EVODISCO: STRATEGY DISCOVERY FOR EVOLVING AGENT MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents accumulate experience in memory, and how that memory evolves during interaction is as important as what it stores. Recent methods update memory online, yet both the operators that transform memory and the rules that select them remain fixed by design. We propose **EvoDisco**, which formulates agent memory evolution as *evolutionary strategy discovery*: the agent must discover which operator to apply, when to apply it, and how to instantiate it, rather than rewriting memory under a static operator schedule. A learnable policy conditions on the current memory state and recent task history, then selects and instantiates one of four semantic operators—ABSTRACT, REFINE, BRIDGE, and REFRAME—which respectively compress related memories, decompose overloaded entries, connect weakly linked experiences, and reorganize retrieved reasoning chains. The policy is updated with a bi-level fitness. An inner signal scores each transformation using a retrieval-quality improvement regularized by memory complexity. An outer signal scores strategic efficacy over a window of past decisions, combining historical operator-utility regret, utility-prediction error, and selection diversity. Under a one-pass online plus frozen-transfer protocol on ALFWorld, WebShop, and LifelongAgentBench (OS and DB), EvoDisco improves over strong memory baselines including MemP and MemRL. The largest gains are on ALFWorld ($+8.8$ online / $+12.9$ frozen-transfer points over MemP; $+16.4$ / $+17.9$ over MemRL). Ablations attribute these gains to both fitness levels and the operator library.

1 INTRODUCTION

Large Language Model (LLM)-based agents have emerged as a powerful paradigm for tackling complex, long-horizon tasks that require reasoning, planning, and multi-turn interaction. However, effective agency in such settings demands more than the ability to solve isolated prompts—agents must accumulate experience, retain task-relevant information across sessions, and selectively reuse past knowledge. This requirement exposes a fundamental limitation of context-only interaction: even with long-context models, relevant information can be diluted or forgotten as interactions grow, leading to unstable recall and degraded reasoning performance.

Memory systems have thus become a core module in LLM-based agents. Recent work has increasingly recognized that *how* memory is organized matters as much as *what* is stored. A growing body of research has moved beyond flat vector retrieval toward structured memory representations, particularly knowledge graphs, which offer superior capabilities for modeling inter-memory relations, enabling multi-hop reasoning, and providing interpretable access. Representative systems include AriGraph (Anokhin et al., 2025), which integrates semantic and episodic memories within a unified knowledge graph; Zep (Rasmussen et al., 2025), which provides a temporal knowledge graph architecture for agent memory; A-MEM (Xu et al., 2025), which adopts a Zettelkasten-inspired interconnected knowledge network; and G-Memory (Zhang et al., 2025a), which employs a hierarchical graph-based memory design. These graph-based approaches have demonstrated significant advantages over unstructured memory in tasks requiring structured access and associative retrieval.

Despite these advances, a critical limitation persists across existing memory systems: **the memory structure itself remains static after its initial construction**. Most structured memory mechanisms rely on one-time restructuring workflows to build their memories, neglecting ongoing up-

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.

dates and optimization of the memory structures. An advanced agent should progressively enrich and refine its memory structure throughout interactions with the environment. While some recent works have begun to address memory evolution—for instance, HAGE (Jiang et al., 2026a) employs reinforcement learning to drive weighted graph evolution, and MemEvolve (Zhang et al., 2025b) introduces a meta-evolutionary framework that jointly evolves experiential knowledge and memory architecture—these approaches still rely on *fixed evolutionary operators* (e.g., mutation and crossover with predetermined probabilities) or *manually engineered adaptation mechanisms*.

This raises a fundamental question: **If the agent itself is capable of reasoning about its own memory, why should the *way* memory evolves be predetermined by human designers?**

In this paper, we propose **EvoDisco** (Strategy Discovery for Evolving Agent Memory), a novel framework that elevates agent memory evolution from *rewriting memory under a fixed operator schedule* to *discovering how memory should evolve*. Unlike prior work that treats evolution as a blind search using fixed genetic operators, EvoDisco enables the agent to *discover which evolutionary operation to apply, when to apply it, and how to instantiate it* based on the current memory state and recent task performance. The agent maintains a **strategy policy**—a learnable module that selects from a curated cognitive operator library (abstraction, refinement, bridging, and reframing)—and the *policy itself* undergoes evolution guided by a bi-level fitness function that evaluates both the immediate utility of each operation and the strategic efficacy of the agent's historical decisions. Figure 1 gives an overview.

Our contributions are threefold:

- **Evolutionary Strategy Discovery.** We formulate memory evolution as a learnable strategy-selection problem, shifting from a fixed operator schedule to state-conditioned selection and instantiation of semantic memory transformations.
- **Bi-Level Fitness with Strategic Efficacy.** We introduce a dual fitness framework that evaluates the retrieval-quality change and structural cost of each memory update, together with the policy's historical accuracy in selecting high-utility operators.
- **Cognitive Operator Library.** We design a curated set of four semantic-level operators—ABSTRACT (compressing related memories into higher-level concepts), REFINE (decomposing overloaded memories), BRIDGE (connecting weakly linked experiences), and REFRAME (reorganizing retrieved reasoning chains)—that provide sufficient expressive power while maintaining controllability.

Through extensive experiments on ALFWorld, WebShop, and LifelongAgentBench (OS and DB), we evaluate the learned policy against fixed-strategy and memory baselines.

2 Related Work

2.1 Memory Systems for LLM Agents

Memory lets an agent retain experience beyond the current context. Early systems store transcripts, embeddings, or key-value records and retrieve them by semantic similarity (Lewis et al., 2020). Similarity search is effective for direct lookup, but it provides little support for multi-hop relations or for deciding how stored experience should be reorganized.

Graph-structured memory addresses the relational part of this problem. GraphRAG-style pipelines extract entities and relations and use the resulting graph for retrieval and summarization (Edge et al., 2024; Li & Du, 2023; Panda et al., 2024; Li et al., 2024). Agent memory systems then keep such graphs online: AriGraph (Anokhin et al., 2025) links semantic and episodic memories, Zep (Rasmussen et al., 2025) indexes a temporal knowledge graph, A-MEM (Xu et al., 2025) grows a Zettelkasten-style note network, and G-Memory (Zhang et al., 2025a) organizes memory hierarchically for multi-agent settings. Other systems do not rely on an explicit graph. Mem0 merges atomic facts (Chhikara et al., 2025), MemP distills trajectories into procedures (Fang et al., 2026), Self-RAG gates retrieved text (Asai et al., 2024), and MemRL ranks episodic notes with a non-parametric Q-value while the backbone stays frozen (Zhang et al., 2026). Surveys organize these designs by structure and function (Zhang et al., 2024). A separate limitation cuts across them: most

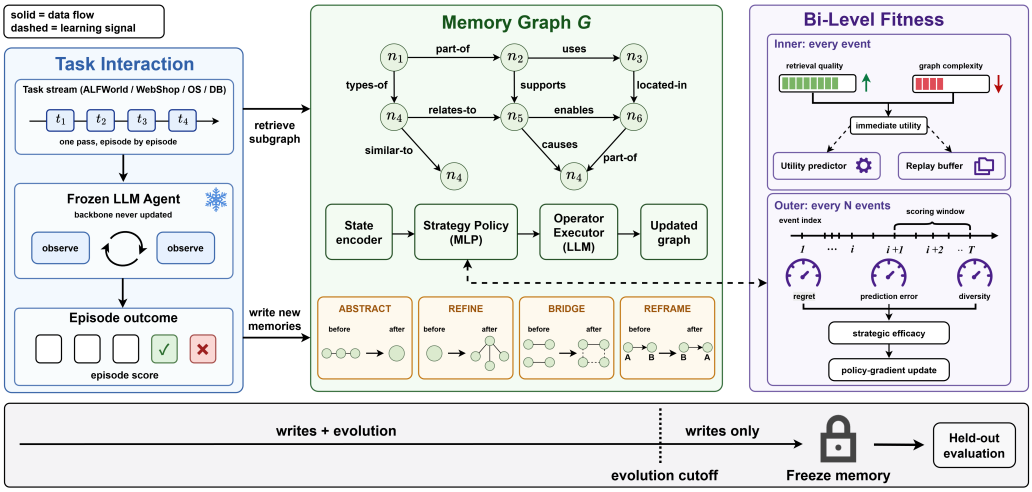


Figure 1: Overview of EvoDisco. A frozen language model interacts with a one-pass task stream, retrieves a subgraph, and writes new memories after each episode. A strategy policy reads the memory-graph state and selects one of four operators—ABSTRACT, REFINE, BRIDGE, and REFRAME—which the operator executor applies to the graph. Immediate utility scores each edit by the change in retrieval quality regularized by graph complexity. Every N events, strategic efficacy combines operator-utility regret, utility-prediction error, and selection diversity, and updates the policy. Evolution runs only before the cutoff; afterward the agent continues to write memories. Held-out evaluation freezes the accumulated memory.

structured memories are built by a one-time workflow and are not subsequently reorganized as the task distribution changes (Luo et al., 2026).

2.2 Memory Evolution

Several systems do update memory over time. Production agents already separate declarative, procedural, and episodic stores and run background consolidation to distill or prune them (Packer et al., 2023). In the research literature, MemEvolve jointly evolves experiential knowledge and the memory architecture (Zhang et al., 2025b); EvolveMem treats retrieval configuration as an action space tuned by an LLM diagnosis module (Liu et al., 2026b); HAGE trains edge weights on a graph from downstream task signals (Jiang et al., 2026a); MemRL keeps the topology fixed and updates only the utility scores used at retrieval (Zhang et al., 2026); and FORGE evolves natural-language memory with a population-based protocol (Bogdanov et al., 2026). Surveys describe this line as a shift from storing trajectories, to reflecting on them, to abstracting them into reusable experience (ang Gao et al., 2026; Luo et al., 2025).

What stays fixed is the evolutionary mechanism itself. Mutation and crossover use predetermined probabilities, edge updates follow a chosen reinforcement-learning rule, and configuration search stays inside a human-specified action space. The agent can change memory content, weights, or a small set of architecture parameters, but it does not learn which kind of reorganization is appropriate for the current memory state.

2.3 Strategy Selection and Evolutionary Search

Choosing among strategies is well studied when the strategy is an external behavior. SMART models strategy selection for reasoning as a Markov decision process (Liu et al., 2024). MetaClaw continually updates both a base policy and a library of reusable skills (Xia et al., 2026). Meta-reinforcement learning gives a broader account of adapting from environmental feedback at test time (Jiang et al., 2026b). These methods select an action in the task, not an edit to the agent's own knowledge struc-

ture. Metacognitive work on LLM agents similarly monitors answers and reasoning traces (Liu et al., 2026a), rather than the rule used to reorganize memory.

Evolutionary computation supplies the other nearby toolset. Genetic programming and Genetic Network Programming evolve behaviors or graph-structured programs (Koza, 1994; Mabu et al., 2007); LLM-in-the-loop methods mutate prompts or code (Guo et al., 2025; Morris et al., 2024); and EvoLattice stores a population of candidate behaviors in one directed acyclic graph (Yuksel, 2025). The operators in these loops are part of the algorithm design. When evolution is applied to memory, it typically decides which experiences to retain, not how the memory graph should be restructured, and the operator set remains constant.

EvoDisco targets that missing choice. Relative to MemEvolve, the learned object is the policy that selects operators, not a schedule of architecture parameters. Relative to HAGE, the agent chooses among qualitatively different semantic edits rather than training one edge representation. Relative to SMART and MetaClaw, the action is an internal reorganization of memory. Bi-level fitness then scores both the immediate utility of an edit and the policy's historical accuracy in choosing high-utility operators.

3 METHOD

EvoDisco learns a state-dependent strategy for reorganizing an agent's persistent memory. Task interactions supply new memories, cognitive operators transform their organization, and observed outcomes guide subsequent operator selection. The backbone language model remains frozen; adaptation occurs through the memory graph and a lightweight strategy policy. Figure 1 summarizes this loop.

3.1 PROBLEM SETUP AND MEMORY

An agent processes tasks drawn from a distribution $\mathcal{T}$ while maintaining a memory graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. For a fixed graph, the population objective is

$$F(\mathcal{G}) = \mathbb{E}_{t\sim\mathcal{T}}\left[\text{score}(t;\mathcal{G})\right]. \tag{1}$$

We distinguish this expectation from the observed score y_j of task episode j.

Memory is a directed, multi-relational graph. Nodes store facts, episode summaries, procedures, or concepts; typed edges encode semantic and structural relations. Each node v stores an embedding $\mathbf{e}_v$, a timestamp t_v, a confidence $c_v \in [0,1]$, and a retrieval count r_v. An edge is a typed relation (u,ρ,v) with type $\rho \in \mathcal{R}$, strength $w_e \in [0,1]$, and evidence-recency τ_e. The fixed vocabulary includes `causes`, `precedes`, `analogous-to`, `is-a`, and `generalizes`. After each episode, an LLM-based extractor writes memory units and candidate relations. During task execution, query-centered spreading activation retrieves a subgraph for the agent's context.

At evolution event i, the agent encodes the current graph and recent history as a state s_i, then selects and executes

$$a_i = (o_i, c_i, \theta_i), \qquad o_i \in \mathcal{O}, \tag{2}$$

$$\mathcal{G}_i^{+} = \mathsf{T}_{o_i}(\mathcal{G}_i; c_i, \theta_i), \tag{3}$$

where c_i indexes a graph candidate and θ_i is the instantiation. Events index triggered attempts, including attempts that leave the graph unchanged, rather than task episodes. Ordinary memory writes may occur between events. The state s_i concatenates four groups: graph statistics (node and edge counts, mean degree, modularity, and path length); retrieval statistics (latency, hit rate, and retrieved-node diversity); recent outcomes (the latest operator choices and utilities, together with recent observed scores y_j); and structural-deficiency summaries (high-degree nodes, disconnected components, and long paths). Episode-score features use y_j, not $F(\mathcal{G})$.

3.2 COGNITIVE OPERATOR LIBRARY

The fixed library $\mathcal{O} = \{\mathcal{O}_{\text{abs}}, \mathcal{O}_{\text{ref}}, \mathcal{O}_{\text{brg}}, \mathcal{O}_{\text{refr}}\}$ contains four semantic transformations. EvoDisco learns their selection and instantiation, not additional operator categories.

Abstract: consolidate related memories. Given a seed neighborhood, $\mathcal{O}_{\text{abs}}$ synthesizes a higher-level concept or reusable procedure and connects it to supporting memories through `generalizes` edges. A configured pruning criterion optionally removes redundant memories.

Refine: separate overloaded memories. $\mathcal{O}_{\text{ref}}$ decomposes a broad memory into more selective units along a temporal, causal, or categorical dimension. The original node is retained as a summary or replaced, depending on the instantiation.

Bridge: connect separated knowledge. $\mathcal{O}_{\text{brg}}$ identifies semantically related but weakly connected memory regions and generates typed relations between them.

Reframe: reorient edges on a path. $\mathcal{O}_{\text{refr}}$ keeps every node's text and adds no node. It rewrites `causes` and `precedes` edges along one existing path. The policy selects the frame, and one frozen LLM call returns a reorientation; the executor then reverses or relabels each consecutive pair (Appendix A.8).

The executor proposes at most eight candidates for each operator by a fixed rule, and masks any operator whose set is empty. ABSTRACT ranks seeds whose 2-hop neighborhood contains at least two nodes, by neighborhood size and retrieval count. REFINE ranks hubs by degree. BRIDGE ranks component pairs at graph distance at least 2 by embedding similarity; if none exists and the graph has at least six nodes, it proposes one pair from a two-means split of the embeddings. REFRAME uses a fixed sample of long paths. A second head selects the candidate index by a dot product between candidate features and a learned projection of s.

Five categorical heads are sampled from s on every attempt, including heads the chosen operator does not read. Neighborhood radius $k \in \{1, 2, 3\}$ and pruning level $p \in \{0, 0.5\}$ are read by ABSTRACT, and BRIDGE also reads p. Child count $m \in \{2, 3, 4\}$ and a dimension in $\{\text{temporal}, \text{causal}, \text{categorical}\}$ are read by REFINE. A logical frame in $\{\texttt{temporal_to_causal}, \texttt{causal_to_temporal}, \texttt{effect_cause}\}$ is read by REFRAME. The frozen LLM writes new node text for ABSTRACT, REFINE, and BRIDGE only after o, c, and θ are fixed. The REFRAME call returns a reorientation and writes no node. Similarity cutoffs controlled by p are fixed rules (Appendix A.8).

3.3 BI-LEVEL FITNESS FRAMEWORK

Feedback operates at two timescales: immediate utility evaluates individual transformations, and strategic efficacy guides policy updates. Here, "bi-level" denotes these feedback timescales, not an exact nested optimization procedure.

3.3.1 IMMEDIATE UTILITY

For $\mathcal{G}_i \to \mathcal{G}_i^+$, define

$$U_i = \alpha\, \widehat{\Delta F}_i - \beta \left[C(\mathcal{G}_i^+) - C(\mathcal{G}_i) \right], \tag{4}$$

where $\alpha, \beta > 0$. The proxy is the retrieval-quality change on queries already observed before the attempt,

$$\widehat{\Delta F}_i = \Delta Q_i = Q(\mathcal{G}_i^+; \mathcal{Q}_i) - Q(\mathcal{G}_i; \mathcal{Q}_i), \tag{5}$$

Let $\mathcal{Q}_i^+$ be the cached queries that activate at least one node, and write $\text{Top}_3(\mathcal{G}, \mathbf{q})$ for the three largest spreading-activation scores. Retrieval quality is

$$Q(\mathcal{G}; \mathcal{Q}_i) = \frac{1}{|\mathcal{Q}_i^+|} \sum_{\mathbf{q} \in \mathcal{Q}_i^+} \text{mean}\big(\text{Top}_3(\mathcal{G}, \mathbf{q})\big), \tag{6}$$

and $Q = 0$ when $\mathcal{Q}_i^+$ is empty. Both graphs are scored on the same cache immediately after the edit, so U_i does not use later episode scores. $\widehat{\Delta F}_i$ is this graph-local surrogate. It is not an estimate of $F(\mathcal{G}_i^+) - F(\mathcal{G}_i)$. Structural complexity is

$$C(\mathcal{G}) = \boldsymbol{\lambda}^\top \mathbf{c}(\mathcal{G}), \tag{7}$$

where $\mathbf{c}(\mathcal{G})$ contains node-count, edge-density, and path-length statistics. An attempt that commits no edit records $U_i = -0.05$ and does not apply Eq. 4. The query cache is specified in Appendix A.6.

Utility is an ex post learning signal: the transformed graph is retained regardless of the sign of U_i, with no utility-based rollback. Recorded utilities inform subsequent states, strategic efficacy, and utility-predictor training.

3.3.2 Strategic Efficacy

Before executing a_i, the predictor records $\hat{U}_i = q_\psi(s_i, a_i)$. Over the most recent $m_i = \min(K, i)$ attempts, collected in $\mathcal{W}_i$, we compute

$$E_i = -\gamma \widehat{\mathrm{Regret}}_i - \delta \,\mathrm{PredErr}_i + \eta \,\mathrm{Div}_i. \tag{8}$$

For each operator, $\bar{U}_i(o)$ is the mean utility of its most recent $h = 8$ executions, or of every execution when fewer are available. Let $\mathcal{O}_i^{\mathrm{obs}}$ be the operators with a nonempty history, and write $[x]_+ = \max(x, 0)$. With empirical frequency $p_i(o) = m_i^{-1} \sum_{\ell \in \mathcal{W}_i} \mathbb{I}[o_\ell = o]$,

$$\widehat{\mathrm{Regret}}_i = \frac{1}{m_i} \sum_{\ell \in \mathcal{W}_i} \Big[\max_{o \in \mathcal{O}_i^{\mathrm{obs}}} \bar{U}_i(o) - U_\ell \Big]_+, \tag{9}$$

$$\mathrm{PredErr}_i = \frac{1}{m_i} \sum_{\ell \in \mathcal{W}_i} |\hat{U}_\ell - U_\ell|, \tag{10}$$

$$\mathrm{Div}_i = -\sum_{o \in \mathcal{O}} p_i(o) \log p_i(o), \tag{11}$$

using $0 \log 0 = 0$. $\bar{U}_i(o)$ is a truncated arithmetic mean. The positive part is zero when realized utility exceeds that reference. Because the reference mixes different states and instantiations, $\widehat{\mathrm{Regret}}_i$ is not same-state counterfactual regret. Div_i is a soft preference for diversity, not a requirement of equal usage. The window includes attempts that do not commit an edit.

3.4 Strategy Policy Learning

A two-layer MLP parameterized by ϕ defines

$$\pi_\phi(a \mid s) = \pi_\phi(o \mid s)\, \pi_\phi(c \mid s, o)\, \pi_\phi(\theta \mid s), \tag{12}$$

where o is the operator, c is one candidate from the executor-built set for o, and θ collects the five instantiation heads. Executor construction rules and frozen-LLM generations do not enter the log-probability. The weighted log-probability used for learning is

$$\log \pi_\phi^{\mathrm{w}}(a \mid s) = \log \pi_\phi(o \mid s) + \log \pi_\phi(c \mid s, o) + \tfrac{1}{4} \sum_j \log \pi_\phi(\theta_j \mid s), \tag{13}$$

The factor $1/4$ is a fixed scale on those heads and is outside each categorical normalizer.

Every N attempts, the policy receives a score-function update using strategic efficacy as a window-level return:

$$\widehat{\nabla_\phi J}_i = \mathrm{sg}(E_i) \sum_{\ell \in \mathcal{B}_i} \nabla_\phi \log \pi_\phi^{\mathrm{w}}(a_\ell \mid s_\ell), \tag{14}$$

where $\mathcal{B}_i$ contains new events since the preceding policy update and sg treats the observed feedback as fixed. The last-K scoring window and the update batch have distinct roles: older events used to calculate E_i are not reused as fresh on-policy samples.

A utility predictor $q_\psi(s_i, a_i)$ is supervised by recorded U_i values. State–action pairs with $U_i > 0$ enter a replay buffer $\mathcal{D}_+$. Scheduled updates minimize

$$\mathcal{L}_{\mathrm{replay}}(\phi) = -\mathbb{E}_{(s,a) \sim \mathcal{D}_+} \left[\log \pi_\phi^{\mathrm{w}}(a \mid s) \right]. \tag{15}$$

This auxiliary objective is separate from the policy-gradient update; eligibility depends on utility, not task success alone.

Table 1: Online and frozen-memory transfer success rates (%). Online SR is measured over the full training stream. Transfer SR is measured on the held-out split with all memory writes and evolution disabled. Bold denotes the highest value in each column.

	ALFWorld		WebShop		OS		DB	
Method	Online	Transfer	Online	Transfer	Online	Transfer	Online	Transfer
No Memory	56.7	52.9	32.2	32.5	67.7	72.7	88.1	84.9
RAG	60.8	70.0	29.8	33.0	70.9	72.0	80.9	72.7
Self-RAG	56.0	62.9	31.8	32.5	69.7	74.7	85.3	84.2
Mem0	58.0	57.9	29.6	28.0	67.7	71.3	85.9	86.3
MemP	68.9	75.7	31.4	34.0	71.7	76.0	83.1	73.4
MemRL	61.3	70.7	31.2	33.0	68.6	72.7	83.7	74.1
EvoDisco	**77.7**	**88.6**	**32.8**	**35.0**	**74.0**	**77.3**	**88.4**	**87.1**

3.5 Online Learning and Evaluation

The agent processes the online stream once, writing memories after each episode. Evolution is enabled only during the first $\rho = 0.8$ fraction of episodes. Within this interval, an attempt is triggered either after T episodes without evolution or by a moving-average score decline exceeding ϵ (Appendix A.9). Both triggers are disabled after the cutoff, while ordinary memory writes continue. Algorithm 1 in the appendix summarizes the procedure.

For held-out evaluation, the accumulated memory is frozen: retrieval remains active, but writes and evolution are disabled. This frozen-memory evaluation differs from cross-benchmark policy transfer, where policy parameters are frozen while a new target-domain memory is constructed online.

4 Experiments

Our experiments examine whether adaptive memory evolution improves online task performance, whether the resulting memory remains useful on held-out tasks, and whether the learned operator-selection policy transfers across environments. We evaluate EvoDisco against six baselines on four interactive benchmarks, followed by component ablations, cross-benchmark policy transfer, and an analysis of computational cost. Representative graph edits are reported in Appendix B.4.

4.1 Experimental Setup

Baselines & Benchmarks. We compare EvoDisco against RAG-based (RAG, Self-RAG), agentic-memory (Mem0, MemP), and MemRL baselines, together with a no-memory agent, under a frozen-backbone setting (Appendix A.3). Evaluations span four benchmarks: ALFWorld (household interaction), WebShop (web shopping), LifelongAgentBench-OS (system administration), and LifelongAgentBench-DB (database manipulation). Details are in Appendix A.2. Every method uses the same frozen backbone, GPT-5-mini; implementation choices are in Appendix A.4.

Metrics. We report success rate (SR), the fraction of episodes completed successfully. Online SR is this ratio over the training stream, and transfer SR is the same ratio on the held-out split.

We evaluate EvoDisco and the baselines under two settings. **Runtime Learning** assesses the ability to learn and adapt within a single pass over the training stream. **Transferring** evaluates whether the learned memory generalizes to held-out tasks from the same benchmark once writes and evolution are disabled. The protocol is specified in Appendix A.1.

4.2 Main Results

Table 1 reports full-stream online SR and frozen-memory transfer SR. Cumulative online trajectories and success rates over the final 20% are reported in Appendix B.1. All results are expressed as percentages; differences discussed below are in percentage points.

Table 2: ALFWorld ablations. Online and Transfer denote success rates (%). ΔTransfer is the difference from Full in percentage points. Bold denotes the highest success rate in each column.

Variant	Online	Transfer	ΔTransfer
EvoDisco (Full)	**77.7**	**88.6**	—
w/o Outer Fitness	75.3	80.7	−7.9
w/o Inner Fitness	76.0	79.3	−9.3
w/o Cognitive Operators	76.3	82.1	−6.5
Fixed Abstract	76.0	85.7	−2.9
Fixed Refine	75.2	81.4	−7.2
Fixed Bridge	75.9	80.7	−7.9
Fixed Reframe	76.0	80.0	−8.6

EvoDisco achieves the highest observed online and frozen-transfer SR on all four benchmarks. The largest gains occur on ALFWorld, where it reaches 77.7% online SR and 88.6% transfer SR, exceeding the strongest baseline, MemP, by 8.8 and 12.9 points, respectively. Relative to MemRL, the corresponding gains are 16.4 and 17.9 points. On OS, EvoDisco improves over MemP from 71.7% to 74.0% online and from 76.0% to 77.3% in transfer. The gains are smaller on WebShop and DB: transfer SR exceeds the strongest baseline by 1.0 and 0.8 points, respectively. These smaller margins should be interpreted as numerical differences rather than evidence of statistical significance.

The benefit of memory varies substantially across environments. On DB, several memory-based baselines underperform No Memory, whereas EvoDisco reaches 88.4% online SR and 87.1% transfer SR, compared with 88.1% and 84.9% without memory. Adding memory therefore does not uniformly improve performance in this setting.

EvoDisco also achieves the highest late-stream SR on ALFWorld, WebShop, and OS, and ties No Memory on DB (Appendix Table 8). Appendix Figure 2 shows that the ALFWorld lead is present throughout training, rather than only in the final aggregate. EvoDisco's final-20% SR exceeds its full-stream average on ALFWorld (82.4% versus 77.7%), WebShop (34.0% versus 32.8%), and OS (80.0% versus 74.0%). On WebShop, it is the only evaluated method with a late-stream SR above its overall online SR. Because evolution is disabled during this interval, these results are consistent with continued utility of the evolved memory alongside ordinary writes. However, the comparison involves different subsets of tasks and does not by itself isolate the effect of memory growth.

4.3 Ablation Studies

We ablate immediate utility, strategic efficacy, and the cognitive operator library, and replace learned operator selection with each of the four fixed-operator policies (Appendix A.12). Table 2 reports ALFWorld, where Full matches the main run in Table 1. OS ablations, operator-usage statistics, and strategic-efficacy trajectories are in Appendices A.13, B.2, and B.3.

Removing either fitness signal reduces both online and transfer performance, with substantially larger differences in frozen transfer. Without outer fitness, transfer SR decreases from 88.6% to 80.7%; without inner fitness, it decreases to 79.3%. The corresponding online decreases are 2.4 and 1.7 points. This pattern suggests that the two feedback signals contribute not only to performance during memory construction but also to the subsequent usefulness of the resulting graph. Removing the cognitive operators likewise reduces transfer SR to 82.1%, a 6.5-point decrease from Full.

Among the fixed-operator variants, ABSTRACT performs best on ALFWorld, reaching 85.7% transfer SR, but remains 2.9 points below Full. Fixed REFINE, BRIDGE, and REFRAME trail Full by 7.2, 7.9, and 8.6 points, respectively. The ordering differs on OS: REFINE is the strongest fixed operator online at 72.9%, whereas BRIDGE is strongest in transfer at 76.0%, compared with 77.3% for Full. Appendix B.2 does not reproduce those rankings as commit frequencies. On the logged runs, REFRAME is the most frequent committed edit on both benchmarks, and ABSTRACT commits least often because later attempts hit the abstraction cap. All four operators still occur at least once. The fixed-operator gaps and the commit frequencies are therefore different measurements.

Table 3: Cross-benchmark transfer of the operator-selection policy. Values are frozen-memory transfer success rates (%). The ALFWorld-trained policy is held fixed while each target graph is constructed from scratch. ΔTransfer is relative to the target-trained Full model, in percentage points.

Policy Training	WebShop	OS	DB
ALFWorld	31.0	75.3	82.7
Target Benchmark (Full)	35.0	77.3	87.1
ΔTransfer	−4.0	−2.0	−4.4

4.4 Analysis

4.4.1 Cross-Benchmark Policy Transfer

Frozen-memory evaluation tests whether a graph transfers to held-out tasks within its original benchmark. We separately examine whether the operator-selection policy π transfers across benchmarks. Specifically, we initialize the target agent with the MLP weights learned on ALFWorld and keep those weights fixed throughout the target run. Each target memory graph is constructed from scratch using the same one-pass online and frozen-memory evaluation protocol, with no further REINFORCE updates. This experiment therefore transfers the policy, not the ALFWorld memory graph, and still permits memory construction from target training episodes.

As shown in Table 3, reusing the ALFWorld-trained policy yields 31.0%, 75.3%, and 82.7% transfer SR on WebShop, OS, and DB, which are 4.0, 2.0, and 4.4 points below the target-trained models. Appendix B.2 is a separate commit log, not the transferred checkpoint, so the gaps should not be read off those frequencies. These results do not establish that the source-trained policy is preferable to an untrained selector; that comparison would require an additional control.

4.4.2 Computational Cost

An EvoDisco evolution step consists of local graph-statistics computation and state encoding, a forward pass through the policy network, and an LLM call when the selected operator requires generation or relation inference. Immediate utility is computed locally from retrieval quality and graph complexity; the default implementation does not invoke a separate LLM call to estimate operator effects. Thus, an evolution event requires approximately one operator-generation call when applicable, in addition to local computation, rather than evaluating a population of graph variants through repeated task rollouts.

In the reported ALFWorld runs, end-to-end wall-clock time is 21.6 hours for EvoDisco, compared with 33.6 hours for MemRL and 36.8 hours for No Memory. On OS, EvoDisco completes in 4.2 hours, compared with 6.1 hours for MemRL. These measurements include task execution and memory maintenance: successful episodes can end early, while unsuccessful ALFWorld episodes may reach the 50-step cap, so the lower runtime is total agent cost rather than isolated evolution overhead.

5 Conclusion

EvoDisco learns which cognitive operator to apply when evolving agent memory, instead of following a hand-specified rule. Its policy selects among Abstract, Refine, Bridge, and Reframe and is trained with bi-level feedback: retrieval-quality gain regularized by structural complexity, together with operator-utility regret, utility-prediction error, and selection diversity. The study is limited by a fixed library of four operators, a trigger that is scheduled or heuristic rather than learned, and evaluation with a single backbone and an ALFWorld-only transfer source. Within that scope, EvoDisco matches or exceeds MemP and MemRL on ALFWorld, WebShop, and LifelongAgentBench (OS and DB). The largest gain is on ALFWorld, where online and frozen-transfer success rise by 8.8 and 12.9 points over MemP, and ablations tie that transfer gap to both fitness signals and learned operator selection.

AI USE STATEMENT

We used generative AI tools to assist with polishing the manuscript text. We have reviewed all AI-assisted writing, and we take responsibility for the final content of this work, including text, claims, and any artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies memory evolution for LLM-based agents on publicly available academic benchmarks (ALFWorld, WebShop, and LifelongAgentBench OS/DB). It does not involve human subjects, private user data, or the release of sensitive datasets. We do not foresee significant dual-use or fairness risks beyond those already associated with general-purpose LLM agents. Upon publication, we will release code and experimental configurations to support transparent evaluation.

REPRODUCIBILITY STATEMENT

We provide complete experimental settings in Section 4.1 and Appendix A, including the one-pass online / frozen-transfer protocol, split sizes, baselines, hyperparameters, and the full EvoDisco algorithm (Algorithm 1). Upon publication, we will release source code, per-benchmark YAML configs, and the scripts used to produce the reported tables.

REFERENCES

Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence, 2026. URL https://arxiv.org/abs/2507.21046.

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Andrey Kravchenko, Mikhail Burtsev, and Evgeny Burnaev. Arigraph: Learning knowledge graph world models with episodic memory for llm agents, 2025. URL https://arxiv.org/abs/2407.04363.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avi Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *International conference on learning representations*, volume 2024, pp. 9112–9141, 2024.

Igor Bogdanov, Chung-Horng Lung, Thomas Kunz, Jie Gao, Adrian Taylor, and Marzia Zaman. Forge: Self-evolving agent memory with no weight updates via population broadcast. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 292–310. ACM, May 2026. doi: 10.1145/3786335.3813155. URL http://dx.doi.org/10.1145/3786335.3813155.

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

Darren Edge et al. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers, 2025. URL https://arxiv.org/abs/2309.08532.

Dongming Jiang, Yi Li, Guanpeng Li, Qiannan Li, and Bingzhe Li. Hage: Harnessing agentic memory via rl-driven weighted graph evolution, 2026a. URL https://arxiv.org/abs/2605.09942.

Yulun Jiang, Liangze Jiang, Damien Teney, Michael Moor, and Maria Brbic. Meta-rl induces exploration in language agents, 2026b. URL https://arxiv.org/abs/2512.16848.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, 2020.

Ruosen Li and Xinya Du. Leveraging structured information for explainable multi-hop question answering and reasoning, 2023. URL https://arxiv.org/abs/2311.03734.

Shilong Li, Yancheng He, Hangyu Guo, Xingyuan Bu, Ge Bai, Jie Liu, Jiaheng Liu, Xingwei Qu, Yangguang Li, Wanli Ouyang, Wenbo Su, and Bo Zheng. Graphreader: Building graph-based agent to enhance long-context abilities of large language models, 2024. URL https://arxiv.org/abs/2406.14550.

Gabrielle Kaili-May Liu, Areeb Gani, Jacqueline Lu, Jordan Thomas, Mark Steyvers, and Arman Cohan. Metacognition in llms: Foundations, progress, and opportunities, 2026a. URL https://arxiv.org/abs/2607.11881.

Jiaqi Liu, Xinyu Ye, Peng Xia, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Evolvemem:self-evolving memory architecture via autoresearch for llm agents, 2026b. URL https://arxiv.org/abs/2605.13941.

Rongxing Liu, Kumar Shridhar, Manish Prajapat, Patrick Xia, and Mrinmaya Sachan. Smart: Self-learning meta-strategy agent for reasoning tasks, 2024. URL https://arxiv.org/abs/2410.16128.

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, 2026. URL https://arxiv.org/abs/2605.06716.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A graph-based evolutionary algorithm: Genetic network programming (gnp) and its extension using reinforcement learning. *Evolutionary computation*, 15(3):369–398, 2007.

Clint Morris, Michael Jurado, and Jason Zutty. Llm guided evolution-the automation of models advancing models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 377–384, 2024.

Charles Packer et al. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Pranoy Panda, Ankush Agarwal, Chaitanya Devaguptapu, Manohar Kaul, and Prathosh A P. Holmes: Hyper-relational knowledge graphs for multi-hop question answering using llms, 2024. URL https://arxiv.org/abs/2406.06027.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory, 2025. URL https://arxiv.org/abs/2501.13956.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Metaclaw: Just talk – an agent that meta-learns and evolves in the wild, 2026. URL https://arxiv.org/abs/2603.17187.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents, 2025. URL https://arxiv.org/abs/2502.12110.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022.

Kamer Ali Yuksel. Evolattice: Persistent internal-population evolution through multi-alternative quality-diversity graph representations for llm-guided program discovery, 2025. URL https://arxiv.org/abs/2512.13857.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems, 2025a. URL https://arxiv.org/abs/2506.07398.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems, 2025b. URL https://arxiv.org/abs/2512.18746.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents, 2024. URL https://arxiv.org/abs/2404.13501.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

A Additional Implementation Details

This appendix provides the experimental protocol, benchmarks, baselines, and implementation choices deferred from Section 4.1, together with further implementation details for EvoDisco, including the shared hyperparameter setting, operator instantiation settings, the evolution-trigger configuration, the execution of the three component ablations, OS ablations (Table 7), operator-usage statistics, strategic-efficacy trajectories, two qualitative graph edits (Figure 5), and the shared prompt templates (Appendix C).

A.1 Evaluation Protocol

We adopt a *one-pass online learning followed by frozen-memory transfer* protocol, with the backbone LLM held fixed throughout. Runtime Learning is the online phase: each agent processes the training split once and may write memories after every episode. For methods with memory evolution, evolution is restricted to the first 80% of the training stream. Ordinary memory writes remain enabled throughout the stream, including the final 20%.

We report success rate (SR) over the full online stream and separately over its final 20%. The latter measures late-stream performance after evolution has stopped, although ordinary memory accumulation continues; those rates are in Appendix B.1. Transferring is the frozen-memory phase. After the online pass, we freeze all memory state and evaluate on the held-out split without further writes or evolution. Frozen-memory transfer is our primary measure of whether the accumulated memory supports unseen tasks within the same benchmark.

A.2 BENCHMARKS

We evaluate on four benchmarks chosen to cover household interaction, web shopping, operating-system administration, and database manipulation. Within each benchmark, every method uses the same agent loop, split files, backbone LLM, and step budget. Split sizes and budgets are summarized in Table 6.

- **ALFWorld** (Shridhar et al., 2021): A text-based household environment for embodied navigation and manipulation. Each episode states a goal in natural language, such as finding an object and placing it at a destination, and the agent must issue a sequence of high-level actions (navigate, open, take, and place) until the goal holds or the step limit is reached. Success therefore depends on retrieving a multi-step plan that matches the current scene, rather than on a single lookup. We use all 3553 training episodes and the 140-task in-distribution evaluation split, with a budget of 50 steps per episode.
- **WebShop** (Yao et al., 2022): A simulated shopping website on which the agent must satisfy a natural-language product instruction by searching, selecting an item, and choosing options. The environment returns a page observation at each step, so the agent has to keep the instruction constraints in view while navigating search results. In addition to binary success, WebShop assigns a graded purchase score that reflects how closely the chosen product matches the instruction. We use 500 training instructions and 200 validation instructions, with a budget of 15 steps per episode. Reported tables use binary success.
- **LifelongAgentBench-OS** (Zheng et al., 2025): Interactive Ubuntu/Bash administration tasks. The agent issues shell commands, reads their output, and must finish the requested administration steps over a multi-turn session. The benchmark is designed for continual tool use: later tasks reuse commands and patterns that appeared in earlier tasks. We use the official MemRL split of 350 training tasks and 150 validation tasks, with a budget of 15 steps per episode.
- **LifelongAgentBench-DB** (Zheng et al., 2025): SQL interaction with a live database. The agent writes queries, inspects results, and revises them until the requested table state or answer is obtained. Tasks share a schema, so a query pattern learned on one task can apply to a later request against the same tables. We use the official split of 361 training tasks and 139 validation tasks, with a budget of 15 steps per episode.

A.3 BASELINES

We compare against protocol-matched reimplementations of the memory systems in the MemRL evaluation (Zhang et al., 2026). Every baseline uses the same frozen backbone, agent loop, splits, and step budgets as EvoDisco. During Runtime Learning a baseline may write after each episode; during Transferring its memory is frozen. None of these systems selects or instantiates a graph-evolution operator.

- **No Memory**: The agent conditions only on the current episode: the task text, the latest observation, and the actions already taken in that episode. It does not write a note, embed past experience, or retrieve anything from previous episodes. This baseline measures task performance when no cross-episode memory is available.
- **RAG** (Lewis et al., 2020): After each episode the backbone writes a reflection of the trajectory and stores it as an embedding. At the start of a new episode the system embeds the task and returns the top-k notes by cosine similarity. Retrieval ranks notes by cosine similarity alone. Both successful and failed episodes are stored, so later retrieval can surface either kind of note.
- **Self-RAG** (Asai et al., 2024): Self-RAG uses the same note store as RAG and adds a decide-then-critique filter. Before retrieval, the backbone answers whether stored memories would help the current task; a negative answer skips retrieval. When retrieval runs, each candidate is marked keep or drop, and only the kept notes are passed to the agent. The filter is intended to suppress notes that are similar in embedding space but not useful for the current instruction.

- **Mem0** (Chhikara et al., 2025): A memory layer that maintains a persistent store through add, retrieve, and update operations. After an episode it extracts one to three atomic facts—a lesson, a constraint, or a working pattern—rather than saving the raw trajectory. Each fact is embedded and compared with existing notes. If the nearest note has cosine similarity of at least 0.82, that note is overwritten with the new fact; otherwise the fact is inserted as a new note. Retrieval is the same cosine top-k used by RAG.
- **MemP** (Fang et al., 2026): A procedural-memory method. On a successful episode it distills the trajectory into a numbered, high-level procedure that states the sequence of operations that worked, and stores that procedure for later retrieval. The stored text is this distilled procedure. On a failed episode it does not add a new procedure; it instead lowers the reliability of the procedures that were retrieved for that episode. The resulting store is updated by this build–retrieve–update cycle and is queried by cosine similarity.
- **MemRL** (Zhang et al., 2026): An episodic-memory baseline that keeps the backbone frozen and ranks stored notes with a two-phase score. The first phase retrieves candidates by semantic similarity; the second re-ranks them with a non-parametric Q-value estimated from past task outcomes. The memory topology is held fixed: MemRL updates the utility used at retrieval, and it does not rewrite, merge, or restructure the stored notes. We use the same frozen-backbone protocol as the other baselines.
- **Fixed-Strategy Evolution**: This variant is an ablation of EvoDisco. Each variant clamps the operator to one of ABSTRACT, REFINE, BRIDGE, or REFRAME and applies that operator at every evolution step. The candidate and the instantiation parameters are still sampled, but the policy receives no REINFORCE update, so the choice of operator does not change with experience. These four runs are reported in Section 4.3 and Appendix A.13.

A.4 IMPLEMENTATION DETAILS

All agents use **GPT-5-mini** with temperature 0.0 and `text-embedding-3-large` for embeddings. Memory graphs are maintained in process rather than in an external graph database. The strategy policy π is a two-layer MLP with hidden dimension 128, optimized using REINFORCE-style updates with a learning rate of 1×10^{-3}.

Immediate utility weights the online performance proxy $\widehat{\Delta F} = \Delta Q$ and the structural-complexity change ΔC by $\alpha = 1.0$ and $\beta = 0.3$. Strategic efficacy uses $\gamma = 1.0$, $\delta = 1.0$, and $\eta = 0.3$, with a history window of $K = 20$ and a policy-update interval of $N = 5$. Within the evolution window, a hybrid trigger initiates evolution every $T = 10$ episodes or when the moving average of the last 10 episode scores decreases by more than $\epsilon = 0.15$. Neither trigger permits evolution beyond the first 80% of the online stream.

At each evolution attempt, the policy samples an operator, one executor-built candidate, and the instantiation parameters under π. An attempt is counted as an applied operation only when that choice commits a graph edit. Retrieval, memory-writing formats, and domain routing follow the per-benchmark configuration files released with the code. The shared hyperparameter setting is in Table 5, and the trigger is specified further in Appendix A.9.

A.5 MEMORY ATTRIBUTES AND STATE ENCODING

Section 3.1 specifies node attributes, the relation vocabulary, and the four feature groups in s_i. The lengths of the recent-outcome windows, feature normalization, and the encoding of variable-size deficiency summaries are left as implementation conventions.

A.6 FEEDBACK DEFINITIONS

Online performance proxy. Equation 6 defines Q. The implementation retains the most recent 32 query embeddings observed before event i and scores the last 12. Spreading activation returns node scores $a_v(\mathcal{G}, \mathbf{q})$. No later episode, held-out task, or task score y_j enters ΔQ_i.

Structural complexity. Equation 7 defines $C(\mathcal{G})$. The vector $\mathbf{c}(\mathcal{G})$ uses the configured node-count, edge-density, and path-length statistics, and $\boldsymbol{\lambda}$ gives their relative weights.

Table 4: Two timescales of evolution feedback. Graph modification is performed by operator execution.

Level	Signal	Learning role	Frequency
Inner	U_i	Feedback and supervision	Each attempt
Outer	E_i	Strategy-policy update	Every N attempts

Pre-execution utility prediction. The forecast $\hat{U}_i = q_\psi(s_i, a_i)$ is stored before execution, as used in Eq. 10. Stored predictions are not recomputed after predictor updates, so each prediction error uses the forecast available before its outcome.

Scoring window. Equations 9–11 are evaluated on $\mathcal{W}_i = \{\max(1, i-K+1), \ldots, i\}$. The operator mean $\bar{U}_i(o)$ averages the most recent $h = 8$ executions of o, or every execution when fewer than h are available.

A.7 Learning and Execution Details

Policy decisions. Equation 12 is the sampling distribution, and Eq. 13 is the log-probability substituted into Eqs. 14 and 15. Section 3.2 states which heads each operator reads; Section A.8 records the numeric cutoffs. Full runs draw o from the four-operator library, with no clamped operator and no benchmark-specific operator prior.

Scoring versus update batches. The scoring window $\mathcal{W}_i$ contains the most recent K attempts, whereas $\mathcal{B}_i$ contains new attempts since the preceding policy update. Only $\mathcal{B}_i$ supplies action log-probabilities in Eq. equation 14. An older event can contribute to the aggregate score without being treated as a fresh on-policy sample.

Utility prediction and positive replay. The predictor is trained on recorded (s_i, a_i, U_i) tuples. State–action pairs enter $\mathcal{D}_+$ only when $U_i > 0$, and scheduled updates minimize Eq. 15.

Two feedback timescales. Immediate utility updates the records used for subsequent decisions after each attempt. Strategic efficacy supplies an aggregate policy-learning signal every N attempts. Neither utility calculation directly changes the graph; graph edits are performed by operator execution.

Attempts and committed edits. An evolution event is a triggered attempt. Attempts that leave the graph unchanged remain in the event history and count toward the policy-update interval. Operator-usage statistics separately report committed graph edits, as described in Appendix B.2. No utility-based rollback is performed.

Overall procedure. Algorithm 1 gives the online loop. Evolution is disabled after the first $\lfloor \rho n \rfloor$ episodes, where n is the online stream size and $\rho = 0.8$. Ordinary writes continue until the stream ends, after which the graph is frozen for held-out evaluation.

A.8 Policy-Controlled Parameters

Section 3.2 gives the candidate-construction rules and the supports of the five instantiation heads. The executor does not draw another target after c is sampled. ABSTRACT retrieves the k-hop neighborhood of the chosen seed and prunes a supporting node when $p > 0$, its similarity to the new concept is at least $0.70 + 0.25p$, and it fails the fixed retention checks on retrieval count and degree. BRIDGE has no separate threshold head: it rejects the chosen pair when the embedding similarity is below $0.35 + 0.3p$. The distance cutoff in Section 3.2 is a minimum: candidate pairs stand at graph distance at least 2. That cutoff is a construction rule, not a policy output.

REFRAME keeps every node's text, writes no node, and rewrites only `causes` and `precedes` edges along one existing path. The sampled frame replaces any frame echoed by the model. The executor reads `REORIENT` and ignores `SUMMARY`. On each consecutive pair (a, b) the reverse

Algorithm 1 EvoDisco: Strategy Learning for Memory Evolution

Require: Online stream of n episodes; initial graph $\mathcal{G}_0$; policy π_ϕ; predictor q_ψ; operator library $\mathcal{O}$; scoring window K; update interval N

$\mathcal{G} \leftarrow \mathcal{G}_0$, $i \leftarrow 0$
$\mathcal{H} \leftarrow [\,]$, $\mathcal{B} \leftarrow [\,]$, $\mathcal{D}_+ \leftarrow \emptyset$
for each online task episode j **do**
  Run the agent with $\mathcal{G}$; observe y_j and interaction history
  Write the new experience into $\mathcal{G}$
  if $j \leq \lfloor \rho n \rfloor$ and trigger is met **then**
    $i \leftarrow i + 1$
    $s_i \leftarrow \mathrm{encode_state}(\mathcal{G}, \mathcal{H}, y_{1:j})$
    Sample $a_i = (o_i, c_i, \theta_i) \sim \pi_\phi(\cdot \mid s_i)$
    $\hat{U}_i \leftarrow q_\psi(s_i, a_i)$
    $\mathcal{G}^+ \leftarrow \mathsf{T}_{o_i}(\mathcal{G}; c_i, \theta_i)$
    if the operator commits an edit **then**
      Compute U_i using Eq. equation 4
    else
      $U_i \leftarrow -0.05$
    end if
    $\mathcal{G} \leftarrow \mathcal{G}^+$ ▷ No utility-based rollback
    Append $(s_i, a_i, \hat{U}_i, U_i)$ to $\mathcal{H}$ and $\mathcal{B}$
    if $U_i > 0$ **then**
      Add (s_i, a_i) to $\mathcal{D}_+$
    end if
    Update q_ψ using recorded utility targets
    if $i \bmod N = 0$ **then**
      Compute E_i on the last $\min(K, i)$ attempts using Eq. equation 8
      Update ϕ using Eq. equation 14
      Apply scheduled replay updates using Eq. equation 15
      $\mathcal{B} \leftarrow [\,]$
    end if
  end if
end for
Freeze $\mathcal{G}$ for held-out evaluation

branch applies when `REORIENT` is `reverse` or the frame is `effect_cause`. That branch moves an existing `precedes` or `causes` edge from a to b onto b to a, keeping weight $\max(w, 0.4)$, and then writes `causes` from b to a. Otherwise `causal_to_temporal` writes `precedes` from a to b, and `temporal_to_causal` writes `causes` from a to b. `relabel_causes` and `relabel_precedes` both take this second branch. A missing edge is inserted at weight 0.75. An edge that is already present, including one just reversed, keeps its weight and adds 0.15×0.75. After that write, every `causes` or `precedes` edge on the pair, in either direction, has its weight increased by 0.1, clipped to $[0, 1]$, and its recency set to the current task index.

Fixed by rule or by the LLM. The frozen LLM writes the ABSTRACT concept, the REFINE children, the BRIDGE relation type, and the REFRAME reorientation. The endpoint pair inside a bridge candidate is the closest embedding pair. For REFINE, the relation label follows from the sampled dimension once θ is chosen. For REFRAME, the edit follows from the sampled frame together with whether the returned reorientation is `reverse`. The numeric cutoffs above are fixed rules. The prompts are the shared templates in Appendix C.

A.9 EVOLUTION TRIGGER

We use a hybrid evolution trigger. An evolution step is executed when either (i) $T = 10$ new task episodes have elapsed since the last evolution, or (ii) the moving average of the last 10 episode scores drops by more than $\epsilon = 0.15$. Evolution stops after 80% of the online stream; frozen transfer never

writes or evolves. This design prevents both overly frequent reorganization and delayed response to memory degradation.

A.10 HYPERPARAMETERS

Table 5 lists the setting used for all four benchmarks. We chose these values by repeated trials on a development split held out from the reported evaluation, then froze them for the online and frozen-transfer runs. All four benchmarks share this setting. Primary terms use unit weight ($\alpha = \gamma = \delta = 1$); $\beta = \eta = 0.3$ keep the structural-complexity penalty and the diversity term on a smaller scale.

Table 5: Hyperparameters used in the reported EvoDisco experiments.

Symbol / Name	Value	Description
α	1.0	Immediate-utility weight on the online performance proxy $\widehat{\Delta F} = \Delta Q$
β	0.3	Immediate-utility weight on structural-complexity change ΔC
γ	1.0	Strategic-efficacy weight on $\widehat{\mathrm{Regret}}$
δ	1.0	Strategic-efficacy weight on $\mathrm{PredErr}$
η	0.3	Strategic-efficacy weight on operator diversity
K	20	Strategic-efficacy window size
N	5	Policy-gradient interval (evolution attempts)
T	10	Episodes between evolution triggers
ϵ	0.15	Drop in the 10-episode moving-average score that fires evolution
Score window	10	Episodes in the moving average for the drop trigger
Cutoff	0.8	Fraction of online stream after which evolution stops
Hidden dim.	128	MLP policy hidden size
Learning rate	1×10^{-3}	REINFORCE step size
Temperature	0.0	Backbone sampling temperature
Embedding	`text-embedding-3-large`	Text encoder
Seed	42	Split shuffle / runner seed

A.11 SPLITS AND BUDGETS

Table 6 lists the online-stream size, the frozen-transfer split size, and the per-episode step budget.

Table 6: Split sizes and per-episode step budgets.

Benchmark	Online n	Transfer n	Max steps
ALFWorld	3553	140	50
WebShop	500	200	15
LifelongAgentBench-OS	350	150	15
LifelongAgentBench-DB	361	139	15

A.12 COMPONENT ABLATION EXECUTION

The rows *w/o Outer Fitness*, *w/o Inner Fitness*, and *w/o Cognitive Operators* in Table 2 and Table 7 each remove one component of the evolution step. The online-then-frozen protocol, splits, backbone, step budgets, retrieval, ordinary memory writes, hybrid trigger, and 80% evolution cutoff are those of the Full run. An attempt is applied only when the chosen edit commits a graph change.

Without outer fitness. Outer fitness is the strategic-efficacy signal E_i (Section 3.3.2) that trains the strategy policy. This variant keeps the cognitive operators and immediate utility, and disables policy selection and policy learning. At each evolution attempt the executor builds the same per-operator candidate sets as Full. It then draws an operator uniformly among those with a nonempty candidate set, draws one candidate uniformly from that set, and draws the instantiation uniformly from the grids in Section 3.2: radius $k \in \{1, 2, 3\}$, child count $m \in \{2, 3, 4\}$, pruning threshold, decomposition dimension, and logical frame. The drawn operator is executed with the same

generator used by Full. Immediate utility follows the committed-edit rule in Eq. equation 4, and a non-committed attempt stores $U_i = -0.05$. The MLP is not queried, and the REINFORCE update that would apply E_i every N events is skipped, so later attempts remain uniform over the valid candidates.

Without inner fitness. Inner fitness is the immediate utility U_i (Section 3.3.1). Candidate construction, policy sampling, cognitive-operator execution, and the outer-loop update every $N = 5$ events match Full. The retrieval-quality change and the complexity penalty are omitted from the recorded outcome: every event stores $U_i = 0$, whether or not the operator commits an edit. The utility predictor is supervised with these zero targets. The replay buffer, which admits only events with $U_i > 0$, receives no events. Strategic efficacy is still evaluated on the resulting history, so $\widehat{\mathrm{Regret}}_i$ and $\mathrm{PredErr}_i$ are computed from zero utilities while Div_i remains the operator entropy of the window. The policy therefore continues to update, and the update no longer depends on retrieval quality or structural complexity.

Without cognitive operators. This variant keeps the hybrid trigger and replaces the library $\{$ABSTRACT, REFINE, BRIDGE, REFRAME$\}$ with three structural edits. No candidate set is built, the policy is not queried, and the language model is not called to author new node text. Each attempt draws one edit uniformly.

- **Add edge.** Two distinct nodes are drawn uniformly. A relation is drawn uniformly from `analogous-to`, `causes`, `precedes`, and `is-a`, and inserted with weight 0.5.
- **Delete edge.** One existing relation is drawn uniformly and removed. When the graph contains no relations, the attempt commits no edit.
- **Merge.** Two distinct nodes are drawn uniformly. The retained node's text is the concatenation of the two contents, truncated at 2000 characters. Its embedding is the ℓ_2-normalized sum of the two embeddings, and its confidence is their arithmetic mean. Relations incident on the discarded node are rewritten onto the retained node, after which the discarded node is removed.

Immediate utility is computed on the edited graph and stored. The policy receives no update, so the next edit is again a uniform draw over the three structural edits. A graph with fewer than two nodes commits no edit.

The fixed-operator rows are a separate intervention. Each clamps *o* to one named cognitive operator. The initial MLP still samples that operator's candidate and the instantiation heads in Section 3.2, and those weights receive no REINFORCE update. These rows do not use the structural edits above.

A.13 OS ABLATIONS

Table 7 reports the OS analogue of Table 2. EvoDisco (Full) is the same system as Table 1 (74.0 online / 77.3 transfer).

Table 7: OS ablations (%). Deltas are relative to Full (Table 1). Best in each column is bold.

Variant	Online	Xfer	ΔXfer
EvoDisco (Full)	**74.0**	**77.3**	—
w/o Outer Fitness	71.1	75.3	-2.0
w/o Inner Fitness	69.1	74.0	-3.3
w/o Cognitive Operators	71.7	74.0	-3.3
Fixed Abstract	72.0	74.0	-3.3
Fixed Refine	72.9	75.3	-2.0
Fixed Bridge	71.7	76.0	-1.3
Fixed Reframe	71.1	74.0	-3.3

Removing outer fitness costs 2.0 transfer points; removing inner fitness is the largest drop (-3.3 transfer and -4.9 online). w/o Cognitive Operators matches that transfer drop. No fixed operator beats Full on either metric; Fixed REFINE is the strongest fixed variant on online SR (72.9%), and Fixed BRIDGE is closest on frozen transfer (76.0%, -1.3).

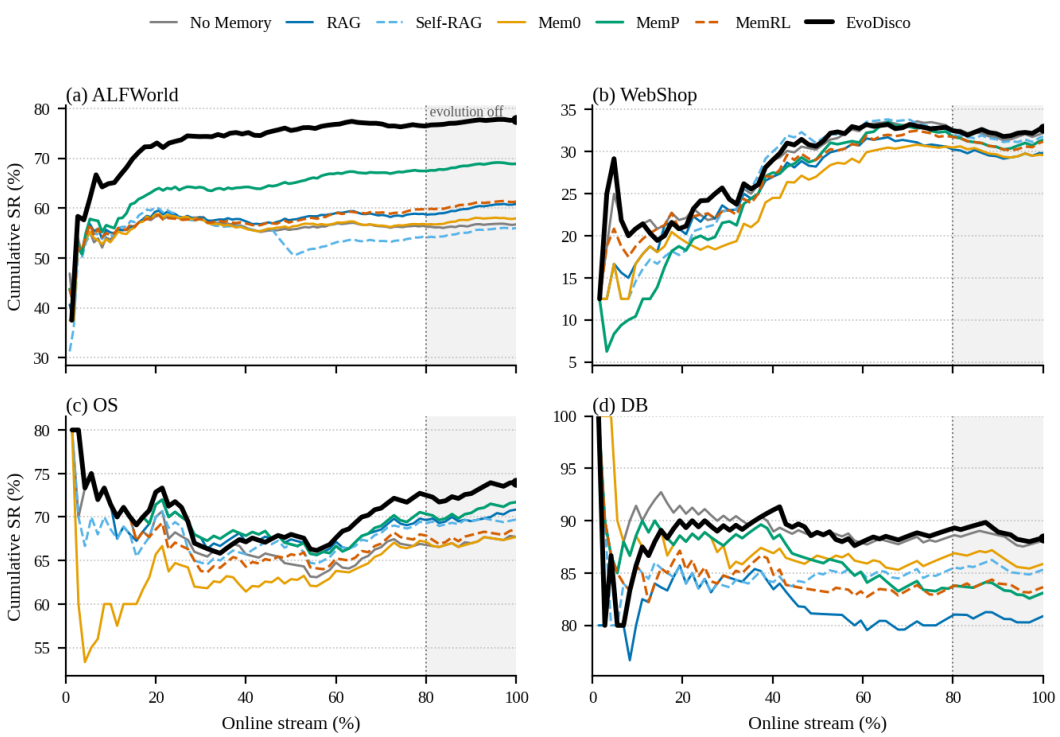


Figure 2: Cumulative online success rate over the training stream. Each point is the success rate on all episodes observed so far. The shaded interval is the final 20%, during which memory evolution is disabled and ordinary memory writes continue. EvoDisco separates from every baseline on ALFWorld and finishes highest on all four benchmarks. On WebShop and DB the curves overlap for much of the stream, and the final margins are small.

B Additional Experimental Analysis

B.1 Late-Stream Online Success Rates

Table 8 reports success rates over the final 20% of the online stream. Memory evolution is disabled during this interval, while ordinary memory writes remain enabled. Figure 2 plots cumulative online SR over the full stream. The right-hand edge of each curve is the full-stream online SR in Table 1; the shaded suffix is the interval summarized by Table 8.

Table 8: Success rates (%) over the final 20% of the online stream. Memory evolution is disabled during this interval, while ordinary memory writes remain enabled. Bold denotes the highest value in each column, including ties.

Method	ALFWorld	WebShop	OS	DB
No Memory	58.6	31.0	71.4	**86.1**
RAG	69.1	28.0	75.7	80.6
Self-RAG	63.4	31.0	71.4	84.7
Mem0	62.7	26.0	70.0	81.9
MemP	74.7	30.0	77.1	80.6
MemRL	67.4	29.0	71.4	83.3
EvoDisco	**82.4**	**34.0**	**80.0**	**86.1**

On ALFWorld, EvoDisco pulls ahead early and remains clearly above MemP for the rest of the stream. On OS, EvoDisco is highest at the evolution cutoff and at the end of training, although

MemP is narrowly ahead near 40% progress (67.9% versus 67.1%). On WebShop and DB the trajectories overlap. EvoDisco still finishes first, by 0.6 and 0.3 points over the next-best online SR.

B.2 Operator Usage Distribution

Figure 3 and Table 9 report operators that commit a graph edit. The logs contain 400 evolution attempts on ALFWorld, all inside the first 80% of the training stream (2842 episodes), and 39 attempts on OS, inside the first 280 of 350 training episodes. Of these, 357 and 32 are applied. The remainder are ABSTRACT attempts that return `max_abstraction` and leave the graph unchanged (43 on ALFWorld, 7 on OS). REFINE, BRIDGE, and REFRAME commit on every logged attempt. Policy-gradient updates run every $N = 5$ attempts, which yields 80 updates on ALFWorld and 7 on OS. Failed attempts stay in the strategic-efficacy window; they are omitted from Figure 3 and Table 9.

Table 9: Applied operator counts. Attempts that do not commit an edit are excluded. Updates are policy-gradient steps, one every $N = 5$ attempts.

Benchmark	Abstract	Refine	Bridge	Reframe	Total	Updates
ALFWorld	7	117	69	164	357	80
OS	3	9	9	11	32	7

The mix is not uniform, and it does not collapse to one operator:

- **ALFWorld.** The first fifth of the evolution window still applies all four operators (6 ABSTRACT, 21 REFINE, 22 BRIDGE, 20 REFRAME). ABSTRACT then drops out: 6 of its 7 committed edits fall in that first bin, and none are committed after 40% of the window, because later attempts hit the abstraction cap. REFRAME rises from 20 to 43 commits per bin. REFINE stays between 19 and 30. BRIDGE falls from 22 to 4 and then returns to 10 in the last bin. The window totals are 164 REFRAME, 117 REFINE, 69 BRIDGE, and 7 ABSTRACT.
- **OS.** The 32 commits are 11 REFRAME, 9 REFINE, 9 BRIDGE, and 3 ABSTRACT. ABSTRACT is selected 10 times and commits 3 times. This is a short stream, so the four counts should not be read as a stable ranking. They do not match the fixed-operator order in Table 7, where REFINE is strongest online and BRIDGE is closest on transfer.

B.3 Strategic Efficacy Over Time

Figure 4 plots the three windowed terms in Eq. equation 8 on ALFWorld, including ABSTRACT attempts that do not commit. The window is $K = 20$. Event i is drawn only when $m_i = K$, so the curves start at event 20 and each point averages 20 attempts. The summary below compares events 20–29 with the last ten attempts.

On ALFWorld (400 attempts), $\mathrm{PredErr}$ falls from 0.063 to 0.030. $\widehat{\mathrm{Regret}}$ is 0.014 on events 20–29 and 0.013 on the last ten attempts; over events 200–400 its mean is 0.010. Div starts at 1.36, close to $\log 4 \approx 1.39$, and ends at 1.08. The drop lines up with Figure 3a: ABSTRACT stops committing, and the window moves from a four-operator mix toward REFRAME and REFINE. Entropy 1.08 is still above a single-operator window, which would be 0. The combined signal E_i stays positive, moving from 0.33 to 0.28.

On OS there are 39 attempts and 7 updates, and 20 attempts have a full window. The same comparison gives $\mathrm{PredErr}$ 0.055 to 0.036, $\widehat{\mathrm{Regret}}$ 0.016 to 0.021, and Div 1.36 to 1.33. With only twenty full-window attempts, OS is reported here as these means. E_i stays near 0.34.

The curves stop at the ALFWorld evolution cutoff. They do not cover the final 20% of the online stream, where evolution is disabled. Table 8 is that later interval on the main evaluation runs. It is not a continuation of Figure 4.

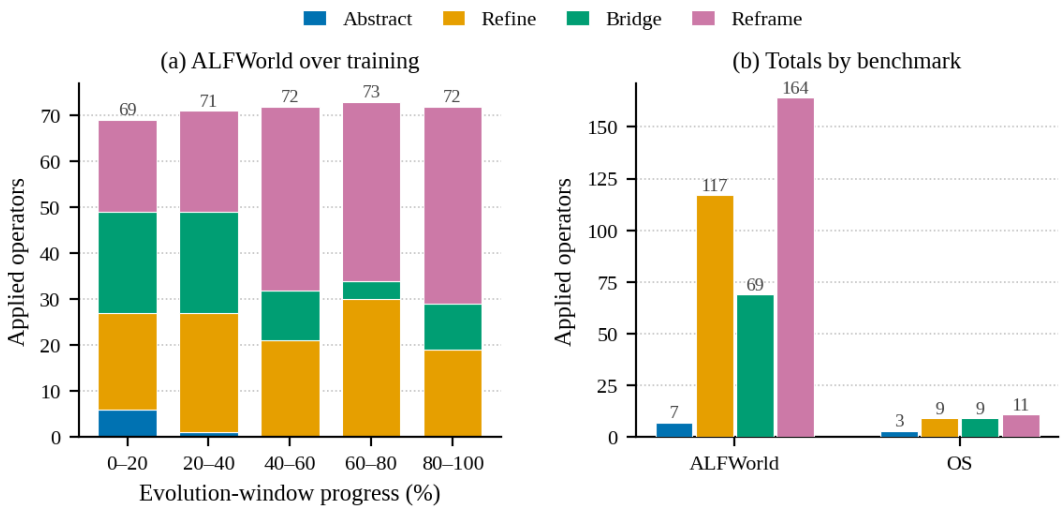


Figure 3: Applied operators. (a) ALFWorld, five equal bins of the evolution window (first 2842 episodes). (b) Totals on ALFWorld (357 commits) and OS (32 commits). ABSTRACT is confined to the early ALFWorld bins. REFRAME is the most frequent commit on both benchmarks.

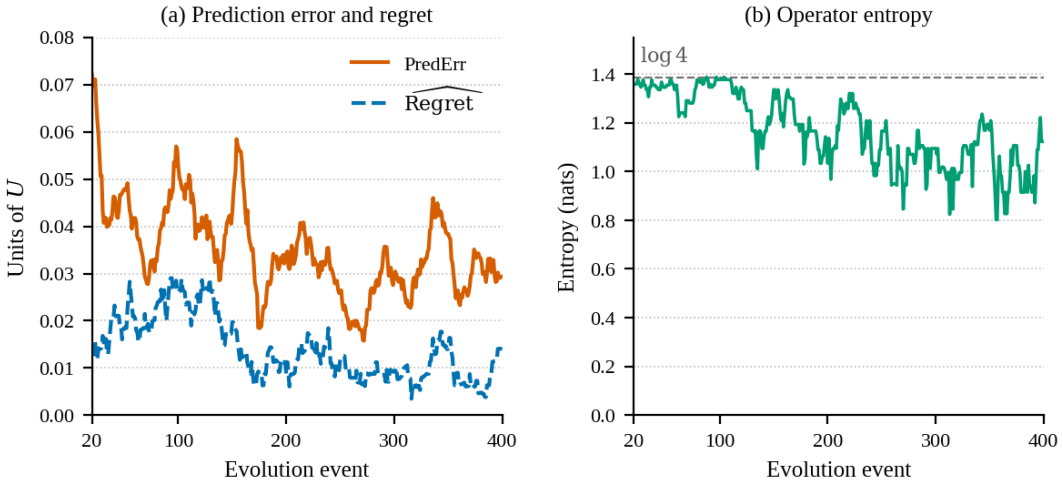


Figure 4: ALFWorld strategic-efficacy terms once the window contains $K = 20$ attempts (events 20–400), including ABSTRACT attempts that do not commit. (a) PredErr and $\widehat{\text{Regret}}$ on the utility scale (Eqs. equation 10 and equation 9). (b) Operator entropy Div in nats (Eq. equation 11). The gray line is $\log 4$, the maximum over four operators. A one-operator window has entropy 0.

B.4 QUALITATIVE EDITS

Figure 5 traces two applied edits from the runs in Table 1. For each edit we record the graph fragment before the change, the operator and the arguments that were applied, the nodes written into the graph, and a later episode that retrieved one of those nodes.

ALFWorld. At stream position 960 the applied operator is ABSTRACT, with neighborhood radius k=1 and pruning threshold 0. The seed is a successful pick-and-place procedure for one concrete scene: go to shelf 1, take statue 1, go to coffeetable 1, and move the statue onto the table. Its 1-hop relation neighborhood contains 13 notes. The edit writes a single `META_STRATEGY` and links it to six of those notes. No note is removed.

Five of the six linked notes remain in the frozen graph. They are separate successful procedures: a newspaper from an armchair to an ottoman, a watch to a coffeetable, a handtowel from a holder, a statue to a coffeetable, and two books to a dining table. The new strategy keeps the statue-to-coffeetable goal. Its procedure is the placement loop shared by that neighborhood. Search shelves, tables, cabinets, and drawers; open a closed source before taking; use an object index when several objects share a name; confirm that the statue is in hand; clear anything blocking the coffeetable; issue the place command only while standing at the table; then inspect the table and repeat the placement if the statue is absent. The seed procedure names one shelf, one statue index, and one table. The meta-strategy states the closed-container, index, blocker, and verification checks in the same procedure.

That meta-strategy is the only retrieved note on five later statue-to-coffeetable episodes, at stream positions 1347, 1387, 1539, 1545, and 1579. All five succeed, in 11, 8, 4, 6, and 9 steps. On the episode at position 1579, retrieval returns only that strategy. No Memory, RAG, Self-RAG, Mem0, and MemP each reach the 50-step limit on the same game without success.

OS. At stream position 160 the applied operator is REFINE, with a categorical decomposition into $m{=}4$ children. The target is one successful procedure tagged `perms`, whose command list is `find`, `sort`, `mkdir`, `cp`, `cat`, `chmod`, and `echo`. The note performs five roles in sequence. It creates `/var/log/app/backup`. It lists non-recursive `.log` files under `/var/log/app/errors` and `/var/log/app/debug` in sorted, null-separated order. It copies each file into the backup directory with `cp --preserve=mode,timestamps --backup=numbered`. It concatenates that same list into `/var/log/app/full_report.log`. It sets the report to mode 644. The parent note is kept.

The four children split those roles and receive their own skill tags. The first, tagged `files`, only runs `mkdir -p` on the backup directory. The second, tagged `text`, only emits the sorted `find` list. The third, tagged `files`, copies the listed logs, including the numbered-backup `cp`. The fourth, tagged `perms`, concatenates the list into the report and applies `chmod 644`. Each child is tied to the parent by an `is-a` edge from the parent and a `generalizes` edge from the child.

At stream position 170 the instruction is to copy every `.log` file from `/var/log` into `/backup`, preserving directory structure, and to write a text file that lists each copied name and size. Retrieval returns three notes. The copy child has activation 0.611, the original parent has activation 0.614, and a separate procedure for copying `.txt` files from `/data` into `/backup` has activation 0.509. The listing child and the concatenate-and-chmod child are absent. The episode succeeds in 3 steps. On the same instruction, No Memory fails in 4 steps, RAG in 3, Self-RAG in 5, Mem0 in 4, and MemP in 2.

The ALFWorld episode receives one placement strategy. The OS episode receives the copy step of a note that also listed files, built a report, and changed permissions. In both cases the paired baseline failures are outcomes on the same task. They do not isolate the edit from the other contents of the graph.

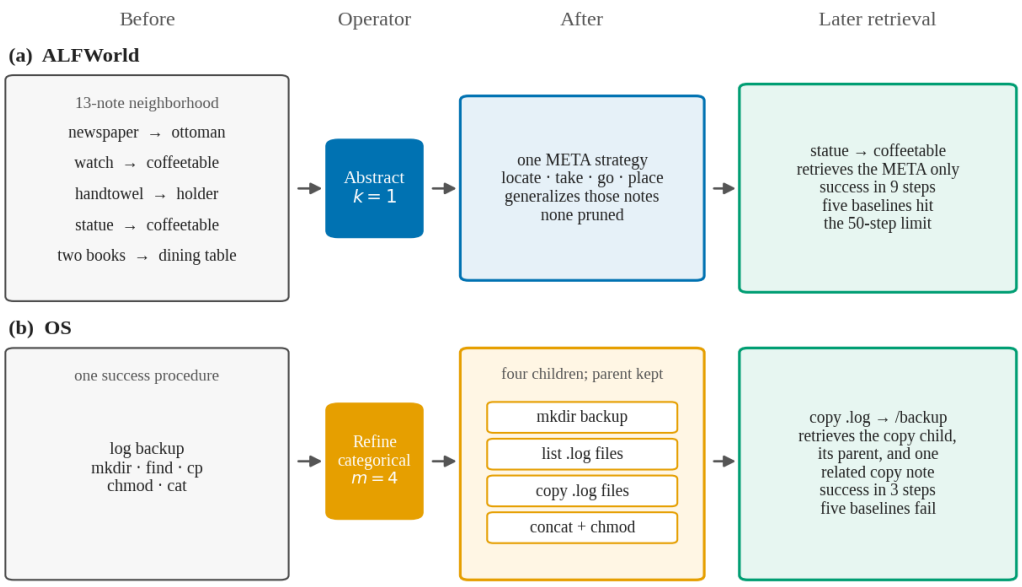


Figure 5: Two applied edits from the Table 1 runs, read left to right: the graph fragment before the edit, the operator that was applied, the structure written into the graph, and a later retrieval. (a) ALFWorld. ABSTRACT (k=1) links a meta-strategy to a 13-note neighborhood and prunes nothing. Five of the six linked notes remain in the frozen graph and are shown. At stream position 1579, a new statue-to-coffeetable episode retrieves only that strategy and succeeds in 9 steps. The same game reaches the 50-step limit under No Memory, RAG, Self-RAG, Mem0, and MemP. (b) OS. REFINE (categorical, m=4) splits one log-backup procedure into four children and keeps the parent. At stream position 170, the instruction to copy `.log` files from `/var/log` into `/backup` retrieves the copy child, that parent, and one other copy note, and succeeds in 3 steps. The same five systems fail that instruction.

C PROMPT TEMPLATES

Relation extraction, ABSTRACT, BRIDGE, and REFRAME use the templates below on ALFWorld, WebShop, OS, and DB. REFINE uses the template below on ALFWorld, WebShop, and DB. The OS system in Table 1 asks for at most m source-grounded shell subprocedures and reads them as a JSON array of `intent` and `procedure` fields. The agent instruction and the episode-note writer are shown for ALFWorld. WebShop, OS, and DB use the same two calls, with that benchmark's action syntax and note instructions. Each system message fixes the role of the call and the fields the implementation reads. Each user message supplies the current episode, the retrieved notes, or the graph fragment chosen by the executor. Text in braces is a placeholder filled at runtime. A note-producing call uses the five fields `TYPE`, `TASK_TYPE`, `ROOT_CAUSE`, `LESSON`, and `PROCEDURE`.

C.1 AGENT STEP

At every environment step the agent receives the task, the notes retrieved for that task, and the interaction so far, and returns one next action. On the first two steps the user message also includes one instructional example for the current task type. The box below is the ALFWorld instruction. WebShop replaces the action list with `search` and `click`. OS returns a bash block under `Act: bash` or `Act: finish`. DB returns one SQL statement under `Action: Operation`, or a final answer under `Action: Answer`.

Agent Step (ALFWorld)

System. You solve a household task one action at a time. The environment describes the current scene and the goal, then returns an observation after each action. When the observation is "Nothing happened", the previous action was invalid. Place an object with `move {obj} to {recep}`. The legal actions are:

1. `go to {recep}`
2. `take {obj} from {recep}`
3. `move {obj} to {recep}`
4. `open {recep}`
5. `close {recep}`
6. `toggle {obj} {recep}`
7. `clean {obj} with {recep}`
8. `heat {obj} with {recep}`
9. `cool {obj} with {recep}`

Fill both fields below.

1. `Thought`: the current state and the plan for the next action.
2. `Action`: one legal action.

User.
`Primary Goal:`
{task}
`Instructional Example:`
{example}
`Archived Memories:`
{memories}
`Current Task Progress:`
{history}
Now, it's your turn.

C.2 EPISODE NOTE

After an episode, the writer turns the task, the outcome, and the trajectory into one note, which is stored as a node. The box below is the ALFWorld writer. WebShop, OS, and DB keep these five

fields and this user message. Their system messages name shopping, shell, or SQL categories, and ask for `search`/`click` steps, concrete shell commands, or a SQL query shape.

Episode Note (ALFWorld)

System. You write one reusable memory note from a completed household episode. Fill every field below.

1. `TYPE`: `SUCCESS_PROCEDURE`, `FAILURE_REFLECTION`, or `META_STRATEGY`.
2. `TASK_TYPE`: a household category, such as `pick_and_place`, `heat`, or `clean`.
3. `ROOT_CAUSE`: why the episode failed, or `N/A` when it succeeded.
4. `LESSON`: one lesson that could apply to a later episode.
5. `PROCEDURE`: numbered high-level steps. Keep them environment-agnostic where possible.

User.
`Task:` {task}
`Outcome:` {SUCCESS or FAILURE}
`Trajectory:`
{trajectory}
Write a compact reusable memory note.

C.3 Relation Extraction

When a new note is written, a second call proposes typed edges to notes already in the graph. The system message fixes the relation vocabulary and the block format. The user message is the new note and a short list of candidate notes. An empty answer is a valid result when no relation is supported.

Relation Extraction

System. You propose typed edges between a new memory note and existing candidate notes. Prefer `causes`, `precedes`, or `is-a`. Use `analogous-to` only when two notes are variants of the same procedure, and then set the weight at most 0.4. Emit at most two blocks. Leave the answer empty when every candidate link would be a weak guess. Each block uses the fields below. Use only identifiers that appear in the user message, and connect `NEW` to a candidate.

1. `REL`: `causes`, `precedes`, `is-a`, or `analogous-to`.
2. `SRC`: `NEW` or a candidate identifier.
3. `DST`: `NEW` or a candidate identifier.
4. `W`: a weight in $[0.3, 1.0]$.

User.
`NEW:`
{new_note}
`CANDIDATES:`
{candidates}
Extract relations.

C.4 Abstract

Abstract receives the seed neighborhood selected by the policy and writes one higher-level concept. The new node is stored as a `META_STRATEGY` and linked to the neighborhood. The system message asks for a single shared pattern; the user message is the seed hint and the neighborhood notes.

Abstract

System. You compress several related memory notes into one higher-level concept. Fill every field below.

1. `TYPE`: `META_STRATEGY`.
2. `TASK_TYPE`: a short name for the shared pattern.
3. `ROOT_CAUSE`: `N/A`.
4. `LESSON`: the theme common to the neighborhood. Drop episode-specific names unless they are hard constraints.
5. `PROCEDURE`: the procedure shared by those notes.

User.
`Seed:` {seed}
`Neighborhood notes:`
{notes}
Write one abstractive concept node.

C.5 REFINE

REFINE splits one overloaded note into m children along the sampled dimension, which is temporal, causal, or categorical. Each child is a separate note. The system message fixes the split; the user message is the source note together with m and the dimension.

Refine

System. You decompose one overloaded memory note into finer-grained child notes. Return exactly the requested number of children. Split along the requested dimension: temporal, causal, or categorical. Keep the children complementary. Each child uses the same five fields as a memory note, under the header `CHILD_k`.

1. `CHILD_1`: `TYPE`, `TASK_TYPE`, `ROOT_CAUSE`, `LESSON`, `PROCEDURE`.
2. `CHILD_2`: the same fields.
3. Continue through `CHILD_m`.

User.
`Decomposition dimension:` {dimension}
`Number of children:` {m}
`Overloaded node:`
{note}
Emit `CHILD_1` through `CHILD_m`.

C.6 BRIDGE

BRIDGE receives two weakly connected regions and returns one relation type together with a short note that states how the two regions are used together. The system message restricts the relation to the fixed vocabulary; the user message is the two clusters.

Bridge

System. You connect two memory regions that are semantically related but weakly linked. Prefer `causes`, `precedes`, or `is-a`. Use `analogous-to` when the two regions are variants of the same procedure. Fill every field below.

1. `RELATION`: `causes`, `precedes`, `is-a`, or `analogous-to`.
2. `JUSTIFICATION`: one sentence.

3. BRIDGE_NOTE: a note with TYPE META_STRATEGY, a short TASK_TYPE, ROOT_CAUSE N/A, a LESSON stating the link between the two regions, and a PROCEDURE stating how to reuse them together.

User.
Cluster A:
{cluster_a}
Cluster B:
{cluster_b}
Infer the missing relation and write a short bridging note.

C.7 REFRAME

REFRAME receives an existing path and the logical frame sampled by the policy. The user message is the path, in its current order, and that frame. The system message asks for FRAME, REORIENT, and SUMMARY. Section A.8 states which of these fields the executor applies: the sampled frame overrides FRAME, REORIENT selects the edge edit, and SUMMARY is not written into the graph. Node text is left unchanged. A reverse reorientation, or the frame effect_cause, reverses causes and precedes edges on the path. The box below is the prompt sent to the model. Its instruction to keep the underlying facts constrains that generation; the graph edit is the edge rule in Section A.8.

Reframe

System. You reorganize the logical frame of a reasoning chain. Keep the underlying facts and change the retrieval-oriented order. Fill every field below.

1. FRAME: temporal_to_causal, causal_to_temporal, or effect_cause.
2. REORIENT: reverse, relabel_causes, or relabel_precedes.
3. SUMMARY: a note with TYPE SUCCESS_PROCEDURE or META_STRATEGY, a short TASK_TYPE, ROOT_CAUSE N/A, a LESSON stating what the new frame emphasizes, and a PROCEDURE giving the same steps in the new order.

User.
Target logical frame: {frame}
Reasoning-chain notes, in the current order:
{notes}
Reorganize the logical skeleton.