considered here and leads to simple gain labels relative to abstention.

Once the historical experience pool is determined, the decision problem can be formulated as estimating, for each candidate experience, whether applying it yields a positive gain over abstention. We then define the gain of applying experience m relative to abstention as

$$G_t(m) = Y_t(m) - Y_t(\varnothing), \tag{4}$$

so that $G_t(m) \in \{-1, 0, 1\}$. By definition in equation 1, the expectation can be written as

$$\begin{aligned} \tau(t, m) &= 1 \cdot \Pr(G_t(m) = 1) + (-1) \cdot \Pr(G_t(m) = -1) \\ &= \Pr(G_t(m) > 0) - \Pr(G_t(m) < 0). \end{aligned} \tag{5}$$

In practice, we observe one rollout outcome for each sampled pair and use the realized gain as noisy supervision for whether the effect is positive or negative. For each task-experience pair (t, m), we compute a d-dimensional representation $x_{tm} \in \mathbb{R}^d$. Based on this representation, we parameterize the effect model $f(x_{tm}) = \big(f_+(x_{tm}),\ f_-(x_{tm})\big)$, where $f_+(x_{tm})$ and $f_-(x_{tm})$ are independently trained to estimate the marginal probabilities of positive and negative gain, respectively. We define the effect score

$$\hat{\tau}_f(t, m) = f_+(x_{tm}) - f_-(x_{tm}), \tag{6}$$

which serves as an estimate of the expected gain $\tau(t, m)$. Let $m_f^\star \in \arg\max_{m \in \mathcal{M}} \hat{\tau}_f(t, m)$. Since abstention yields zero gain, the induced policy applies $m_f^\star$ only if its score is positive, i.e.,

$$\pi_f(t) = \begin{cases} m_f^\star, & \text{if } \hat{\tau}_f\big(t, m_f^\star\big) > 0, \\ \varnothing, & \text{otherwise.} \end{cases} \tag{7}$$

When $\hat{\tau}_f(t, m)$ provides a reliable estimate of $\tau(t, m)$, the resulting policy π_f equation 7 is expected to make decisions that are well aligned with the optimal policy $\pi^\star$ equation 3.

Candidate Experiences. We construct the candidate experience pool from all training-set trajectories generated by the experience-free agent, including both successful and failed trajectories. For each successful trajectory, we apply the procedure described in appendix A to extract an additional reusable sub-trajectory. For each failed trajectory, we first use the procedure in appendix A to assess whether it contains reusable experience and, if so, extract an additional reusable sub-trajectory. Each resulting experience consists of the task prompt, actions, and observation history.

Feature Construction. We represent each task-experience pair (t, m) as a sparse binary lexical vector. We first define the text associated with the current task and the candidate experience. The task text consists of the task prompt, while the experience text includes the experience prompt together with the trajectory text associated with historical experience m, including thoughts, code, and truncated observations. We then extract unigram and bigram features separately from both the task text and the experience text. To capture pair-specific effects, we further construct explicit interaction features between task and experience lexical units. Specifically, we first apply inverse document frequency (IDF) filtering to retain informative task and experience tokens. We then form cross features between the selected task tokens and selected experience tokens, yielding sparse indicators of joint lexical patterns. The final representation concatenates the task lexical features, the experience lexical features, and these interaction features into a single sparse binary vector. Full details are provided in appendix B and we also give ablation study for the features in section 5.4.

Training objective. Let $\mathcal{D}_G \subseteq \mathcal{T} \times \mathcal{M}$ denote the set of sampled task-experience pairs. For each $(t, m) \in \mathcal{D}_G$, let x_{tm} be the corresponding representation and $G_t(m) \in \{-1, 0, 1\}$ the observed gain. Although the two heads f_+ and f_- can be instantiated with any probabilistic classifier, we deliberately use simple linear models (Fan et al., 2008). Since x_{tm} depends only on text and not on the observed gain $G_t(m)$, it is computed once and reused across both heads and across the training process. We decompose the ternary gain label into two binary targets,

$$y_{tm}^+ := \mathbb{1}\left[G_t(m) = 1\right], \qquad y_{tm}^- := \mathbb{1}\left[G_t(m) = -1\right], \tag{8}$$

and train two heads, f_+ and f_-, independently with binary cross-entropy as follows

$$\mathcal{L}_+(f_+) = -\frac{1}{|\mathcal{D}_G|} \sum_{(t,m)\in\mathcal{D}_G} \left[y_{tm}^+ \log f_+(x_{tm}) + (1 - y_{tm}^+) \log(1 - f_+(x_{tm})) \right], \tag{9}$$

$$\mathcal{L}_-(f_-) = -\frac{1}{|\mathcal{D}_G|} \sum_{(t,m)\in\mathcal{D}_G} \left[y_{tm}^- \log f_-(x_{tm}) + (1 - y_{tm}^-) \log(1 - f_-(x_{tm})) \right]. \tag{10}$$

The resulting score $\hat{\tau}_f(t, m)$ measures the relative effect of using experience m for task t.

5 EXPERIMENTS

In section 5.2, we analyze the failure mode of similarity-based experience retrieval and show that historical experience usage should be treated as a decision problem. In section 5.3, we show that AgentEAES improves performance on AppWorld with the stronger Qwen-3.7-Max backbone, covering both the `test-normal` and `test-challenge` splits, and remains effective on WorkBench with the smaller and earlier Qwen3-30B-A3B backbone. In section 5.4, we verify the contribution of the proposed feature design through ablation experiments. In section 5.5, we analyze the inference-time cost of AgentEAES.

5.1 Experimental Setup

Benchmarks and metrics. We evaluate on AppWorld (Trivedi et al., 2024), an interactive coding benchmark where an agent solves multi-step tasks by calling application APIs, and WorkBench (Styles et al., 2024), which evaluates agents on realistic workplace tasks. For AppWorld, we report results on the official `test-normal` split with 168 tasks and the `test-challenge` split with 417 tasks. We use two metrics: task goal completion (TGC), which is the binary task success rate, and scenario goal completion (SGC), which is the fraction of scenarios for which all task variants succeed. WorkBench contains 690 tasks from six domains. We use a $50/50$ train/test split within each domain with seed 42, and we only inject experiences drawn from the same domain. For WorkBench, we report the official state-based accuracy. Since WorkBench has no scenario structure, there is no SGC counterpart.

Backbones and candidate pools. We evaluate AgentEAES under two different backbone settings to study its behavior across models with different capability levels. AppWorld is evaluated with the strong Qwen-3.7-Max backbone, while WorkBench is evaluated with the smaller and earlier Qwen3-30B-A3B backbone. This setup allows us to assess whether effect-aware experience selection remains useful across both stronger and weaker backbones on different benchmarks. Within each benchmark, all methods share the same agent scaffold and differ only in how experiences are selected. Candidate experiences are constructed only from trajectories of an experience-free agent run on training tasks. Evaluation tasks are never used as sources of experience and only serve as recipients of experience utilization. For AppWorld, candidates are sampled from the full set of constructed experiences, as described in appendix A.4. For WorkBench, we build domain-specific candidate pools, as described in appendix A.5.

Baselines and protocol. All methods share the same candidate pool and differ only in how the candidate experience is selected. We compare five methods. **No experience** uses the experience-free agent. **Random** samples a candidate uniformly. **Text-Sim** uses top-1 retrieval based on token-overlap Dice similarity (Dice, 1945). **Vector-Sim** uses top-1 retrieval based on BGE-M3 cosine similarity (Chen et al., 2024). **AgentEAES** selects the top-1 candidate according to the effect score $\hat{\tau}_f(t, m)$ and abstains when the maximum score is non-positive. Each AppWorld task and each domain-specific WorkBench task is run for three independent trials. We report mean $\pm$ standard deviation. Statistical significance against the no-experience baseline is assessed using an exact paired McNemar test (Fagerland et al., 2017) and a paired bootstrap over tasks.

5.2 Failure Mode of Similarity-Based Experience Utilization

We study this failure mode on the two benchmarks described in section 5.1. For each benchmark, we correlate task-experience embedding similarity with the observed gain defined by equation 1. We report both Pearson r and Spearman ρ, which capture linear and rank-based associations, respectively. In AppWorld, we compute gain on the dev split using both TGC and SGC metrics. The correlation is weak for both task-level gain and scenario-level gain. Figure 2 shows that the binned mean gain is small and non-monotone across the similarity range. The same pattern appears on WorkBench, but with clear domain dependence. Figure 3 shows that the correlation ranges from mildly positive in calendar and the multi-domain set to near-zero or negative in analytics, project management, and CRM. In these domains, high similarity does not imply that an experience will be helpful and can even be misleading. This domain dependence helps explain the downstream selector results in section 5.3. Retrieval is competitive only where similarity carries signal, whereas AgentEAES is strongest where similarity is weak or misleading. Appendix F also provides qualitative examples from both benchmarks.

5.3 Performance Comparison

AppWorld (Qwen-3.7-Max backbone). Table 1 reports the end-to-end comparison. AgentEAES substantially improves over the no-experience agent on both splits, raising TGC by $+10.1$ points on `test-normal` and by $+13.3$ points on `test-challenge`. These gains are not explained by adding context alone: on `test-normal`, random injection is statistically indistinguishable from no injection, whereas all content-aware selectors improve significantly. On the harder split,

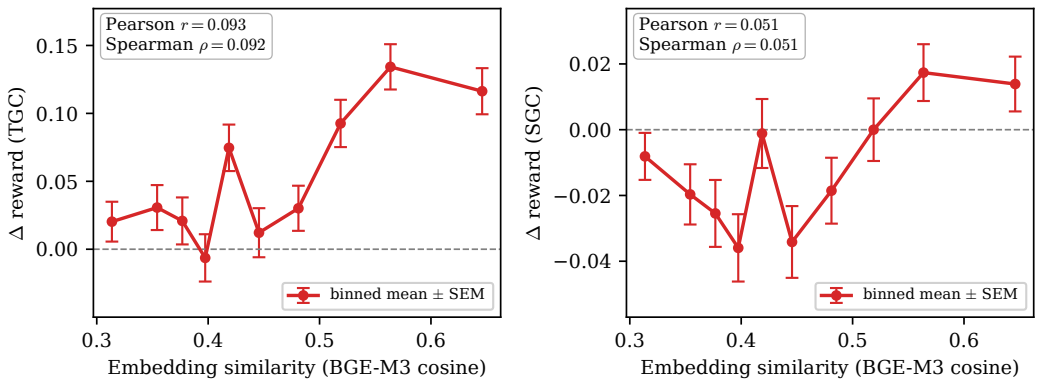

Figure 2: The x-axis is BGE-M3 cosine similarity between the task prompt and the candidate experience, computed over all task-candidate pairs from the dev split. Each point shows the mean gain within an equal-count similarity bin, with ± 1 SEM error bars. Left shows task-level gain measured by TGC, and right shows scenario-level gain measured by SGC. The correlation is weak in both cases (TGC $r = 0.093$, SGC $r = 0.051$).

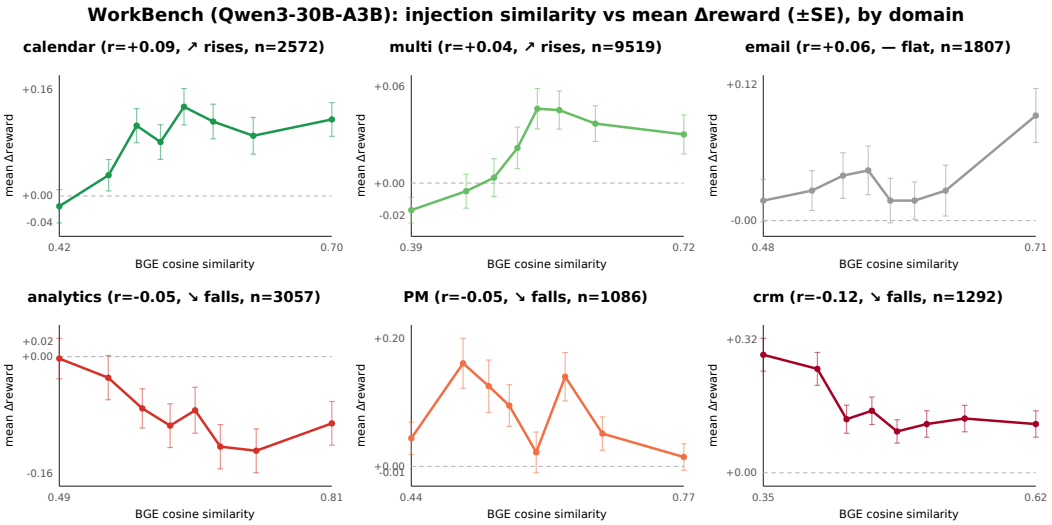

Figure 3: On WorkBench, the similarity-gain relationship is domain-dependent. Each panel shows one domain. Each point shows the mean gain within an equal-count similarity bin, with ± 1 SEM error bars. Correlation is mildly positive in calendar and multi-domain, near zero in email, and negative in analytics, project management, and CRM.

AgentEAES significantly outperforms random, text-similarity, and vector-similarity selection. It also reduces the average number of interaction rounds, though, as §5.5 shows, this does not translate into lower token cost.

WorkBench results (Qwen3-30B-A3B backbone). Table 2 reports the per-domain comparison on WorkBench. AgentEAES yields large gains on several domains, improving CRM from 3.3 to 44.2, project management from 25.2 to 55.0, and analytics from 32.2 to 58.3. It is the strongest method overall, leading in four of six domains and achieving the highest average accuracy at 48.3, compared with 39.0 for text similarity, 38.3 for vector similarity, and 31.5 for random selection. The remaining two domains are also informative. On email, all methods cluster closely, suggesting that performance is limited more by candidate-pool coverage than by selection. On the multi-domain set, vector similarity remains competitive, consistent with the stronger similarity signal shown in

Figure 3. Overall, these results show that effect-aware selection is most beneficial where similarity is a poor proxy for utility.

Table 1: Performance comparison on AppWorld (Qwen-3.7-Max backbone). Both TGC and SGC are reported as mean ± std over three independent trials.

	AppWorld `test-normal`		AppWorld `test-challenge`	
Method	TGC	SGC	TGC	SGC
No experience	64.3±3.0	44.6±1.8	61.1±1.4	35.7±1.5
Random	65.7±2.7	47.6±2.7	68.8±1.6	47.5±3.3
Text-sim (Dice)	69.4±1.2	52.4±5.7	70.6±4.1	51.1±9.7
Vector-sim (BGE-M3)	71.4±0.0	**55.9**±3.7	68.4±2.8	50.8±6.3
AgentEAES (ours)	**74.4**±3.1	54.2±3.7	**74.3**±0.9	**53.3**±2.5

Table 2: Per-domain WorkBench comparison on the Qwen3-30B-A3B backbone, reported as state-based accuracy (%), mean ± std over three independent trials. WorkBench has no SGC analogue. The average is over all six domains.

Method	Email (45)	Calendar (55)	Analytics (60)	CRM (40)	Proj. Mgmt. (40)	Multi (105)	Avg.
No experience	43.6±2.9	20.8±1.9	32.2±2.5	3.3±1.4	25.2±2.9	35.4±3.8	26.8
Random	47.4±4.6	20.1±1.4	39.6±2.1	17.5±2.5	32.5±4.3	31.9±3.9	31.5
Text-sim (Dice)	**52.6**±1.3	34.4±2.5	38.8±1.4	24.2±1.4	41.7±1.4	42.0±1.5	39.0
Vector-sim (BGE-M3)	49.6±1.3	35.6±6.5	43.9±5.1	17.5±5.0	40.0±0.0	**43.3**±1.7	38.3
AgentEAES (ours)	48.9±0.0	**40.9**±3.9	**58.3**±2.9	**44.2**±3.8	**55.0**±5.0	42.4±0.2	**48.3**

5.4 Feature Ablation Study

We next analyze which components of the task-experience representation drive the effect estimate defined in section 4. We evaluate feature subsets using held-out help-versus-harm GAUC on the dev split, where each task-wise AUC measures how well the model ranks helpful pairs above harmful pairs, and GAUC is the average of these task-wise AUC values. Table 3 reports the results.

Three findings stand out. First, task-only features yield a GAUC of 0.500, since they are constant within a task and therefore cannot rank candidates. Second, candidate-intrinsic features (0.613) outperform cross features (0.597) on their own, suggesting that candidate quality is a stronger signal than lexical matching. Third, combining all features yields the best performance at 0.634, compared to 0.629 without cross features. This shows that interaction features provide additional signal for task-specific ranking beyond generic candidate quality.

Table 3: Feature ablation on the task-experience representation. Candidate-only features are unigram and bigram lexical features from the candidate experience text; task-only features are lexical features from the task prompt; cross features are sparse lexical interactions between selected task and candidate-experience tokens.

Feature subset	Dev GAUC
All features (8 families)	**0.634**
Candidate-only features	0.613
Cross features (task×candidate)	0.597
All features w/o cross features	0.629
Task-only features	0.500

5.5 Cost Analysis

Section 5.3 showed that selective experience reduces interaction rounds on AppWorld. However, experience utilization also lengthens the prompt, so fewer rounds do not necessarily imply lower

overall cost. We therefore report both interaction efficiency (rounds) and token usage in Tables 4 and 5.

On AppWorld, all content-aware selectors reduce the number of rounds by roughly one quarter to one third. For example, on `test-normal`, AgentEAES reduces rounds from 17.2 to 12.4, and on `test-challenge`, from 19.7 to 15.1. However, because the selected experience remains in context across subsequent turns, it more than doubles the tokens per round. As a result, total tokens per task increase rather than decrease, by 69% for AgentEAES on `test-normal` and 62% on `test-challenge`. Thus, the main efficiency benefit of AgentEAES on AppWorld is fewer rounds and lower interaction latency rather than lower token cost.

On WorkBench, the pattern is different. Tasks are short and usually complete in about two rounds, leaving little room to shorten the interaction. Accordingly, experience utilization does not materially reduce rounds, and total token usage increases only modestly, by 11%-14% for the similarity-based and random selectors and by 21% for AgentEAES. In return, AgentEAES delivers the strongest accuracy gains. Overall, the cost profile differs by benchmark. On longer tasks, experience utilization trades higher token usage for fewer interaction rounds, whereas on shorter tasks it mainly adds modest overhead.

Table 4: Cost analysis on AppWorld (Qwen-3.7-Max). Rounds/task is the mean interaction steps; tokens/round is total tokens divided by total rounds. Percentages are relative to the experience-free agent. Fewer rounds coincide with higher per-round and total token cost.

Method	Rounds/task	Tokens/round	Tokens/task
AppWorld `test-normal`			
No experience	17.2	6,164	106,091
Random	12.3 (−28%)	12,972 (+110%)	159,831 (+51%)
Text-sim (Dice)	10.8 (−37%)	14,899 (+142%)	161,404 (+52%)
Vector-sim (BGE-M3)	11.5 (−33%)	16,453 (+167%)	189,751 (+79%)
AgentEAES (ours)	12.4 (−28%)	14,513 (+135%)	179,297 (+69%)
AppWorld `test-challenge`			
No experience	19.7	7,858	154,547
Random	15.3 (−22%)	15,392 (+96%)	235,455 (+52%)
Text-sim (Dice)	14.5 (−26%)	16,451 (+109%)	238,517 (+54%)
Vector-sim (BGE-M3)	14.6 (−26%)	17,262 (+120%)	251,377 (+63%)
AgentEAES (ours)	15.1 (−23%)	16,536 (+110%)	250,294 (+62%)

Table 5: Cost analysis on WorkBench (Qwen3-30B-A3B), aggregated over all six domains (three trials each). Columns and percentages are as in Table 4.

Method	Rounds/task	Tokens/round	Tokens/task
No experience	2.0	13,395	26,643
Random	2.0 (+0%)	15,235 (+14%)	30,351 (+14%)
Text-sim (Dice)	2.0 (+3%)	14,706 (+10%)	30,002 (+13%)
Vector-sim (BGE-M3)	2.0 (+3%)	14,530 (+8%)	29,686 (+11%)
AgentEAES (ours)	2.0 (+2%)	15,980 (+19%)	32,286 (+21%)

Training cost. The analysis above reflect inference-time cost only. AgentEAES additionally incurs a one-time cost to build the effect model training set. This requires a sampling-and-labeling step in which each sampled task-experience pair from the training split is assigned a label via an additional LLM rollout. The resulting number of labels is therefore the number of sampled pairs, which is 21,585 on WorkBench and 9,090 on AppWorld. At single-rollout cost, one label averages about 32K tokens on WorkBench and 0.18M tokens on AppWorld, for total labeling costs of roughly 0.7×10^9 and 1.6×10^9 tokens, respectively. Because this expense is incurred only once, it is amortized across all later inference runs and should be distinguished from the per-task inference costs reported in tables 4 and 5.

6 CONCLUSION

We study how LLM agents can learn to use experience beyond similarity-based retrieval. Our experiments show that retrieval alone can cause negative transfer, even when the retrieved experience appears highly relevant. This suggests that experience usage should be framed not only as a retrieval problem, but also as a decision problem.

To address this, we propose AgentEAES, a framework that learns experience-usage decisions from sampled interventions. Across two agentic benchmarks, AgentEAES consistently outperforms retrieval-based baselines and improves overall task success rates.

At the same time, our study has several limitations. First, in rapidly changing environments, an effect model trained offline may become stale. Extending AgentEAES with exploration mechanisms to collect online feedback and update its decisions based on newly logged interaction traces is an important direction for future work. Second, when the underlying backbone is already very strong, the available headroom may be limited, which can reduce the absolute gains from experience selection. Even in such settings, however, AgentEAES can still serve as a model-based harness for improving the baseline performance of agent systems.

Overall, our results highlight the value of moving beyond retrieval toward intervention-aware, decision-oriented experience usage for LLM agents. We hope this work encourages future research on more adaptive, principled, and continually improving approaches to agent experience utilization.

REPRODUCIBILITY STATEMENT

All benchmarks, data splits, and evaluation metrics are either public or fully specified in the paper. Section 5.1 describes AppWorld's official `test-normal` and `test-challenge` splits, as well as the per-domain WorkBench $50/50$ split constructed with random seed 42. Pool construction, acceptance criteria, feature design, and model-fitting configurations are detailed in appendices A-B, together with Tables 7 and 10. Prompt templates and runtime scaffolding are documented in appendix E. Code, candidate pool-construction scripts, and the complete prompt suite are provided in the anonymous repository linked in the abstract.

LLM USAGE STATEMENT

LLMs are used in three roles in this work. First, they serve as the task-solving backbones, specifically Qwen-3.7-Max and Qwen3-30B-A3B. Second, they are used in an offline processing pipeline for outcome extraction, intention segmentation, hindsight relabeling, and second-pass judgment, as described in appendix A. Third, they were used for limited prose polishing during manuscript preparation. All reported metrics are computed programmatically from environment states or unit tests. No reported experimental result or empirical claim depends on an LLM judgment.

REFERENCES

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, et al. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. BGE M3-Embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216*, 2024.

Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3): 297–302, 1945.

Liang Ding. Agenther: Hindsight experience replay for llm agent trajectory relabeling. *arXiv preprint arXiv:2603.21357*, 2026.

Morten Fagerland, Stian Lydersen, and Petter Laake. *Statistical analysis of contingency tables*. CRC press, 2017.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

Shengda Fan, Xin Cong, Zhong Zhang, Yuepeng Fu, Yesai Wu, Hao Wang, Xinyu Zhang, Enrui Hu, and Yankai Lin. Generalizing experience for language agents with hierarchical metaflows. *Advances in Neural Information Processing Systems*, 38:64103–64132, 2026.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. *arXiv preprint arXiv:2402.03610*, 2024.

Jinghao Luo, Yuchen Tian, Chuxue Cao, Ziyang Luo, Hongzhan Lin, Kaixin Li, Chuyi Kong, Ruichao Yang, and Jing Ma. From storage to experience: A survey on the evolution of llm agent memory mechanisms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 41622–41652, 2026.

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

Olly Styles and Sam Miller. Workbench revisited: Workplace agents two years on. *arXiv preprint arXiv:2606.13715*, 2026.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. Workbench: a benchmark dataset for agents in a realistic workplace setting. *arXiv preprint arXiv:2405.00823*, 2024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Yuxia Wang, Minghan Wang, Muhammad Arslan Manzoor, Fei Liu, Georgi Nenkov Georgiev, Rocktim Jyoti Das, and Preslav Nakov. Factuality of large language models: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 19519–19529, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

APPENDIX

A EXPERIENCE POOL CONSTRUCTION AND HINDSIGHT RELABELING

We construct candidate experiences from the agent's recorded interactions. The outcome-extraction and hindsight goal-relabeling stages follow AgentHER (Ding, 2026); we add a deterministic failure filter, intention-boundary prefixes from both successful and failed rollouts, and a confidence-gated acceptance rule (Algorithm 2). The resulting experiences serve as inference-time context. Figure 4 summarizes the pipeline.

A.1 SOURCE TRAJECTORIES AND FOUR POOLS

Let $z = (q, \xi, r)$ denote a source record with original instruction q, stored trajectory $\xi = (s_1, \ldots, s_n)$, and source-task reward r. Each step contains the recorded thought, action, arguments, and observation. We collect source trajectories by running the agent on training tasks without injected experiences and exclude empty execution records. Success is determined by the original task reward ($r \geq 1$). Per-pool sizes for AppWorld appear in Appendix A.4.

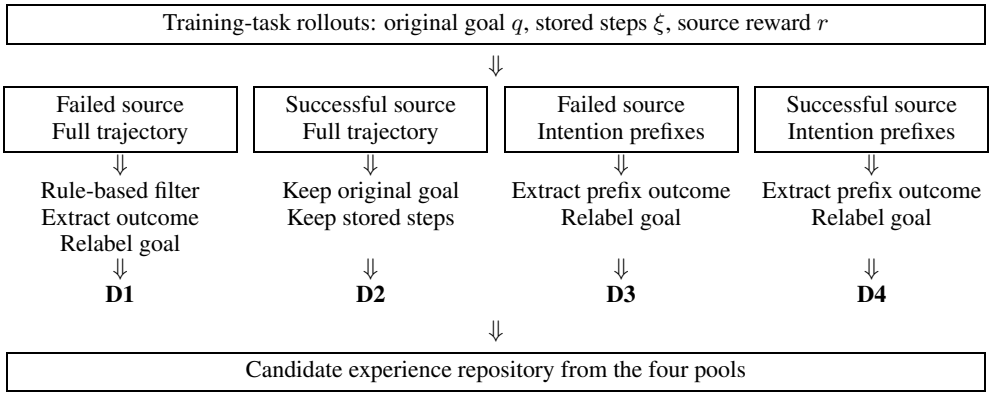

Figure 4: Four-pool experience construction. Relabeling changes the instruction, not the actions or observations. D3/D4 prefixes inherit their source rollout's outcome label; they are not independently verified successful executions. Confidence-gated relabeling is expanded in Algorithm 2; per-pool sizes are given in Appendix A.4.

D1 contains failed full trajectories paired with hindsight goals; D2 retains successful full trajectories with their original goals. D3 and D4 contain hindsight-relabeled prefixes of failed and successful source rollouts, respectively. "Failed" and "successful" refer to the complete source rollout. A retained prefix may contain errors, and acceptance of a hindsight goal does not guarantee that the original task succeeds.

A.2 FAILURE FILTERING AND INTENTION-BOUNDARY PREFIXES

Filtering full failures. D1 uses a deterministic rule-based detector to ensure reproducibility and avoid additional LLM cost. The detector matches category-specific keywords in aggregated trajectory text, selects the category with the most hits (ties follow a fixed order), and defaults to `wrong_result` when no category matches. A record is recoverable if at least one stripped observation exceeds 20 characters and the category is neither `tool_error` nor `early_termination`. The severity weight is $w = \min(1, 0.3 + 0.1h)$ with threshold $w \geq 0.3$, where h is the largest category hit count. The filter operates at the trajectory level to select candidates for hindsight relabeling.

Extracting prefixes. We send the full stored trajectory to an LLM to jointly summarize achieved outcomes and identify consecutive intention segments, each with one-indexed inclusive `start_step`, `end_step`, and `sub_goal`. From all but the final segment, we use the end index

Algorithm 1 Four-pool construction from recorded source rollouts

Require: Source records Z; outcome extractor E; segmenter S; relabeler H (Algorithm 2)
Initialize record lists $D_1, D_2, D_3, D_4 \leftarrow [\,]$
for each nonempty $z = (q, \xi, r) \in Z$ **do**
  $n \leftarrow |\xi|$
  if $r \geq 1$ **then**
    Append Serialize(q, ξ, z) to D_2
  else if Recoverable(z) and $w(z) \geq 0.3$ **then**
    $(q', c') \leftarrow H(z, E(z))$
    if $q' \neq \perp$ **then**
      Append Serialize(q', ξ, z) to D_1
    end if
  end if
  $B \leftarrow$ non-final segment end indices returned by $S(z)$
  for each $k \in B$ with $n \geq 3$ and $2 \leq k < n$ **do**
    $z_k \leftarrow$ prefix $\xi_{1:k}$, empty final answer, source metadata and cut index k
    $(q'_k, c'_k) \leftarrow H(z_k, E(z_k))$
    if $q'_k \neq \perp$ **then**
      Append Serialize$(q'_k, \xi_{1:k}, z_k)$ to D_4 if $r \geq 1$, else D_3
    end if
  end for
end for
return D_1, D_2, D_3, D_4

k as a cut if $n \geq 3$ and $2 \leq k < n$. The experience contains the cumulative prefix $s_1, \ldots, s_k$; we clear its final-answer field and extend its ID by _sub$k$.

Each prefix passes through outcome extraction and relabeling independently, without the D1 failure detector. We impose no cap on the number of valid boundaries. Cut points are validated for range; the segmentation properties requested in the prompt serve as soft guidance. An intention boundary marks a proposed sub-goal transition.

A.3 Outcome Extraction, Relabeling, and Acceptance

The construction pipeline uses a GPT-5.4-class LLM for outcome extraction, intention segmentation, hindsight relabeling, and second judgment. This model is separate from the Qwen-3.7-Max task-solving agent. D2 records retain their original goals and require none of these calls.

The four LLM stages adapt AgentHER-style templates (Ding, 2026). Outcome extraction requests evidence-supported `actual_achievements`, `key_observations`, and `limitations`. Relabeling receives those fields plus the original prompt as a style reference and the trajectory length, and requests a new instruction that does not mention failure or hindsight. The second judgment assesses whether the trace satisfies the proposed requirement. We never rewrite stored actions and observations. System instructions appear in Appendix E.1; full templates are in the code release.

Algorithm 2 gives the acceptance rule; Table 7 summarizes the defaults. The first attempt uses temperature 0.3, subsequent attempts use 0.7. A valid proposal with confidence $\geq \theta = 0.5$ triggers a second-judgment call at temperature 0 using the same model. Acceptance requires the second confidence $\geq \theta$ and stores the average of both confidences. After at most three attempts, we accept a best valid first-judge proposal at confidence $\geq 0.8\theta = 0.4$.

Malformed structured responses consume an attempt; verifier failures allow later attempts. API errors propagate to the calling pipeline. All checks operate on recorded evidence without environment replay.

A.4 Serialization and Pool Composition

The serializer stores the hindsight instruction, original instruction, source ID, structured steps, provenance, confidence, and prefix metadata. The SFT-style `chosen` conversation contains a user goal and a flattened execution record with `Step`, `Action`, and `Result` fields. For D2, the goal is the

Algorithm 2 Confidence-gated hindsight relabeling of a full record or prefix

Require: Record z, extracted outcome o, threshold $\theta = 0.5$, attempts $K = 3$
$b \leftarrow \perp$; $c_b \leftarrow -\infty$
for $j = 1, \ldots, K$ **do**
  $T \leftarrow 0.3$ if $j = 1$, else 0.7
  $(q', v, c) \leftarrow \mathrm{Relabel}(o, \mathrm{goal}(z), \mathrm{numSteps}(z); T)$
  if v **then**
    if $c > c_b$ **then**
      $(b, c_b) \leftarrow (q', c)$
    end if
    if $c \geq \theta$ **then**
      $(v_2, c_2) \leftarrow \mathrm{Verify}(q', z; T = 0)$
      if $c_2 \geq \theta$ **then**
        return $(q',\ (c + c_2)/2)$
      end if
    end if
  end if
end for
if $b \neq \perp$ and $c_b \geq 0.8\theta$ **then**
  return (b, c_b)
end if
return $(\perp, 0)$

original instruction. The severity weight is stored as a record field but not used as a sample weight or for fine-tuning.

Table 6 reports the AppWorld per-pool sizes from a single trial, analogous to the per-domain WorkBench pools in Table 8. The counterfactual sweep draws candidates by sampling from these pools. Feature extraction, injection rendering, and case attribution all use these pools. Sweep sampling is distinct from the task-experience pair sampling used to train the effect model.

Table 6: AppWorld per-pool experience sizes from a single trial (records per pool D1–D4), the analogue of the per-domain WorkBench pools in Table 8.

AppWorld pool	D1	D2	D3	D4	Total
Records	11	58	131	241	441

Table 7: Construction defaults for AppWorld pool construction. These settings describe record processing, not environment verification of a relabeled goal.

Parameter	Value
D1 observation-length requirement	At least one stripped observation > 20 characters
D1 minimum severity weight	0.3
Prefix length and cut range	$n \geq 3$; $2 \leq k < n$; no cut-count cap
Relabel attempts; temperatures	3; $0.3, 0.7, 0.7$
Extraction / segmentation / verification temperature	0
First / second confidence threshold	0.5 / 0.5
Best valid single-judge fallback	Confidence ≥ 0.4

A.5 WorkBench Pools and Within-Domain Injection

WorkBench uses the same four-pool construction (Algorithms 1 and 2) with two differences. We collect source trajectories from the experience-free ReAct agent on each domain's training split, and success uses WorkBench's state-based reward (Appendix E.5). We perform injection per domain: candidates are drawn from pools built on the same domain's training split, excluding the target's own source trajectory. The per-domain $50/50$ train/test split (seed 42) covers email $45/45$, calendar $55/55$, analytics $60/60$, project management $40/40$, CRM $40/40$, and multi-domain $105/105$,

totaling $345/345$ over 690 tasks. Table 8 lists the per-domain pool sizes. On the smaller Qwen3-30B-A3B backbone, the failed-source pools (D1, D3) are comparatively large. Effective pool sizes are slightly smaller after last-occurrence ID resolution (e.g., analytics resolves to 52 candidates).

Table 8: Per-domain WorkBench experience-pool sizes (records per pool D1–D4), built on the Qwen3-30B-A3B backbone's training-split trajectories and injected only within the same domain.

Domain	D1	D2	D3	D4	Total
Email	29	10	1	3	43
Calendar	38	8	4	0	50
Analytics	24	35	7	5	71
Project management	9	12	6	1	28
Customer relationship management	28	3	3	0	34
Multi-domain	73	23	5	5	106

B FEATURE DESIGN

We represent a task-experience pair using eight families of lexical indicator features (Table 9). The input matrix is sparse: each pair activates a small subset of a large vocabulary (519,762 dimensions for AppWorld).

B.1 INPUT TEXT AND TOKENIZATION

Let a_t be the target's original instruction (*anchor prompt*), p_m the experience's hindsight instruction (original instruction for D2), and c_m its trajectory text. The trajectory text c_m concatenates each step's nonempty thought, Python code, and first 300 observation characters. We impose no overall trajectory cap. This feature-only cap does not limit the agent's injected context (Appendix E.4, Table 15).

We lowercase spans matched by `[a-zA-Z0-9]+` and remove fixed stopwords and tokens of length at most one. Punctuation and underscores separate API-name components. Let $U(v)$ be the remaining unigram set and $B(v)$ the adjacent-bigram set of the filtered sequence. For example, "songs from my album library" produces $(\texttt{songs}, \texttt{album})$ and $(\texttt{album}, \texttt{library})$. Each distinct term activates a feature only once.

Table 9: The eight sparse feature families. Each observed unigram, bigram, or ordered cross pair defines a separate binary column. A_t, P_m, C_m are the filtered top-IDF token sets defined below; cross terms are not restricted to identical words.

Group	Family	Active terms for (t, m)
Candidate	`cp_uni`	$U(p_m)$
	`cp_bi`	$B(p_m)$
	`ct_uni`	$U(c_m)$
	`ct_bi`	$B(c_m)$
Task	`ap_uni`	$U(a_t)$
	`ap_bi`	$B(a_t)$
Interaction	`x_ap_cp`	$A_t \times P_m$, at most 15×20 terms
	`x_ap_ct`	$A_t \times C_m$, at most 15×30 terms

Fixed stopwords. The implementation uses the following set, rather than a library-dependent language stoplist:

```
a, an, the, is, are, was, were, am, be, been, being, i, you,
he, she, it, we, they, my, your, his, her, its, our, their,
me, him, us, them, to, of, in, on, at, for, with, and, or,
but, that, this, these, those, have, has, had, do, does,
did, not, no, so, if, then, than, there, here, from, by, as,
just, please, would, could, can, will, want, like, get, got.
```

Negations such as `not` and `no` are removed. The representation is a lexical interaction model.

B.2 BOUNDED CROSS-TASK-EXPERIENCE INTERACTIONS

Cross terms allow a target word to interact with a different experience word or API-name component. To bound their number, we select distinct tokens by inverse document frequency:

$$\mathrm{idf}(u) = 1 + \log \frac{N+1}{\mathrm{df}(u)+1}. \tag{11}$$

The corpus contains one prompt per represented anchor and per represented candidate; N is the total number of such prompts. Trajectory text is excluded; unseen trajectory tokens receive $\mathrm{df} = 0$.

Before top-token selection, we discard purely numeric tokens, hexadecimal strings of at least eight characters, and alphanumeric strings of at least ten characters containing a digit, preventing rare

entity IDs from consuming the token budget. Unigram and bigram families retain such tokens. Remaining distinct tokens are sorted by decreasing IDF (lexicographic order breaking ties). We keep up to 15 anchor tokens (A_t), 20 candidate-prompt tokens (P_m), and 30 candidate-trajectory tokens (C_m). The interaction families are the full Cartesian products

$$\mathcal{X}_{tm}^{\text{prompt}} = A_t \times P_m, \qquad \mathcal{X}_{tm}^{\text{trace}} = A_t \times C_m. \tag{12}$$

An ordered pair is encoded as two tokens separated by a NUL delimiter; bigrams use the same delimiter within their separate family namespace. IDF only selects tokens: all final feature values remain 0 or 1.

B.3 Sparse Encoding, Fitting, and Inference

For family j with vocabulary V_j, define

$$[x_{tm}^{(j)}]_u = \mathbf{1}[u \text{ occurs in family } j \text{ for } (t,m)], \quad u \in V_j, \qquad x_{tm} = \bigoplus_{j=1}^{9} x_{tm}^{(j)}. \tag{13}$$

Term indices define a binary float32 CSR matrix without counts, TF-IDF weights, or normalization. Candidate-intrinsic features capture task-independent preferences; cross terms permit pair-specific interactions. Task-only features cannot rank candidates alone but can shift both heads' operating points.

Table 10: Configuration of the AppWorld model used in the main end-to-end comparison. Both logistic heads use the same feature map.

Item	Value
Training / dev task–experience pairs	9,090 / 50,662
Training / dev anchors	90 / 57
Training candidates per anchor	101
Combined design matrix	59,752 × 1,028,477
Nonzero entries	61,447,890
Positive / negative labels	$\Delta > 0$ / $\Delta < 0$
Regularization; inverse strength	L2; $C = 0.01$
Solver; maximum iterations	`liblinear`; 2,000
Intercept; class weights; fitting seed	Enabled; none; 42
Inference score and rule	$s_f = f_+ - f_-$; top-1 if positive

Data boundaries. Vocabulary and IDF are fit on unlabeled train and dev texts jointly; only the 9,090 training labels enter the classifier loss (Section 4). Inference freezes all preprocessing objects and ignores unknown terms. Candidate prompts fall back to original instructions when the hindsight field is empty.

Because no dev label ever enters the loss, this preprocessing is transductive rather than a train-only feature fit. To confirm that the reported rankings do not depend on dev-text access, we refit the feature map on train texts alone, freeze it, and transform dev with it, discarding out-of-vocabulary terms exactly as deployment does. The vocabulary shrinks from 1,028,477 to 1,017,113 columns (1.1% drop), yet help-versus-harm GAUC is 0.634 under both settings (transductive versus train-only, on the same 27 tasks and 23,999 pairs). The feature map therefore does not rely on dev text.

C Effect of Sample Size on Scorer Quality

Section 5.5 reports that training the effect model requires 9,090 labeled task-experience pairs on AppWorld, a substantial one-time cost. We analyze whether this sample size is necessary or whether a smaller sample would suffice. We use *uniform random sampling*, which draws K candidates per anchor uniformly from the full sweep. We train the effect model and report help-versus-harm GAUC on the full dev split (889 candidates × 57 anchors, 27 informative tasks).

Table 11 shows that scorer quality improves with sample size but exhibits diminishing returns. Uniform sampling achieves GAUC 0.634 at $K = 101$ (9,090 pairs), compared to the full sweep's 0.660

at 79,120 pairs. Increasing to $K = 400$ (36,000 pairs) yields 0.652, capturing most of the full-sweep performance. The results show that a sample of ∼9,000 pairs provides reasonable performance and reduces the one-time labeling cost by roughly $9\times$.

Table 11: Effect of sample size on scorer quality with uniform random sampling (seed 42). The full sweep uses all available task–experience pairs. Evaluated on the full dev set (889 candidates, 57 anchors, 27 informative tasks).

Sample size	**Training pairs**	**Dev GAUC (hvh)**
$K = 30$	2,700	0.602
$K = 60$	5,400	0.624
$K = 101$	9,090	0.634
$K = 200$	18,000	0.643
$K = 400$	36,000	0.652
Full sweep	79,120	0.660

D TRIAL-LEVEL RESULTS

All main-text tables report results from three independent trials (t1–t3). Tables 12–14 provide the complete trial-level breakdown for reproducibility. Several entries show ± 0.0 standard deviation. These reflect deterministic behavior: BGE-M3 retrieval produces identical rankings across trials, and on certain domains the agent's temperature is low enough that task-level outcomes are identical across all three trials.

Table 12: Trial-level raw results for AppWorld test-normal (168 tasks). TGC and SGC are percentages. The ± 0.0 for Vector-sim TGC is real: BGE-M3 retrieval is deterministic and the agent produced identical task-level outcomes across all three trials.

Method	**Trial**	**TGC (%)**	**SGC (%)**
No experience	t1	61.3	42.9
	t2	67.3	44.6
	t3	64.3	46.4
	mean ± std	*64.3 ± 3.0*	*44.6 ± 1.8*
Random	t1	63.1	44.6
	t2	68.5	48.2
	t3	65.5	50.0
	mean ± std	*65.7 ± 2.7*	*47.6 ± 2.7*
Text-sim (Dice)	t1	68.5	48.2
	t2	70.8	50.0
	t3	69.0	58.9
	mean ± std	*69.4 ± 1.2*	*52.4 ± 5.7*
Vector-sim (BGE-M3)	t1	71.4	58.9
	t2	71.4	57.1
	t3	71.4	51.8
	mean ± std	*71.4 ± 0.0*	*55.9 ± 3.7*
AgentEAES (ours)	t1	78.0	57.1
	t2	72.6	50.0
	t3	72.6	55.4
	mean ± std	*74.4 ± 3.1*	*54.2 ± 3.7*

Table 13: Trial-level raw results for AppWorld test-challenge (417 tasks). All main-text tables report trials t1–t3. TGC and SGC are percentages.

Method	Trial	TGC (%)	SGC (%)
No experience	t1	61.9	37.4
	t2	59.5	35.2
	t3	61.9	34.5
	mean ± std	*61.1 ± 1.4*	*35.7 ± 1.5*
Random	t1	70.5	51.1
	t2	68.6	44.6
	t3	67.4	46.8
	mean ± std	*68.8 ± 1.6*	*47.5 ± 3.3*
Text-sim (Dice)	t1	71.2	51.8
	t2	74.3	60.4
	t3	66.2	41.0
	mean ± std	*70.6 ± 4.1*	*51.1 ± 9.7*
Vector-sim (BGE-M3)	t1	65.2	43.9
	t2	70.0	52.5
	t3	70.0	56.1
	mean ± std	*68.4 ± 2.8*	*50.8 ± 6.3*
AgentEAES (ours)	t1	73.4	50.4
	t2	74.6	54.7
	t3	75.1	54.7
	mean ± std	*74.4 ± 0.9*	*53.3 ± 2.5*

Table 14: Trial-level raw results for WorkBench (Qwen3-30B-A3B backbone). All main-text tables report trials t1–t3. State-based accuracy (%). The ±0.0 for Vector-sim Proj. Mgmt. and AgentEAES Email are real: these arms produced identical task-level outcomes across all three trials.

Method	Trial	Email	Calendar	Analytics	CRM	Proj.M.	Multi
No experience	t1	40.9	18.9	30.0	2.5	28.2	37.9
	t2	43.2	20.8	31.7	2.5	25.0	37.2
	t3	46.7	22.6	35.0	5.0	22.5	31.1
	mean±std	*43.6±2.9*	*20.8±1.9*	*32.2±2.5*	*3.3±1.4*	*25.2±2.9*	*35.4±3.8*
Random	t1	51.1	20.8	40.0	15.0	27.5	31.7
	t2	42.2	18.5	37.3	17.5	35.0	35.9
	t3	48.9	21.1	41.4	20.0	35.0	28.2
	mean±std	*47.4±4.6*	*20.1±1.4*	*39.6±2.1*	*17.5±2.5*	*32.5±4.3*	*31.9±3.9*
Text-sim	t1	51.1	35.9	39.0	22.5	42.5	40.4
	t2	53.3	31.5	37.3	25.0	40.0	42.3
	t3	53.3	35.9	40.0	25.0	42.5	43.3
	mean±std	*52.6±1.3*	*34.4±2.5*	*38.8±1.4*	*24.2±1.4*	*41.7±1.4*	*42.0±1.5*
Vector-sim	t1	48.9	37.7	38.3	22.5	40.0	44.2
	t2	48.9	40.7	45.0	12.5	40.0	44.2
	t3	51.1	28.3	48.3	17.5	40.0	41.3
	mean±std	*49.6±1.3*	*35.6±6.5*	*43.9±5.1*	*17.5±5.0*	*40.0±0.0*	*43.3±1.7*
AgentEAES	t1	48.9	45.3	56.7	40.0	55.0	42.2
	t2	48.9	39.6	56.7	47.5	50.0	42.6
	t3	48.9	37.7	61.7	45.0	60.0	42.6
	mean±std	*48.9±0.0*	*40.9±3.9*	*58.3±2.9*	*44.2±3.8*	*55.0±5.0*	*42.4±0.2*

E PROMPT DESIGN

We separate the shared offline prompts that construct experiences from the benchmark-specific runtime prompts. The boxes below reproduce only the core system instructions. The complete template suite—user-side templates, structured-output schemas, and full benchmark scaffolds—is provided in the code release (`experiments/data_augment/prompts.py` and the benchmark runners). Braced variables denote runtime substitutions; marked loops stand for Jinja iterations. No benchmark credentials are included.

E.1 SHARED OFFLINE CONSTRUCTION AND RELABELING PROMPTS

The offline pipeline has four LLM stages: outcome extraction, joint outcome extraction with intention segmentation (prefix path), hindsight relabeling, and second-judge verification. Failure detection is rule-based. Full failures use outcome extraction, relabeling, and (when triggered) verification. Prefix construction segments the full trace and then applies outcome extraction to each retained prefix. D2 requires none of these calls. Every structured call appends a shared JSON-schema suffix. Stage-specific output fields are: outcome extraction (`actual_achievements`, `key_observations`, `limitations`); segmentation (the same plus `intention_segments` with `start_step`, `end_step`, `sub_goal`); relabeling (`trajectory_id`, `hindsight_prompt`, `is_valid_replay`, `rationale`, `confidence`); second judge (`is_valid`, `confidence`, `rejection_reason`). Confidences are in $[0, 1]$; segment indices are positive integers. The two stage-defining system instructions are Prompts SC4a and SC5a. Their user templates supply the original prompt, the flattened step loop, and the final answer.

Prompt SC4a: Hindsight goal relabeling — SYSTEM

```
You are a creative prompt engineer.  Given a summary of what an LLM
agent actually achieved during a task, you must write a NEW user prompt
that would make the agent's trajectory look like a PERFECT, SUCCESSFUL
execution.

Requirements for the hindsight prompt:
1.  It must be natural -- something a real user might actually ask
2.  It must be specific enough that the trajectory clearly satisfies it
3.  It must NOT reference the original failed prompt or mention
"hindsight"
4.  It should match the complexity/style of the original prompt
5.  All constraints in the new prompt must be satisfied by the
trajectory's outcomes

Also assess your confidence (0.0-1.0) that this relabeling produces a
genuinely useful training sample.  Low confidence if the trajectory is
too noisy or the achievements are trivial.
```

Prompt SC5a: Second-judge verification — SYSTEM

```
You are an independent trajectory verifier.  Given a proposed hindsight
prompt and an agent trajectory, decide whether the trajectory is a
SUCCESSFUL execution of that prompt.

Evaluation criteria:
1.  Every factual claim or constraint in the prompt is supported by
actual observations
2.  The trajectory reaches a conclusion that satisfies all of the
prompt's requirements
3.  No requirement in the prompt is contradicted by any observation

Be strict:  assign high confidence (≥ 0.5) only if ALL requirements are
clearly met.  If is_valid is False, briefly explain which requirement is
unmet.
```

The configuration uses a single LLM client (Appendix A) with default temperature 0. Relabeling overrides temperature as specified in Algorithm 2; segmentation, outcome extraction, and verification use 0.

E.2 AppWorld Runtime: Agent Prompt and Message Sequence

The system instruction (Prompt AW1) is a code-agent scaffold. The target instruction is obtained from `world.task.instruction`. The scaffold asks the agent to discover API descriptions through documentation APIs rather than embedding them in advance.

Prompt AW1: Code-agent system instruction

```
You are a code agent that helps users accomplish tasks by writing Python
code to interact with application APIs.

You have access to a set of application APIs through the `apis` module.
Each app's methods are accessible as
`apis.{app_name}.{method_name}(...)`.

You do NOT know the API surface in advance -- discover it by calling
these built-in doc APIs as code:
- `apis.api_docs.show_app_descriptions()` -- list all available apps and
what they do.
- `apis.api_docs.show_api_descriptions(app_name="...")` -- list all APIs
(with one-line descriptions) for a given app.
- `apis.api_docs.show_api_doc(app_name="...", api_name="...")` -- get
the full signature/params/response schema for one API.

Rules:
1.  Write ONLY executable Python code -- no markdown, no explanation
outside code.
2.  Use `apis.{app_name}.{method_name}(args)` to call any API. Look up
unfamiliar apps/APIs via apis.api_docs.* before calling them.
3.  You can use standard Python (loops, conditionals, variables) to
compose multi-step logic.
4.  Print results to see API responses:
`print(apis.spotify.search_songs("hello"))`.
5.  When you have completed the task (or cannot complete it), call
`apis.supervisor.complete_task()`, explaining why in a comment if
unsuccessful.
6.  Each code block you write will be executed, and you'll see the
output.  Then write the next code block.
```

A rollout starts with Prompt AW1 (plus optional experience) and the initial user message "Task: {instruction} — Write Python code to accomplish this task." At each iteration, we retain the assistant response in the history, pass executable code to `world.execute`, and append the execution feedback "Execution output: {output} — Continue with the next step, or call `apis.supervisor.complete_task()` if done." as a user message. The history accumulates without a sliding window for at most 30 iterations. The runner stops its loop when the extracted code contains `complete_task`, and then evaluates the environment state. These are execution rules, not additional instructions inferred from the demonstrations. The base prompt's completion rule does not spell out the answer argument needed by answer-bearing tasks, a distinction illustrated by the C2 case in Appendix F.2.

E.3 AppWorld Runtime: Experience-Augmented Context

The renderer appends the wrapper in Prompt AW4 to the system instruction (Appendix E.2) before the target rollout starts. The task user message is unchanged. The forced-injection sweep supplies one candidate; end-to-end selectors supply one or none. The experience remains in the accumulated system context across steps.

Prompt AW4: AppWorld experience wrapper appended to the system instruction

```
<relevant_experience>
Below are similar task experiences that may help you:
{memory_text}
</relevant_experience>
```

Template SR1: Shared runtime experience block substituted into memory_text

```
### Experience {i}
<meta>{meta_line}</meta>
Task: {hindsight_prompt}
Solution:
{stored_assistant_response}
```

The task field uses `hindsight_prompt`, falling back to `original_prompt` if empty. The solution is the last message in `chosen`, containing the full flattened trajectory with layout `Step k: thought`, `Action: name(JSON arguments)`, `Result: observation`, and a final answer when stored.

The metadata line begins with `provenance: value (outcome)` and conditionally adds `reward`, `steps`, `domain`, `tools`, and `excerpt`, separated by vertical bars. Reward is the source reward. The renderer labels `failure_relabel` and `failure_subtraj_relabel` as `repaired` (presentational only). The wrapper's word "similar" is fixed prompt wording independent of the selection rule.

E.4 AppWorld Runtime: Observation Budgets

Table 15: Distinct text budgets in the AppWorld pipeline. Character limits apply at the specified stage, not to all views of a trajectory.

Stage	Text supplied or retained
Online execution feedback	Full output returned by `world.execute`; no additional character truncation in the execution-feedback message.
Stored source step	First 3,000 observation characters. Later processing cannot recover the removed suffix.
Sparse lexical features	First 300 characters of each stored observation; thought/code retained.
Stored experience solution	The flattened solution retains the complete structured-step observations of the stored trajectory.
Final injection renderer	No additional solution cap in the reported configuration (`max_response_chars=0`).

All effective records' flattened solutions begin with the complete serialization of their stored steps. The 300-character feature cap does not limit the injection length.

E.5 WorkBench Runtime: Agent and Injection Prompts

WorkBench uses a ReAct scaffold. The system instruction (Prompt WB1) combines a date/time prefix, role preamble, tool catalogue, and JSON-action-format instructions. Each assistant turn emits a JSON action blob (`action`/`action_input`). The tool result becomes an `Observation` and the loop runs for at most 20 steps at temperature 0. All selection methods share this scaffold.

Prompt WB1: WorkBench ReAct system instruction (abbreviated)

```
Today's date is Thursday, 2023-11-30 and the current time is 00:00:00.
Remember the current date and time when completing tasks. Meetings must
not start before 9am or end after 6pm.

Respond to the human as helpfully and accurately as possible. You have
access to the following tools:

email.send_email: email.send_email(recipient: Optional[str] = None,
subject: Optional[str] = None, body: Optional[str] = None) -> str
[... one line per available tool ...]

Provide only ONE action per turn as a JSON blob: ```{ "action":
$TOOL_NAME, "action_input": $INPUT }```. Valid actions are "Final
Answer" or any tool name. Follow the standard ReAct format (Thought /
Action / Observation), repeating until the final answer.
```

We append the experience block to the system instruction as in AppWorld. The task turn is unchanged. End-to-end arms reuse the shared renderer of Appendix E.3 (Template SR1) without the `<relevant_experience>` wrapper. WorkBench's own retrieval path uses a `<similar_cases>` block (template in the code release). Both carry a `<meta>` line with provenance, reward, step count, domain, and tool sequence.

WorkBench evaluation. Success is state-based. We re-execute the agent's recorded action list against a fresh sandbox and compare it to the ground-truth state. A task passes only if every action executes and every relevant state table matches the target as an unordered set of rows. State changes absent from the ground truth count as side effects and fail the task (plot creation is exempt). This environment replay is used only for scoring, distinct from relabeling's LLM-only second judgment. Reported accuracy is the fraction of non-errored tasks marked correct, averaged over three trials.

F EXPERIENCE UTILIZATION CASE STUDY

We examine three task-experience pairs spanning distinct applications and both benchmarks. They illustrate cross-application entity-level filtering (Appendix F.1), correct answer submission (Appendix F.2), and transfer of a plotting tool-call argument (Appendix F.3). Each case is drawn from a different application or task template to highlight a distinct reuse mechanism. Source/task panels are followed by aligned execution comparisons. Summaries compress traces; verbatim code preserves the selected operation. The agent received the full stored experience; excerpts are for presentation only.

Selection and provenance. The AppWorld cases (C1–C2) are from forced-injection sweeps (forced candidates). W1 is an AgentEAES top-1 selection on the held-out test split. In every case, the target has zero no-experience reward across baseline trials and the injected rollout flips to reward 1 (Table 16). Pool outcome labels refer to the original source rollout (Appendix A).

Table 16: Three illustrative success flips. The two AppWorld cases (C1–C2) are drawn from the forced-injection sweeps, whereas the WorkBench case (W1) is an AgentEAES top-1 selection on the test split. Each target has zero no-experience reward and an injected reward of 1. The Tests column reports passed/total evaluation tests of the injected rollout; for the state-based WorkBench case there is no per-test count.

Case	Benchmark / pool	Source experience ID	Target task ID	Split	Tests
C1	AppWorld / D1	`appworld_6ea6792_2`	`3ab5b8b_1`	dev	6/6
C2	AppWorld / D2	`appworld_287e338_2`	`fac291d_3`	dev	2/2
W1	WorkBench / D2	`workbench_analytics_26`	`analytics_13`	test	–

Traceability. We resolve pool provenance from the effective bank and reproduce injection fingerprints for the AppWorld cases. C2 has a complete prompt archive; C1 is linked by run names and sweep rewards. W1 is reproduced at full fidelity from the end-to-end test-split run; its actions are quoted verbatim. Dev baselines were rerun without rerunning injected outcomes. As purposefully selected single-injection successes, these cases illustrate reuse mechanisms qualitatively.

F.1 C1: From Payment Requests to Song-Level Selection

Case C1: Task and injected source

Target (dev). "Download all the songs from my Spotify playlists that I have liked."

Original source goal. "Reject all pending Venmo payment requests from my friends and roommates."

Hindsight goal (summary). Enumerate pending requests, identify senders among friends and roommates, deny matching requests, and mark completion. The full 16-step source is retained; its original reward is 0.

Source execution (summary). Steps 10–14 enumerate requests and collect contact emails; step 14 filters request objects by sender identity; step 15 issues denials. The following consecutive lines are from step 14:

```
sender_email = req['sender']['email']
if sender_email in friend_emails:
  requests_to_deny.append(req)
```

Without experience	With the C1 experience
Query. All three trials use `show_liked_playlists`.	**Query.** Read the playlist library and liked songs separately: six playlists, 42 distinct playlist songs, and 37 liked songs.
Select. The agent applies the preference to playlists and operates on a 43-song set, rather than testing whether each song is individually liked.	**Select.** Intersect the song-ID sets, obtaining 22 songs.
Outcome. The requested song-level condition is not implemented.	**Act.** After an already-downloaded error, skip seven existing downloads and download the other 15.
Fail: rewards $(0, 0, 0)$**.**	**Success: reward** 1**;** $6/6$ **tests.**

Key target operation (verbatim).

```
songs_to_download = playlist_song_ids & liked_song_ids
```

What changes. The common structure is enumerate-filter-act. The source tests senders against a contact-email set; the target tests song IDs against a liked-song set. Different applications and predicates, but the same pattern. The baseline already supports loops; the improvement is applying the constraint to the correct entities. This case demonstrates a transferable filtering pattern.

F.2 C2: From Artist Aggregation to Correct Answer Submission

Case C2: Task and injected source

Target (dev; summary). Count unique songs across the user's Spotify song library, album library, and playlists.

Source goal, retained without relabeling. "Name the artist least recommended to me on Spotify." The source contains all 15 steps of a successful rollout.

Source execution (summary). Steps 12–13 paginate recommendations; step 14 counts artist occurrences and selects the least frequent; step 15 submits the name. Its final call is:

```
apis.supervisor.complete_task(answer="Evelyn Rose")
```

The source demonstrates an answer-bearing completion call, not song-ID deduplication.

Without experience	With the C2 experience
Compute. All three trials already obtain the correct answer, 95.	**Compute.** Collect 19, 38, and 54 distinct song IDs from the song library, album library, and playlists; their union has 95 songs.
Submit. Trial 1 mixes natural language into code and produces a syntax error. Trials 2 and 3 call `complete_task()` without an answer.	**Check.** Perform additional checks of playlist scope.
Outcome. Correct internal computation does not become a valid submitted answer.	**Submit.** Pass the value explicitly as the completion API's answer argument.
Fail: rewards $(0, 0, 0)$**.**	**Success: reward** 1**;** $2/2$ **tests.**

Key target operation (verbatim).

```
apis.supervisor.complete_task(answer="95")
```

What changes. Artist-frequency aggregation and song-set union solve different questions but both report a derived value through the same completion interface. The observed change is successful answer submission. A correct intermediate result yields reward only when the interaction contract is satisfied. The source experience demonstrates this contract.

F.3 W1: Transferring a Plotting Tool-Call Argument Across Analytics Objectives

Case W1: Task and AgentEAES-selected source (WorkBench, analytics; Qwen3-30B-A3B, test)

Target (test). "please plot the distribution of the total visits metric on September 9"

AgentEAES-selected source goal. "I need a histogram of average session duration and engaged users between November 3 and November 19." The candidate is a successful three-step rollout selected as the top-1 effect score for this target.

Source execution (verbatim, key action). After a first attempt is rejected for an invalid metric name, the source plots each metric with:

```
{"action": "analytics.create_plot", "action_input": {"time_min":
"2023-11-03", "time_max": "2023-11-19", "value_to_plot":
"session_duration_seconds", "plot_type": "histogram"}}
```

The source plots *different* metrics over a *different* window, but demonstrates that a distribution is rendered with `"plot_type": "histogram"`.

Without experience	**With the W1 experience**
Act. Emit a single `analytics.create_plot` call for `total_visits` on September 9, choosing `"plot_type": "bar"`.	**Act.** Emit the same `analytics.create_plot` call for `total_visits` on September 9, but choose `"plot_type": "histogram"`, following the source's plotting argument.
Outcome. A bar chart is produced, but a "distribution" requires a histogram, so the resulting state does not match the target.	**Outcome.** The distribution is rendered as a histogram and the state matches.
Fail: zero reward across baseline trials.	**Success: reward flips $0 \to 1$.**

Key target operation (verbatim).

```
{"action": "analytics.create_plot", "action_input":
{"time_min": "2023-09-09", "time_max": "2023-09-09",
"value_to_plot": "total_visits", "plot_type":
"histogram"}}
```

What changes. The source and target ask for different plots but share the `analytics.create_plot` schema. The no-experience agent maps "distribution" to `"plot_type": "bar"` and fails the state check. The injected demonstration makes `"plot_type": "histogram"` salient for a distribution request. This is a transfer of a tool-call argument value across differing objectives. W1 is an actual AgentEAES selection on the test split; both rollouts are read at full fidelity and the reward flip is verified directly.