MCMEM: METACOGNITIVE MEMORY FOR SELF-AWARE RELIANCE IN LONG-HORIZON LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The memory mechanism has long been recognized as a reusable knowledge reservoir that supports large language model (LLM) agents to self-evolve in open-ended environments. Nevertheless, the universal utility of a piece of memory across task environments is hardly determinable at the time of abstraction. In this paper, we argue that an ideal memory mechanism should also enable an agent to progressively recognize its own capability boundaries when augmented with accumulated memories, so that it could strike a balance between exploiting existing ones and exploring more in open-ended environments. Based on this insight, we propose a MetaCognitive Agentic Memory mechanism (MCMEM) that augments an agent with adaptable and reliable memory management and utilization throughout self-evolution. It is built on a hierarchical memory graph that progressively abstracts episodic trajectories into reusable memories while preserving provenance links to the underlying evidence. On top of that, a metacognitive layer continuously calibrates the agent's reliance on each memory as new experiential evidence accumulates, enabling sustained utilization of memory over long task horizons. Experiments on long-horizon agent benchmarks validate MCMEM's superiority in terms of long-term task success and exploration-exploitation balance.

1 INTRODUCTION

Large language model (LLM) agents improve by abstracting reusable knowledge from past experience — storing reflections, strategies, and workflows that guide future tasks without parameter updates (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025b; Chen et al., 2024b; Huang et al., 2026; Ouyang et al., 2026). This paradigm carries a hidden fragility: an abstraction's reliability cannot be assessed when it is formed, because its utility depends on the situation it will later meet and the agent that applies it. As the agent faces a continuous stream of tasks under shifting conditions (Wei et al., 2025), a memory that accelerated earlier success can silently mislead later attempts (Liang et al., 2026). Whether self-evolution compounds or corrodes capability (Shao et al., 2026; Yu et al., 2026) therefore hinges on one question: at the moment of use, can the agent tell a helpful memory from a misleading one?

Existing memory methods leave this question open in three ways. First, many works accumulate memories as soon as they are extracted (Wang et al., 2025b; Ouyang et al., 2026; Liang et al., 2026) without estimating whether an abstraction still holds. What later tasks reveal thus never changes how far the agent relies on the memory, hence stale ones are reused like sound ones and hurt most on difficult cases (Hu et al., 2026). Evidence from later tasks are therefore never used to reassess stale memories, which are then reused indiscriminately on different scenarios. Second, methods that actively manage memory treat task outcomes as evidence about the provided memories (Zhao et al., 2024; Zhang et al., 2026), yet agents often disregard retrieved experience during execution (Zhao et al., 2026a), causing outcomes to credit or blame abstractions the agent never actually followed. Self-judging agents further conflate self-consistent trajectories with correct ones (Zhu et al., 2026), so erroneous abstractions persist and keep guiding action (Dixit et al., 2026). Third, most agents inject experience on a fixed schedule (Zhao et al., 2026b) and present well-supported, untested or refuted abstractions alike, so nothing tells the agent how far to lean on any of them.

These shortcomings share a common root: existing memory mechanisms lack a closed loop in which accumulating evidence modulates the agent's reliance on memories and the outcomes of that reliance

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.

are, in turn, incorporated as new evidence. We argue that bridging this gap requires memory to serve not as a passive experience accumulator but as an active substrate through which the agent gauges its own memory-augmented capability and calibrates its reliance accordingly. However, achieving such a memory system faces three major challenges. First, reliability can only be judged after the fact, from what later tasks reveal — yet the abstraction it concerns does not stand still. As experience grows the agent reshapes its stored knowledge, so any reliability estimate needs something stable to anchor to across those changes. Second, whether an abstraction is itself correct cannot be observed directly. The only signal is whether a task succeeded or failed, and that signal conflates multiple causes: the abstraction may have been wrong, or right but not adopted, or adopted but poorly executed — and existing work shows that attributing outcomes to individual causes remains highly unreliable (Zhang et al., 2025b). Third, even a well-calibrated memory reliability estimate does not by itself change behavior. The agent must translate that belief into a concrete decision at every step — how far to lean on the memory versus act on its own — and without such a mechanism, reliance defaults to what is available rather than what is warranted.

To this end, we propose MCMEM, a MetaCognition-powered agentic MEMory mechanism, to improve the agent's long-horizon problem-solving ability. We instantiate MCMEM with a hierarchical graph-structured memory substrate that grounds every stored abstraction in the episodes from which it was derived, and a metacognitive layer that disentangles each memory's evidential support from completed episodes into reliability beliefs, which in turn govern how the agent manages and relies on its memory, promoting a long-term exploration–exploitation balance as the agent accumulates experience. Our contributions are summarized as follows: (1) We propose a hierarchical memory graph that preserves the evidential provenance of every abstraction, enabling the agent to maintain a grounded estimate of its own memory-augmented capability; (2) We develop a metacognitive mechanism under which the agent judges its own experience to form reliability beliefs about stored abstractions, and calibrates its reliance on them accordingly; (3) Our method MCMEM achieves state-of-the-art performance over existing memory methods across long-horizon agent benchmarks and strikes an excellent balance between exploration and memory exploitation.

2 PRELIMINARIES

We consider an LLM agent that solves a sequence of long-horizon tasks $\tau_1, \ldots, \tau_N$ and accumulates experience in a memory, the only information it retains from one task to the next. Each task is attempted in a single episode, within which we model the agent-environment interaction as a partially observable Markov decision process without per-step reward, defined by the tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{A}, \Omega, \mathcal{T}, \mathcal{O} \rangle$. Here, $\mathcal{G}$ is the goal space, $\mathcal{S}$ the hidden state space, $\mathcal{A}$ the action space, Ω the observation space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ the transition dynamics, and $\mathcal{O} : \mathcal{S} \to \Omega$ the observation function. Task τ_i specifies a goal $g \in \mathcal{G}$. At step t, the agent receives an observation $o_t \in \Omega$ and takes an action $a_t \in \mathcal{A}$; the next observation also shows whether the environment accepted a_t or refused it. At the end of the episode, the environment returns a binary outcome $y_i \in \{0, 1\}$, where $y_i = 1$ indicates that the task was completed. The trajectory ξ_i records the observation, the block m_t of memory delivered to the agent, and the agent action at each step. We denote by $\xi^{<t}$ the prefix of ξ_i before step t and by ξ_t^w its last w steps.

Since the LLM parameters are not updated across episodes, the experience an agent accumulates is retained solely in its memory. We next define the memory, the operators that access it, and the resulting agent.

Definition 1 (Agent Memory) *Let M_i denote the agent's memory at the start of task τ_i. It contains records of past episodes and the knowledge abstracted from them.*

Definition 2 (Memory Operators) *The memory is accessed and modified by three operators. i) The **Read Operator** μ is applied at every step and returns the block $m_t = \mu\big(M_i, (g, o_t, \xi_t^w)\big)$ delivered to the agent. ii) The **Write Operator** ϕ is applied at the end of each episode and incorporates the trajectory and its outcome into the memory, $M_i^+ = \phi(M_i, \xi_i, y_i)$. iii) The **Maintenance Operator** ζ is applied after the write and updates the existing contents of the memory, which it may judge again, revise or withdraw, $M_{i+1} = \zeta(M_i^+)$.*

Definition 3 (Memory-augmented LLM Agent) *A memory-augmented LLM agent is a tuple $(f_\theta, \mu, \phi, \zeta)$, where f_θ is a pre-trained LLM with frozen parameters θ. The agent's policy π is*

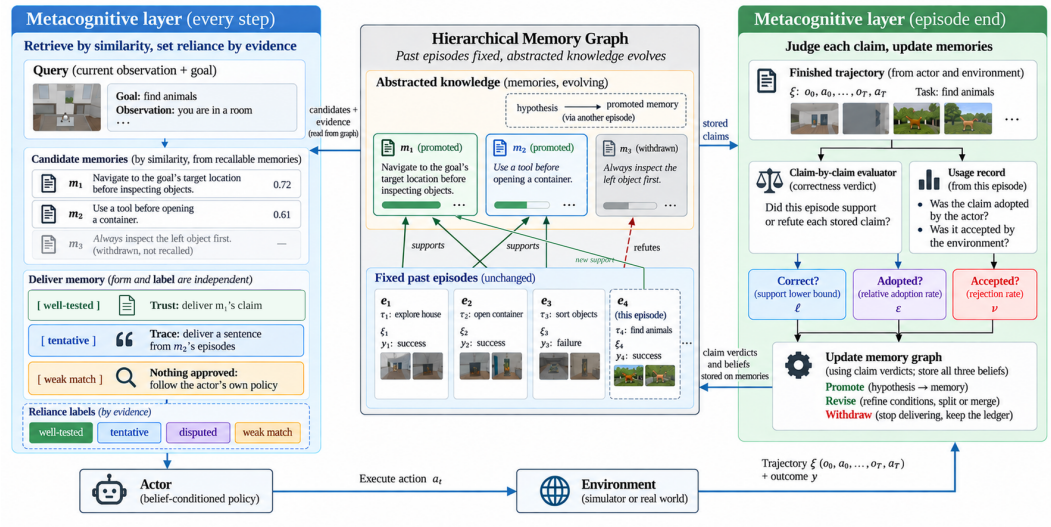

Figure 1: Overview of MCMEM. The hierarchical memory graph (center) maintains fixed episodes linked by provenance edges to evolving, abstracted knowledge. The metacognitive layer operates at two temporal scales: per-step reliance selection (left) that decides how—and whether—to trust a retrieved memory, and per-episode belief updating (right) that promotes, revises, or withdraws memories based on claim-level judgements. The outcome of relying on memory thus becomes the evidence that calibrates future reliance.

obtained by prompting f_θ with the goal, the current observation, the interaction history, and the block read from memory, $a_t \sim \pi(a_t \mid g, o_t, \xi^{<t}, m_t)$. We refer to f_θ acting under π as the actor, and to the component that maintains M_i and implements μ, ϕ, and ζ as the memory mechanism. Because θ is fixed, experience from earlier tasks can influence a_t only through m_t.

3 METHODOLOGY

As shown in Figure 1, MCMEM closes the loop between memory evidence and agent reliance through two coupled modules: i) A hierarchical graph-structured memory substrate stores both memories and the provenance edges linking each memory to the episodes from which it was derived, making content and evidential history jointly available to the metacognitive layer; ii) The metacognitive layer operates upon the substrate at two temporal scales. At boundaries between task episodes, a belief-estimation process consumes the completed episode trace together with the graph's provenance structure, and writes back calibrated reliability beliefs — so that every memory's trustworthiness is updated by the latest evidence. At each step within an episode, a belief-conditioned controller reads relevant memories and their attached beliefs, and translates them into a step-level reliance decision that modulates how far the agent leans on stored knowledge versus its own base policy. We elaborate on each mechanism in turn in the following sections.

3.1 HIERARCHICAL MEMORY GRAPH

Whether a piece of memory holds can be read only off the episodes that bear on it, yet the memory itself is reshaped as experience grows, so any estimate needs an anchor that stays fixed. We therefore organize the memory as a two-level hierarchy joined by provenance edges: the episodes the agent has lived through, which never change, and the knowledge abstracted from them, which may evolve while each piece of evidence about it stays tied to an episode.

Memory structure. When task τ_i begins, the memory is the tuple:

$$M_i = \big(\mathcal{E}_i,\ \mathcal{R}_i,\ \Lambda_i,\ \ldots\big), \tag{1}$$

where the dots stand for parts added by the mechanisms that need them. The episode archive $\mathcal{E}_i$ holds one record per completed episode, in which a model call, the summarizer, marks and narrates

the milestone steps that led to the outcome. The memory archive $\mathcal{R}_i$ holds the memories, knowledge abstracted from those episodes for delivery to the actor. Each memory r carries a current wording, a claim with the conditions under which it applies, and provenance edges to the episodes it was abstracted from.

Evidence ledger and standing. The evidence ledger Λ_i holds one entry per wording and episode judged against it, carrying the verdict assigned at the episode end: whether the episode supported the claim, refuted it, or neither. For a memory r whose current wording carries the entries Λ_r, the *standing* of r is:

$$s_r = \big|\{e : \Lambda_r(e) = \textit{support}\}\big|, \quad q_r = \big|\{e : \Lambda_r(e) = \textit{refute}\}\big|, \quad n_r = s_r + q_r, \tag{2}$$

where $\Lambda_r(e)$ is the verdict episode e left on r and n_r counts the episodes that decided either way. Every decision about whether r is correct reads its standing, which follows any change of wording at once.

3.2 Metacognitive Layer

Building upon this anchor, the metacognitive layer answers the remaining two difficulties: correctness cannot be observed, and a belief changes nothing unless it governs reliance. The belief-estimation process answers the first by reading beliefs from verdicts and usage at each episode end, and belief-driven maintenance acts on them at once, deciding what the memory keeps. The belief-conditioned controller answers the second by deciding at each step how far the actor leans on each memory, so that evidence sets reliance.

3.2.1 Belief Estimation

While the ledger ties every verdict to a fixed episode, an episode yields only its outcome, which conflates three causes: a wrong memory, a right one the actor did not adopt, and an adopted one whose action the environment refused. The belief-estimation process therefore judges each claim against the actions the episode took where the claim applied, rather than against the outcome alone, and keeps one belief per cause.

Verdicts. At each episode end, a model call, the assessor, judges stored claims against semantically similar episodes; for a claim c and an episode e, it reads e's goal, outcome and milestone steps and returns the verdict:

$$v(c,e) = \begin{cases} \textit{support} & e \text{ took } c\text{'s action under } c\text{'s conditions and the predicted result followed,} \\ \textit{refute} & e \text{ took } c\text{'s action under } c\text{'s conditions and the predicted result did not follow,} \\ \textit{set aside} & c\text{'s own conditions never held in } e, \\ \textit{not tested} & \text{otherwise,} \end{cases} \tag{3}$$

where c is read as recommending or warning against an action and predicting its result. Only a *support* or *refute* of Eqn. (3), each citing the milestone steps behind it, enters the standing of Eqn. (2). Verdicts are filed alike for memories and for hypotheses, knowledge abstracted from one finished episode and not yet delivered, whose counts are written s_h, q_h and n_h. One-way gates, which never add support, keep a verdict from confirming a hypothesis on its own episode or crediting what merely shaped an episode: no hypothesis is judged against its own episode, and neither delivery nor adoption counts as support (Appendix A.2).

Usage. The other two causes are read from the usage ledger $\mathcal{U}_i$, one usage record per memory delivered at a step, which extends the memory to $M_i = (\mathcal{E}_i, \mathcal{R}_i, \Lambda_i, \mathcal{U}_i, \dots)$. At the episode end, a model call, the adoption judge, marks whether the actor adopted each delivered memory, and the next observation shows whether the environment accepted the adopted action (Section 2). The three beliefs are:

$$\ell_c = \mathrm{lb}_z\big(s_c/n_c\big), \quad \varepsilon_r = \min\big(1,\ \alpha_r/\bar{\alpha}\big), \quad \nu_r = q_r^{\mathrm{ex}}/n_r^{\mathrm{ex}}, \tag{4}$$

where the correctness belief ℓ_c, a lower confidence bound at level z on the support share of claim c, keeps a short ledger from reading as strong. The adoption belief ε_r compares the share α_r of r's deliveries that the actor adopted with that share $\bar{\alpha}$ over all delivered memories, and the execution

belief ν_r is the share, q_r^{ex} of n_r^{ex}, of r's adopted actions that the environment refused (Appendix A.3, Appendix A.4).

Kept apart, the three cannot mask one another: a memory the actor seldom adopts, or cannot act on, never reads as more or less correct than its verdicts say. For the same reason, a refutation from an episode in which the environment refused every action by which the actor adopted r is re-filed as *not tested*, since the failure is the actor's. The standing therefore decides what the memory keeps, and adoption and execution how a memory is delivered.

3.2.2 Belief-Driven Maintenance

While the beliefs say how far each claim holds now, the knowledge itself must change as evidence arrives: a hypothesis has no evidence beyond its own episode, a wording may outrun its evidence or repeat another, and some memories prove wrong. Belief-driven maintenance lets the standing decide each change at the episode end, through deferred commitment, revision and withdrawal.

Deferred commitment. A model call, the hypothesis generator, abstracts hypotheses from each finished episode into the hypothesis buffer $\mathcal{H}_i$, extending the memory to $M_i = (\mathcal{E}_i, \mathcal{R}_i, \Lambda_i, \mathcal{U}_i, \mathcal{H}_i, \dots)$. A hypothesis is delivered only once promoted into $\mathcal{R}_i$, which needs another episode's support unless it carries a grounded procedure ψ_h, a restatement quoted from its own episode's steps that lets a concrete procedure be delivered to the actor before any episode repeats it. A hypothesis h is promoted when:

$$\text{promote}(h) \iff \big(s_h \geq 1 \ \vee \ \psi_h \neq \varnothing\big) \ \wedge \ q_h/n_h \leq \rho, \tag{5}$$

where ρ caps the refuted share (values of all constants in Appendix B.1). A quotation shows only that the action was taken, so a memory promoted through ψ_h starts with a low correctness belief (Appendix A.5).

Revision. A wording broader than its evidence is refined to stricter conditions or split, rather than discarded, and memories stating one claim in different words are merged. A model call, the editor, proposes a refinement or a split when an episode refutes a memory and, because a refutation may never arrive, at a periodic review, which may carry earlier verdicts over to the new wording. A second call, the merge judge, proposes merges when a memory is promoted. A proposal commits only if the assessor, judging it again against the episodes behind it, finds it supported, so a wording changes only where evidence supports the change; a committed split or merge supersedes the memories it revises (Appendix A.6).

Withdrawal. A memory whose standing has turned against it stops being delivered but keeps its ledger. A promoted memory r is withdrawn when:

$$\text{withdraw}(r) \iff n_r \geq n_{\text{wd}} \ \wedge \ q_r/n_r \geq \rho_{\text{wd}} \ \wedge \ q_r^{\text{fr}} \geq 1, \tag{6}$$

where n_{wd} and ρ_{wd} are the withdrawal floors and q_r^{fr} counts the refutations recorded against the current wording itself, not carried over, so that a revised memory is not withdrawn before its new wording is itself refuted (Appendix A.7). What maintenance leaves in $\mathcal{R}_i$, neither withdrawn nor superseded, is what the next episode may deliver.

3.2.3 Belief-Conditioned Control

While maintenance decides what may be delivered, ranking memories by one relevance score lets similarity stand in for evidence, putting the most similar memory first and in full however weakly supported. The belief-conditioned controller, which implements μ, therefore separates three decisions: which memories are considered, whether and in what form each is delivered, and how far to rely on each. Procedural memory $\mathcal{P}_i$ completes $M_i = (\mathcal{E}_i, \mathcal{R}_i, \Lambda_i, \mathcal{U}_i, \mathcal{H}_i, \mathcal{P}_i)$; its workflows for the goal are appended to the block outside these decisions (Appendix A.11).

Which memories. The situation at step t is the goal g, the current observation and the last w steps; from it, a model call, the query formulator, writes the search keys Q_t. The pool of candidates is:

$$\mathcal{C}_t = \Big(\bigcup_{u \in Q_t} \text{top}_{k_{\text{norm}}}^{\sigma_r} \big\{ r \in \mathcal{R}_i^{\circ} : \sigma_r^{\text{claim}} \geq \eta \big\} \Big) \cup \mathcal{R}_i^{\psi}(g), \quad \sigma_r = (1-\beta)\, \sigma_r^{\text{claim}} + \beta\, \sigma_r^{\text{cond}}, \tag{7}$$

where $\mathcal{R}_i^{\circ}$ holds the memories neither withdrawn nor superseded that have a supporting verdict, those with a grounded procedure being recalled instead as $\mathcal{R}_i^{\psi}(g)$, the k_{proc} whose procedure is most semantically similar to g. For a key u, $\sigma_r^{\mathrm{claim}}$ and σ_r^{cond} are its semantic similarities to r's claim and to r's conditions; the floor η admits a candidate, and $\mathrm{top}_{k_{\mathrm{norm}}}^{\sigma_r}$ keeps the k_{norm} with the highest blend σ_r, at weight β, so that a memory whose conditions match the situation ranks higher (Appendix A.8).

Whether and in what form. A candidate may still not apply as written. A model call, the applicability judge, reads it with the situation and either delivers a compact restatement of its claim, the *Trust* form, or sends it back to its episodes. There a second call, the evidence judge, delivers a sentence grounded in the milestone steps its verdicts cite, the *Trace* form, or withholds it. A memory with a grounded procedure whose adopted actions the environment keeps refusing, a high ν_r, is delivered as that procedure alone, whose quoted steps an episode actually took. When the judges deliver nothing, the most similar candidate is delivered as the fallback pick, and the actor, provided with no memory a judge approved, explores under its own policy π, the *Explore* form (Appendix A.9).

How far to rely. Each memory delivered from $\mathcal{R}_i$ carries a reliance label telling the actor how far to rely on it. The grade γ, a fixed rule over the ledgers and the delivered wording, sets it as the first branch that holds:

$$\gamma(r) = \begin{cases} \textit{weak match} & r \text{ is the fallback pick,} \\ \textit{disputed} & q_r \geq 1, \\ \textit{well-tested} & \ell_r \geq \ell_{\min},\ n_r \geq n_{\min},\ c_r = 1, \\ & s_r^{\mathrm{fr}} \geq \mathbf{1}\big[\text{a refinement, a split or a merge changed the current wording}\big], \\ \textit{tentative} & \text{otherwise,} \end{cases} \tag{8}$$

where $\ell_{\min}$ and $n_{\min}$ are the lines ℓ_r and n_r must reach and s_r^{fr} counts the supporting episodes judged since the current wording was written. The check c_r holds when the evidence reaches what is delivered: the action was seen taken, the delivered wording adds nothing the verdicts did not judge, and the memory's conditions have not often failed where it was delivered (Appendix A.10). The label then orders the delivered memories, ties broken by ε_r and then by σ_r.

The grade thus discloses a refutation rather than averaging it away, neither a memory whose action was never taken nor a revised wording unsupported since can be *well-tested*, and similarity decides which memories are considered but not how far the actor relies on them.

4 Experiments

4.1 Setup

We evaluate on four long-horizon benchmarks from AgentBoard (Ma et al., 2024): ALFWorld (AW), ScienceWorld (SW), BabyAI, and PDDL. Following the streaming protocol of Evo-Memory (Wei et al., 2025), each stream presents the tasks of a given benchmark sequentially from easy to hard, with memory persisting across episodes. Each task is allowed at most 30 steps for exploration with a temperature of 0. Memory is written at episode ends and retrieved at every decision step.

We compare MCMem with nine baselines based on the task success rate: The no-memory agent; Six training-free baselines whose memory is managed by the agent itself as MCMem: A-Mem (Xu et al., 2025), Dynamic Cheatsheet (Suzgun et al., 2026), AWM (Wang et al., 2025b), Memora (Xia et al., 2026), MemRL (Zhang et al., 2026) and MCMA (Liang et al., 2026); Two RL-based baselines that trained a seperate memory management agent: Mem-α (Wang et al., 2025a) and Mem-T (Yue et al., 2026). Memora and MCMA are counted as training-free type since their checkpoints are not released. The backbone of the actor agent is set as Qwen3.5-4B, 9B and 27B, respectively, in order to test the method's universality across different model scales. Detailed configurations are introduced in Appendix B.1.

Table 1: Success rate (%) over each whole task stream; columns 4B, 9B and 27B are the size of the Qwen3.5 backbone model; Bold, underlined, italic: the best, second and third value in a column.

	ALFWorld			ScienceWorld			BabyAI			PDDL		
Method	4B	9B	27B	4B	9B	27B	4B	9B	27B	4B	9B	27B
No memory	53.7	71.6	85.1	42.2	30.7	64.4	28.6	41.1	50.9	**38.3**	35.0	61.7
AWM	56.0	71.9	**89.8**	27.8	41.1	**67.8**	21.4	39.3	50.9	28.3	13.3	*56.7*
A-Mem	61.9	*72.1*	79.9	45.6	31.1	*65.6*	26.8	*42.0*	55.4	23.3	23.3	55.0
Mem-α	40.3	57.5	64.9	*44.4*	50.0	66.7	26.8	41.1	56.2	*31.7*	26.7	53.3
Mem-T	46.3	70.9	83.6	26.7	41.1	62.2	24.1	38.4	51.8	*31.7*	25.0	61.7
MCMA	*59.7*	76.9	*87.3*	*44.4*	*47.8*	63.3	18.8	*42.0*	58.0	*31.7*	*30.0*	50.0
Memora	55.2	64.2	85.8	*44.4*	20.7	66.7	21.4	48.2	*57.1*	30.0	26.7	61.7
Dynamic Cheatsheet	50.7	69.2	82.1	32.2	20.0	63.3	*27.7*	40.2	53.6	18.3	28.3	50.0
MemRL	56.0	66.4	74.6	40.0	23.3	64.4	21.4	34.8	51.8	20.0	13.3	*56.7*
MCMEM	**71.6**	**81.6**	87.6	**55.6**	**56.3**	**67.8**	**31.2**	**52.7**	**61.6**	33.3	**36.7**	**64.2**

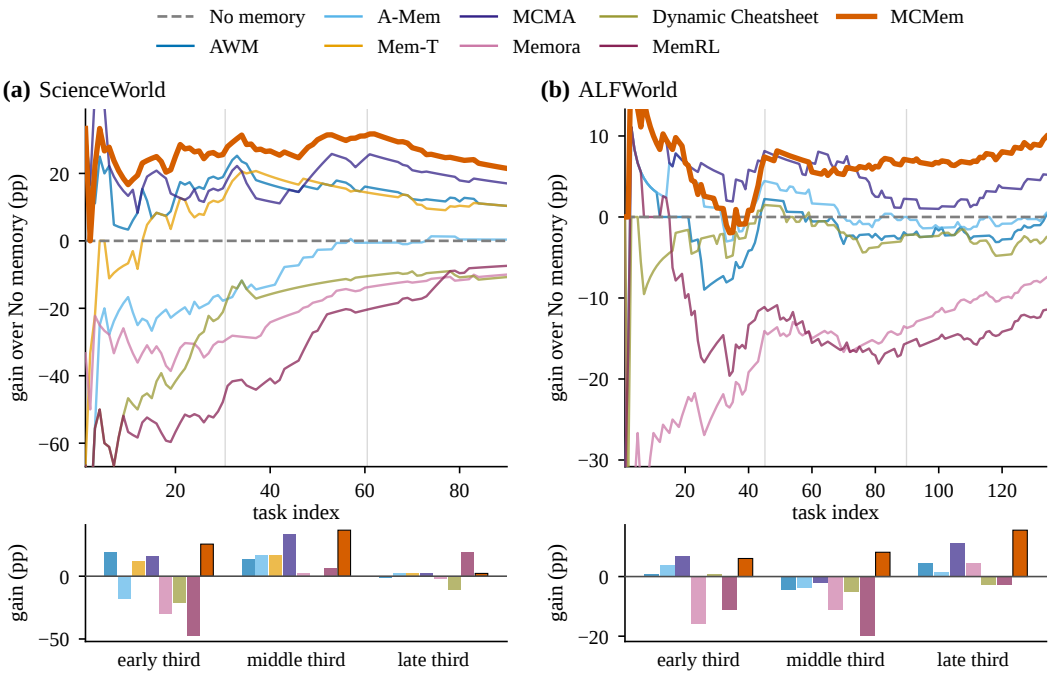

Figure 2: Gain in cumulative success rate over no memory against task index at 9B on (a) SW and (b) AW for MCMEM, the six training-free baselines and, in (a) only, Mem-T; bars: gain per third of the stream; vertical axis clipped.

4.2 Overall Performance

Table 1 reports the success rate of every method on the four datasets at the three backbone model sizes. We can see several findings: i) MCMEM achieves globally best across all datasets, with the largest margins exceeding the strongest baseline by roughly 10%; ii) RL-based methods do not consistently outperform training-free memory baselines despite their richer optimisation in memory management. These methods were originally trained for long-range conversational tasks rather than open-ended sequential decision tasks, which makes them neglect the importance of exploration-exploitation balance; iii) Hierarchical memory methods (MCMA, Memora, A-Mem) can be very strong on individual benchmarks yet swing widely across them, suggesting that their lack of a reliable mechanism for stable transfer. iv) No memory is itself highly competitive at the 4B scale, sometimes outperforming most memory-augmented baselines. This indicates that small actors lack the capacity to manage and utilize external memory reliably. However, MCMEM remains above it

Table 2: Success rate and the three measures of Section 4.4 over the last two-thirds of each 9B stream. Bold, underlined, italic: the best, second and third value in a column (lower is better for extra looping); the no-memory row, against which extra looping and extra coverage are measured, is not ranked on those two.

	ScienceWorld				ALFWorld			
Method	Success (%)	Exploitation share	Extra looping	Extra coverage	Success (%)	Exploitation share	Extra looping	Extra coverage
No memory	31.4	0.500	0.000	0.00	67.8	0.500	0.000	0.00
AWM	28.3	0.315	*0.098*	*−0.27*	67.8	0.557	0.157	−3.15
A-Mem	32.5	*0.381*	**0.015**	−0.06	*66.7*	*0.565*	0.097	−1.68
Memora	*34.2*	0.344	0.104	−0.51	64.4	0.511	0.191	−3.54
Dynamic Cheatsheet	35.8	0.284	0.116	−0.81	63.7	0.518	**0.064**	**−0.81**
MemRL	26.7	0.388	0.233	−1.14	56.3	0.679	0.368	−5.85
MCMEM	**38.3**	**0.530**	0.082	**0.02**	**79.6**	**0.720**	*0.101*	*−2.13*

almost on all the model scales, which validates that accumulated verdicts can let the agent sense its own competence boundary and prevent it from blindly following memories. Appendix B.4 further demonstrates this point via interventional studies.

The success rates of Table 1, taken over whole streams, cannot reveal whether accumulating memory compounds or corrodes capability. Therefore, we further track the agent's progress along the easy-to-hard streaming tasks on SW and AW. As shown in Figure 2, MCMEM's gain gradually widens as the stream progresses, achieving superior performance in the hardest segment. This further verifies its memory management and utilization capability in long-term self-evolving scenarios. We also shift the task structure to testify its robustness in Appendix B.5.

4.3 Ablation

Figure 3 reports ablation results for the key modules. The first four variants each remove a component; in the fifth, every delivered memory carries the plain prefix `Memory:` in place of its reliance label, so the actor sees no reliability wording while the beliefs are still estimated. Every variant degrades success on at least one benchmark. The plain-prefix variant is the most revealing: it retains the entire belief-estimation process and alters only what is delivered to the actor and the position of the reliance label in the block's ordering. That it still degrades success isolates the reliance label's own contribution beyond the beliefs that produce it, suggesting that part of the gain stems from the label as communicated to the actor rather than solely from the underlying beliefs. Detailed agent behaviour analysis for all variants are provided in Appendix B.2 and Appendix B.3.

4.4 Exploration and Exploitation

Once memory accumulates, success rate alone cannot reveal whether the agent acts on useful memory or gets stuck in failed strategies. We thereby propose three measures to dissect the exploration and exploitation behaviors of different methods over the last two thirds of the task stream.

We first pair every memory run task-by-task with every no-memory run. For each pair, we identify the first step where the memory run's action diverges from the no-memory run's, all prior actions and observations being identical. From that step, whichever run completes a new subgoal first, measured by AgentBoard's progress annotations (Ma et al., 2024), wins. The *exploitation share* is the memory run's win

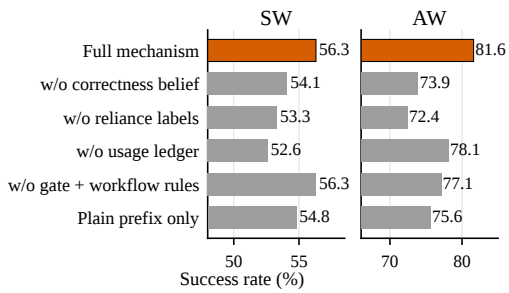

Figure 3: Success rate (%) of the ablation variants at 9B, means of three repeats; horizontal axis clipped. The AW full mechanism is the 9B cell of Table 1.

fraction; 0.5 means the memory-induced change neither helped nor hurt. On tasks both runs fail after all 30 steps, *extra looping* is the added fraction of steps that repeat an earlier action under the same observation without intervening progress, and *extra coverage* is the added count of distinct places and objects encountered.

On exploitation, MCMEM beats every baseline on both SW and AW. This is attributed to the metacognition control on managing and retrieving memory, which ranks the best-supported entry above the most similar one. On exploration, MCMEM maintains top-3 level among all the baselines, which indicates that its exploration tendency is still preserved even as the memory accumulated. Overall, MCMEM is more balanced than every baseline on the two datasets.

5 Related Work

Memory for LLM agents is widely studied (Huang et al., 2026), with early systems already separating context from persistent storage (Packer et al., 2023). In agentic memory the agent itself decides what to store, update, and retrieve. Several methods add, edit, or index items through prompting alone (Xu et al., 2025; Suzgun et al., 2026; Wang et al., 2025b; Xia et al., 2026). MemRL freezes the model and learns a per-item utility from reward (Zhang et al., 2026), while other methods train the memory-managing model with reinforcement learning (Wang et al., 2025a; Yue et al., 2026).

Hierarchical memories differ in what the levels represent: insight versus interaction (Zhang et al., 2025a), planning versus execution (Ye et al., 2025), or degree of abstraction (Liang et al., 2026). A hierarchy organizes storage but by itself says nothing about how far to rely on any item. Some methods go further and score each item by the result of using it (Tang et al., 2026; Forouzandeh et al., 2026; Deng et al., 2026), yet the score conflates a wrong item with one not followed or not carried out. Outside any hierarchy, one line of work scores items by task outcome propagated over a provenance graph, with the same conflation (Liao et al., 2026b); another attaches a confidence that rises with observations and measures support rather than whether reliance paid off (Liao et al., 2026a). MCMEM judges each item against the actions the agent took where it applied, separating adoption and execution from correctness, and sets reliance from that evidence.

Recent works have started to explore metacognitive mechanisms to memory system. MCMA trains a learnable copilot to decide how experience should be abstracted (Liang et al., 2026). MMPO penalizes summaries that induce high uncertainty about the task state (Liu et al., 2026). MetaMem iteratively distills knowledge-utilization policies through self-reflective optimization (Xin et al., 2026). In each case, metacognition governs the *process* of memory management—how to abstract, compress, or utilize—without assessing whether any stored item remains reliable. A related thread applies metacognition to the boundary of parametric knowledge, deciding whether the agent should answer on its own, call a tool, or retrieve (Li et al., 2025; Qian et al., 2025; Zhou et al., 2024); these methods assess what the agent *knows*, not what it has *accumulated*. In MCMEM, metacognition targets whether the agent can manage and rely on each item its memory holds, judged from cross-episode evidence that separates adoption from correctness.

6 Conclusion

As LLM agents accumulate cross-task experience, stored memories may silently become unreliable under distributional shift. We argue that an external memory should not only store knowledge but continuously sharpen the agent's awareness of its own memory-augmented capability boundary. To this end, we propose MCMEM, a metacognitive memory mechanism that calibrates the agent's reliance on each memory against provenance-grounded evidence. MCMEM couples a hierarchical memory graph—anchoring every abstraction to its source episodes—with a metacognitive layer that decomposes accumulated evidence into independent beliefs and modulates reliance accordingly. Across four long-horizon benchmarks at three actor scales, MCMEM achieves state-of-the-art success rates while maintaining a balanced exploration–exploitation tradeoff.

Looking ahead, we first plan to combine agentic reinforcement learning with MCMEM to sharpen the agent's calibration of its own memory-augmented capabilities. Besides, we expect to extend MCMEM to large-scale scientific discovery, where well-calibrated exploration–exploitation over extended hypothesis–experiment cycles is even more critical.

AI USE STATEMENT

We used AI (large language model) assistants to discuss and implementing research ideas. We also leveraged them to draft the paper's prose from an outline the authors wrote, and to polish the language. All text was then checked and revised by the authors, who take full responsibility for the content.

ETHICS STATEMENT

This paper raises no particular concerns under the Code of Ethics: all experiments run on public simulated text benchmarks (ALFWorld, ScienceWorld, BabyAI, and PDDL, accessed through AgentBoard), involve no human subjects, and use no personal or private data or newly collected dataset, with open-weight Qwen3.5 models as the underlying actors. Deploying such self-evolving agents outside simulated environments would nonetheless call for further safeguards beyond the scope of this paper.

REPRODUCIBILITY STATEMENT

The mechanism underlying MCMEM is specified in Section 3, with implementation details in Appendix A. All benchmarks are public, and all the used Qwen models are open-weight. Code will be released at an anonymous repository: `https://anonymous.4open.science/r/MCMEM-D0DD/`.

REFERENCES

Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-Embedding: Multi-Linguality, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335. Association for Computational Linguistics, 2024a.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing Instruction Manuals by LLM Agents via Interactive Environmental Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b. arXiv:2405.16247.

Zihao Deng, Yining Zhu, Leiming Wang, Junbo Wang, and Jingfei Lu. Tree-of-Experience: Hierarchical Experience Management for Self-Evolving Agents. *arXiv preprint arXiv:2608.09044*, 2026.

Prakhar Dixit, Sadia Kamal, and Tim Oates. Honest Lying: Understanding Memory Confabulation in Reflexive Agents. *arXiv preprint arXiv:2605.29463*, 2026. ICML 2026 Workshop on Failure Modes in Agentic AI.

Saman Forouzandeh, Wei Peng, Parham Moradi, Xinghuo Yu, and Mahdi Jalili. Learning Hierarchical Procedural Memory for LLM Agents through Bayesian Selection and Contrastive Refinement. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2026. arXiv:2512.18950.

Qisheng Hu, Quanyu Long, and Wenya Wang. When Continual Learning Moves to Memory: A Study of Experience Reuse in LLM Agents. *arXiv preprint arXiv:2604.27003*, 2026.

Wei-Chieh Huang, Weizhi Zhang, Yueqing Liang, Yuanchen Bei, Yankai Chen, Tao Feng, Xinyu Pan, Zhen Tan, Yu Wang, Tianxin Wei, Shanglin Wu, Ruiyao Xu, Liangwei Yang, Rui Yang, Wooseong Yang, Chin-Yuan Yeh, Hanrong Zhang, Haozhen Zhang, Siqi Zhu, Henry Peng Zou, Wanjia Zhao, Song Wang, Wujiang Xu, Zixuan Ke, Zheng Hui, Dawei Li, Yaozu Wu, Langzhou He, Chen Wang, Xiongxiao Xu, Baixiang Huang, Juntao Tan, Shelby Heinecke, Huan Wang, Caiming Xiong, Ahmed A. Metwally, Jun Yan, Chen-Yu Lee, Hanqing Zeng, Yinglong Xia, Xiaokai Wei, Ali Payani, Yu Wang, Haitong Ma, Wenya Wang, Chenguang Wang, Yu Zhang, Xin Eric Wang, Yongfeng Zhang, Jiaxuan You, Hanghang Tong, Xiao Luo, Xue Liu, Yizhou Sun, Wei Wang, Julian McAuley, James Zou, Jiawei Han, Philip S. Yu, and Kai Shu. A Survey of Agent

Memory in the Second Half: Towards Self-Evolving and Long-Horizon Agents. *Transactions on Machine Learning Research*, 2026. Survey Certification. arXiv:2602.06052.

Wenjun Li, Dexun Li, Kuicai Dong, Cong Zhang, Hao Zhang, Weiwen Liu, Yasheng Wang, Ruiming Tang, and Yong Liu. Adaptive Tool Use in Large Language Models with Meta-Cognition Trigger. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 13346–13370, 2025. arXiv:2502.12961.

Sirui Liang, Pengfei Cao, Jian Zhao, Wenhao Teng, Xiangwen Liao, Jun Zhao, and Kang Liu. Learning How to Remember: A Meta-Cognitive Management Method for Structured and Transferable Agent Memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30733–30753, 2026. arXiv:2601.07470.

Junfeng Liao, Qizhou Wang, Jianing Zhu, Bo Du, Rui Yan, and Xiuying Chen. Belief Memory: Agent Memory Under Partial Observability. *arXiv preprint arXiv:2605.05583*, 2026a.

Junwei Liao, Haoting Shi, Ruiwen Zhou, Jiaqian Wang, Shengtao Zhang, Wei Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Bo Tang, Weinan Zhang, and Muning Wen. MemQ: Integrating Q-Learning into Self-Evolving Memory Agents over Provenance DAGs. *arXiv preprint arXiv:2605.08374*, 2026b.

Ziyan Liu, Zhezheng Hao, Yeqiu Chen, Hong Wang, Jingren Hou, Ruiyi Ding, Yongkang Yang, Wence Ji, Wei Xia, and Feng Liu. Meta-Cognitive Memory Policy Optimization for Long-Horizon LLM Agents. *arXiv preprint arXiv:2605.30159*, 2026.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An Analytical Evaluation Board of Multi-turn LLM Agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024. arXiv:2401.13178.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2509.25140.

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

Cheng Qian, Emre Can Acikgoz, Hongru Wang, Xiusi Chen, Avirup Sil, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. SMART: Self-Aware Agent for Tool Overuse Mitigation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 4604–4621, 2025. arXiv:2502.11435.

Shuai Shao, Qihan Ren, Dongrui Liu, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, and Jing Shao. Your Agent May Misevolve: Emergent Risks in Self-evolving LLM Agents. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2509.26354.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.11366.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2026.

Bo Tang, Yang Zhang, Guomian Zhuang, Wenqiang Wei, Gaoyang Zheng, Lindong Xie, Yanchao Tan, Feiyu Xiong, Qingyu Yang, Edward Chung, and Zhiyu Li. From Memory to Skills: Evidence-Grounded Co-Evolution Governance for Long-Horizon LLM Agents. *arXiv preprint arXiv:2607.16621*, 2026.

Yu Wang, Ryuichi Takanobu, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. Mem-α: Learning Memory Construction via Reinforcement Learning. *arXiv preprint arXiv:2509.25911*, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025b. arXiv:2409.07429.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-Memory: Benchmarking LLM Agent Test-time Learning with Self-Evolving Memory. *arXiv preprint arXiv:2511.20857*, 2025.

Menglin Xia, Xuchao Zhang, Shantanu Dixit, Paramaguru Harimurugan, Rujia Wang, Victor Rühle, Robert Sim, Chetan Bansal, and Saravan Rajmohan. MEMORA: A Harmonic Memory Representation Balancing Abstraction and Specificity. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.03315.

Haidong Xin, Xinze Li, Zhenghao Liu, Yukun Yan, Shuo Wang, Cheng Yang, Yu Gu, Ge Yu, and Maosong Sun. MetaMem: Evolving Meta-Memory for Knowledge Utilization through Self-Reflective Symbolic Optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 5473–5492, 2026. arXiv:2602.11182.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic Memory for LLM Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2502.12110.

Shicheng Ye, Chao Yu, Kaiqiang Ke, Chengdong Xu, and Yinqi Wei. H^2R: Hierarchical Hindsight Reflection for Multi-Task LLM Agents. *arXiv preprint arXiv:2509.12810*, 2025.

Ye Yu, Xiaopeng Yuan, Haibo Jin, Heming Liu, Yaoning Yu, and Haohan Wang. Do Self-Evolving Agents Forget? Capability Degradation and Preservation in Lifelong LLM Agent Adaptation. *arXiv preprint arXiv:2605.09315*, 2026.

Yanwei Yue, Guibin Zhang, Boci Peng, Xuanbo Fan, Jiaxin Guo, Qiankun Li, and Yan Zhang. Mem-t: Densifying rewards for long-horizon memory agents. *CoRR*, abs/2601.23014, 2026.

Guibin Zhang, Muxin Fu, Kun Wang, Guancheng Wan, Miao Yu, and Shuicheng Yan. G-Memory: Tracing Hierarchical Memory for Multi-Agent Systems. In *Advances in Neural Information Processing Systems*, volume 38, pp. 12988–13018, 2025a. arXiv:2506.07398.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025b. arXiv:2505.00212.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-Evolving Agents via Runtime Reinforcement Learning on Episodic Memory. *arXiv preprint arXiv:2601.03192*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024. arXiv:2308.10144.

Weixiang Zhao, Yingshuo Wang, Yichen Zhang, Yang Deng, Yanyan Zhao, Wanxiang Che, Bing Qin, and Ting Liu. Large Language Model Agents Are Not Always Faithful Self-Evolvers. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026a. arXiv:2601.22436.

Weixiang Zhao, Yingshuo Wang, Yichen Zhang, Yanyan Zhao, Yu Zhang, Yang Wu, Dandan Tu, Bing Qin, and Ting Liu. Rethinking Experience Utilization in Self-Evolving Language Model Agents. *arXiv preprint arXiv:2605.07164*, 2026b.

Yujia Zhou, Zheng Liu, Jiajie Jin, Jian-Yun Nie, and Zhicheng Dou. Metacognitive Retrieval-Augmented Large Language Models. In *Proceedings of the ACM Web Conference 2024 (WWW)*, 2024. arXiv:2402.11626.

Shiding Zhu, Yudi Qi, Yajie Wang, Jiaze Li, Chao Song, Yaorui Shi, Yibo Miao, Hanqi Gao, and Kai Zhang. Escaping the Self-Confirmation Trap: An Execute-Distill-Verify Paradigm for Agentic Experience Learning. *arXiv preprint arXiv:2606.24428*, 2026.

A Implementation Details

This appendix gives the exact rules behind the mechanisms of Section 3, in the order in which a piece of knowledge passes through them. An episode is first stored as a record (Appendix A.1). At the end of each episode, stored claims are judged against it (Appendix A.2), and the verdicts, together with records of how delivered memories were used, give each memory its beliefs (Appendix A.3, Appendix A.4). These beliefs decide which hypotheses are promoted (Appendix A.5), how memories are revised (Appendix A.6) and when a memory is withdrawn (Appendix A.7). In the episodes that follow, the memories that remain are recalled, delivered and labelled at every step (Appendix A.8, Appendix A.9, Appendix A.10), and procedural memory offers concrete actions beside them (Appendix A.11). The appendix ends with the whole loop as an algorithm, its cost and the prompts (Appendix A.12, Appendix A.13, Appendix A.14).

Constants. The thresholds, caps and sample sizes of the mechanism are written as symbols. Their values are set before the task stream begins and are listed in Appendix B.1. The only numbers written out in this appendix are limits on how much text a model call receives and a few counts that only one step uses.

Word matching. Several checks compare two texts word by word, for instance whether a grounded procedure really quotes its episode, or whether a delivery adds words that its memory does not contain. These checks ignore function words, the words on a fixed list of articles, pronouns, prepositions, conjunctions, auxiliary verbs and a few discourse adverbs. The list holds no noun and no verb, and the words it lets through are called *content words*. Procedural memory adjusts this rule in a few places, each stated where it applies (Appendix A.11). Where a check allows for inflection, two words count as the same if they become equal once at most one of the endings `-ing`, `-ed`, `-es` or `-s` is removed from each, provided at least three characters remain; `opens` and `opened` therefore match.

A.1 The Episode Record

Every later decision reads an episode through its record, so the record fixes what evidence the memory can ever hold. When an episode ends, a shortened copy of it is prepared for the model calls that read it. The copy holds the first 30 recorded events, with each observation and each environment reply cut to 1,200 characters, together with the goal, taken from the first event, and the final outcome. The summarizer, a model call (Appendix A.14.1), reads this copy and returns the milestone steps: groups of step references, each with a sentence saying how those steps led to the outcome. The references are checked against the real step identifiers of the episode. If the call fails, or cites no real step, the last eight steps of the episode become its milestone steps instead.

The stored record holds the goal, the first observation, the outcome, the milestone steps and the full text of every step; the step text is kept so that later model calls can quote it, not for measuring similarity. Each record is embedded twice, once from its goal and once from its milestone steps together with its outcome, and we call these two embeddings the record's *facets*. At each episode end, both facets are used to find the past episodes and memories that are judged against the new episode (Appendix A.2); when memory is read, the second facet ranks past episodes against a search key (Appendix A.8). A verdict may cite only milestone steps, so what the summarizer marks limits what any later verdict can point to.

A.2 THE VERDICT PROTOCOL

At each episode end, the assessor judges stored claims against episodes (Eqn. (3)). Three choices decide what these verdicts are worth: which claim is judged against which episode, how the assessor reaches its answer, and which answers are kept as evidence. This subsection takes them in turn and then collects the one-way gates that these rules form.

Which pairs are judged. An episode can bear on a claim only if it met the situation the claim describes, so the new episode is paired only with the claims and episodes most similar to it, found separately on each of its two facets. On each facet, the new episode draws the k_{ep} most similar past episodes, the k_{abs} most similar promoted memories that are neither withdrawn nor superseded, and, in a separate ranking, the k_{abs} most similar withdrawn memories. Each facet's draw forms one *context*. Its *historical episodes* are the k_{hist} most similar drawn episodes. Its *target* is the promoted memory, neither withdrawn nor superseded, that is most similar to the new episode on that facet. Its *focus hypotheses* are up to k_{focus} hypotheses proposed by the new episode or by a historical episode, and those of the new episode are taken first. When the two facets give the same target, their contexts are merged into one.

Within a context, three kinds of pair are judged: (i) each focus hypothesis proposed by the new episode against each historical episode, (ii) each focus hypothesis proposed by a historical episode against the new episode, and (iii) the target against the new episode. Every pair thus puts a claim against an episode other than the one it came from, and a validity check rejects any pair of a hypothesis with the episode that proposed it. Each withdrawn memory that was drawn is also judged against the new episode, outside any context, and this verdict only adds an entry to its ledger. A promoted memory can still meet its source episode in two ways. One is the supporting verdict that a promotion through a grounded procedure writes from the source episode (Appendix A.5); the other arises when a proposed revision is judged again against every episode in the memory's ledger, where the source episode is judged like any other.

One further pair comes from no context. Suppose the episode that just ended delivered a memory at its current wording, that wording was written by a revision, and the memory's ledger meets $\ell_{\min}$ and $n_{\min}$ with no refutation but holds no support recorded against the current wording. The well-tested label requires such support ($s_r^{\mathrm{fr}} \geq 1$ in Eqn. (8)), so the memory is also judged against this episode, which gives it a chance to earn that support. Delivery alone never creates this pair: a memory whose wording no revision wrote, or whose ledger falls short of $\ell_{\min}$ or $n_{\min}$, is not paired this way. Because the pair belongs to no context, it cannot trigger a revision; it only adds one entry to the ledger. All pairs are judged in two rounds, first those of hypotheses and memories that are not withdrawn and then those of withdrawn memories; a proposed revision is judged again in a round of its own (Appendix A.6).

How the assessor answers. The assessor, a model call (Appendix A.14.3), judges one pair at a time the way one checks a prediction: it finds where the episode did what the claim is about, and compares what happened there with what the claim predicted. Its inputs are the claim with its source outcome and conditions, and the episode's goal, outcome and milestone steps, with the text of those steps and the narrative of how they led to the outcome. It answers five questions in order. It first states which action the claim is about, whether the claim recommends or denies that action, and what result the claim predicts. It then names the milestone steps at which the episode took that action under the claim's conditions, says what was observed at those steps, and says whether that observation is consistent with the prediction, contradicts it, or does not bear on it.

The verdict follows from the last answer by a fixed map: consistent gives *support* and contradicted gives *refute*. When the observation does not bear on the prediction, one more answer says whether the claim's own conditions never held in the episode, which gives *set aside*, or whether the episode was about something else, which gives *not tested*. An episode that never took the action under the claim's conditions therefore leaves the claim's standing unchanged, however the episode ended.

Which answers are kept. A fixed check, the audit, accepts an answer only if its verdict is the one this map gives, and only if the answer cites milestone steps exactly when the verdict is *support* or *refute*. An answer that fails the audit, or cannot be parsed, is sent back once for correction. If it fails again, the pair is left unresolved, and for this episode the unresolved pair blocks the promotion and

any revision that depends on it. A failed answer is never stored, so every stored verdict is audited except the supporting verdict that a promotion through a grounded procedure writes from the source episode (Appendix A.5). A hypothesis's ledger therefore holds audited verdicts only, which makes the counts of Eqn. (5) counts of audited episodes.

Two rules on direction. Two further rules concern which way a verdict may go. The first is an instruction to the assessor: a claim denying that an action can reach the goal is refuted when the episode succeeded with that action among its milestone steps. The audit compares a verdict only with the assessor's own answers, so this rule holds as far as the assessor follows it. The second is applied by the mechanism itself. When the claim came from a failed episode and the episode judged is a success, a supporting verdict is re-filed as neither support nor refutation, and its cited steps are cleared. A successful episode can therefore never add support to a lesson drawn from a failure.

The one-way gates. Without such rules, a claim could be confirmed by the very episode it came from, or by what merely shaped an episode, such as its outcome or the delivery of the memory itself. The one-way gates of Section 3.2.1 rule this out. They are fixed rules on which verdicts are made or counted as evidence, and none of them can add support. They are the refusal to pair a hypothesis with its own episode, the rule just stated for claims drawn from failures, and the rule that neither the delivery of a memory nor the actor's adoption of it ever counts as support. The execution gap of Appendix A.4 is the one gate that removes a refutation rather than support.

A.3 The Correctness Belief

The standing counts the verdicts on a memory; the correctness belief turns those counts into one number that stays cautious while the verdicts are few. For a promoted memory, the verdicts on its current wording give the sets of distinct supporting and refuting episodes, and hence the counts s_r, q_r and $n_r = s_r + q_r$ of Eqn. (2). The correctness belief ℓ_r of Eqn. (4) is the lower end of the Wilson score interval for the support share at level z:

$$\ell_r = \frac{\hat{p}_r + \frac{z^2}{2n_r} - z\sqrt{\frac{\hat{p}_r(1-\hat{p}_r)}{n_r} + \frac{z^2}{4n_r^2}}}{1 + z^2/n_r}, \tag{9}$$

where $\hat{p}_r = s_r/n_r$ is the observed support share; a memory with no decisive verdict, $n_r = 0$, has no correctness belief yet. Read intuitively, ℓ_r is the lowest support share that the verdicts so far still make plausible. It lies well below $\hat{p}_r$ while few episodes have decided, and it approaches $\hat{p}_r$ as more do. With no refutation it reduces to $n_r/(n_r + z^2)$, so one supporting episode gives about 0.21 and ten give about 0.72. At the values of Appendix B.1, reaching $\ell_{\min}$ therefore takes four supporting episodes, one more than the floor $n_{\min}$.

Support for a revised wording. A wording written by a refinement, a split or a merge is new text, so the verdicts that led to it cannot also confirm it. For such a wording the mechanism counts s_r^{fr} of Eqn. (8), the distinct episodes whose supporting verdict was recorded after the wording entered the memory. A counter that increases with every committed change dates each wording and each verdict. A verdict that the revision itself judged again, or carried over, was made no later than the wording and never counts. A wording set at promotion needs no such count, because its ledger was copied from a hypothesis whose verdicts already judged the same claim on episodes other than its source. A wording whose origin the log does not record is treated as set at promotion.

Which verdicts each decision reads. The only unaudited verdict is the one written at a promotion through a grounded procedure, so whether a decision counts it matters only for a memory promoted that way. Decisions that weigh refutations read audited verdicts only, so this verdict never lowers a refute share. They are the refutation cap of promotion, withdrawal, and the counts shown to the two read-time judges for a refuted memory (Appendix A.8). Decisions that only ask whether some support exists count every verdict: the support $s_h \geq 1$ of Eqn. (5), and the support a memory needs before it can be recalled. The grade reads audited verdicts when any exist and all verdicts otherwise, so a memory whose only verdict is this one still has a correctness belief. Promotion and withdrawal read the refute share rather than the correctness belief. Withdrawal reads q_r^{fr} and the grade reads s_r^{fr}, both over audited verdicts. None of these quantities is stored; each is recomputed from the ledger when read, so every change takes effect at once.

A.4 THE USAGE LEDGER

Verdicts say whether a memory's claim holds; the usage ledger records what happened when the memory was delivered. Its records give the adoption and execution beliefs of Eqn. (4), decide when a refutation should not count against the memory, and raise the demotion flag that the grade reads.

The records. One usage record is written for each promoted memory at each step at which it is delivered. It holds the memory, the wording then in force, the task and the step, the reliance label the memory was delivered under, and the text delivered. Four fields are filled when the episode ends: the episode identifier, whether the actor adopted the memory, whether the memory's conditions held, and, for an adopted record, whether the environment accepted the action taken at that step.

Adoption. Whether the actor adopted a memory is decided by the adoption judge, a model call made after the episode ends, when the whole action sequence is known, so that it can see an action taken a few steps after a delivery. The adoption judge answers once per memory delivered in the episode, not once per delivery. It sees the goal, the whole trajectory as numbered steps with their actions and replies, the steps at which the memory was delivered, and up to three of the delivered texts, each cut to 400 characters. It answers under four instructions. First, adoption is about what the actor did, not the words it used, so an actor that did what a delivery asks in its own words adopted it. Second, the action must come after the first delivery and within five steps of some delivery. Third, a delivery that only states a fact and asks for nothing, or whose situation never arose, was not adopted. Fourth, every memory delivered must be answered. One answer covers every record of that memory in the episode.

The adoption belief. Beside the adoption judge's answer, a word-reuse check records whether the action taken at the delivery step reused the content words of the delivered text. This check decides the record only when the adoption judge left the memory unanswered, because the answer could not be parsed or because it answered fewer memories than it was given. A record that carries either answer is *decided*, and a memory's decided records are the answered deliveries of Eqn. (4). The share α_r of r's decided records that the actor adopted is compared with $\bar{\alpha}$, the same share over the decided records of all promoted memories, which gives $\varepsilon_r = \min(1, \alpha_r/\bar{\alpha})$. A memory adopted at least as often as the average one thus has $\varepsilon_r = 1$, and one adopted half as often has $\varepsilon_r = 0.5$. Until a memory has n_{price} decided records, its adoption belief stays at one, so that a memory with few records is not ranked down for lack of evidence. The adoption belief is read only by the order of the block (Appendix A.9).

Execution and the execution gap. The execution field is filled by a fixed rule, with no model judgement. The environment interface marks at every step whether the action was accepted; on ALFWorld, an action is accepted when the reply is anything other than `Nothing happens.` A step without this mark leaves the field unknown. The field has two uses. First, it keeps a memory from being blamed for the actor's failures. Consider, at the episode end, a memory that the actor adopted at least once in this episode and whose every adopted action the environment refused. Any refutation of it recorded in this episode is re-filed as *not tested*, with the execution gap as its reason: its cited steps are dropped and its refute edge is removed, since the failure is the actor's, not the memory's. A single accepted attempt means the memory was fairly judged, and the refutation stands. The re-filing happens after the rest of the episode's maintenance, so it affects only later decisions.

Second, the field gives the execution belief. At read time, n_r^{ex} counts the adopted records of r whose action was accepted or refused, and q_r^{ex} counts those refused, so $\nu_r = q_r^{\text{ex}}/n_r^{\text{ex}}$ is the share of r's adopted actions that the environment refused. A high ν_r changes only the form in which a memory with a grounded procedure is delivered (Appendix A.9). The memory's counts, correctness belief, reliance label and place in the order are unaffected, so the execution belief changes how a memory is delivered, never whether it is believed.

The demotion flag. The records also show whether a memory's conditions held where it was delivered, which the grade reads through the demotion flag b_r. A record is *verified* when the episode's verdict on the memory says that its conditions held or that they never held, a field filled at the episode end. Like the quantities of Appendix A.3, the flag is not stored but recomputed from the records

whenever it is read. Once a memory has n_{use} verified records, the flag is raised when two things hold. First, the lower bound of Eqn. (9), applied to the verified records with those whose conditions held as successes, falls below ℓ_{use}. Second, at least one record with unmet conditions concerns the current wording. A raised flag bars the well-tested label and moves the memory up the queue of the periodic review (Appendix A.6); it changes nothing else. A memory that keeps being delivered where its own conditions fail is thus kept from the well-tested label, however well its claim has been supported.

A.5 Hypotheses and Promotion

A hypothesis is knowledge drawn from one episode that the actor has not yet been given. This subsection explains how hypotheses are proposed, when a hypothesis may carry a grounded procedure, and how Eqn. (5) turns a hypothesis into a promoted memory.

Proposing hypotheses. Hypotheses are proposed once per finished episode by the hypothesis generator, a model call (Appendix A.14.2) that reads the same shortened copy of the episode as the summarizer (Appendix A.1). Its prompt depends on the outcome. A successful episode is asked for reusable procedures and strategy patterns, and the prompt calls the first kind a workflow, which is not a workflow of procedural memory. A failed episode is asked for failure lessons and guardrails. The generator returns at most k_h hypotheses, each with at most three conditions under which it applies and at most three references to steps of its own episode. For the two kinds that say which action to take, workflows and guardrails, the prompt also asks for a grounded procedure.

Admitting a grounded procedure. A grounded procedure ψ_h restates the claim in the words of the episode that proposed it. It opens the second route of Eqn. (5), which needs no support from another episode, and it is admitted by a fixed check rather than by a model's judgement. A word of ψ_h counts as *copied* only if it lies in a run of at least two consecutive words that appears word for word in the goal or in the steps the hypothesis cites. Over content words, let $\mathrm{cov}(\psi_h)$ be the share of the words of ψ_h that are copied. The grounded procedure is admitted when:

$$\mathrm{cov}(\psi_h) \geq \kappa \ \wedge \ \mathrm{copied}(\psi_h) \not\subseteq \mathrm{words}(c_h), \tag{10}$$

where $\mathrm{copied}(\psi_h)$ is the set of copied content words, $\mathrm{words}(c_h)$ is the set of words of the hypothesis's claim c_h, and κ is the coverage floor.

The first condition makes ψ_h mostly a quotation of its episode rather than a paraphrase. The second makes it quote at least one word that the claim does not already contain; without it, a procedure could pass the floor by repeating the claim instead of the episode. A grounded procedure that fails is dropped, and its hypothesis is kept without one, so it can be promoted only through the first route. Workflows of procedural memory pass a check of their own (Appendix A.11).

The two promotion routes. The condition of Eqn. (5) is checked, without any model call, for every hypothesis that received a verdict at this episode end; it is skipped for a hypothesis with an unresolved pair or one already promoted. The first route, $s_h \geq 1$, needs one supporting verdict in the hypothesis's ledger, which by the pairing rule of Appendix A.2 comes from an episode other than its source. The second route, $\psi_h \neq \varnothing$, applies to a workflow or guardrail whose grounded procedure was admitted. At promotion, this route writes one supporting record from the source episode, citing the hypothesis's own steps. The record is marked unaudited, so no decision that reads audited verdicts only counts it (Appendix A.3). On its own, it also gives a correctness belief of about 0.21 by Eqn. (9), far below the line the well-tested label requires, so such a memory cannot be delivered as well-tested until other episodes support it. Both routes are blocked when the audited refute share of the hypothesis's ledger exceeds ρ.

What promotion writes. At each episode end, the new record, its hypotheses and their edges to the record are written in one step before any verdict is made, so that the new hypotheses can be judged at the same episode end. Hypotheses are never delivered to the actor; only promoted memories are. Promoting a hypothesis creates one promoted memory whose first wording is the hypothesis's claim and conditions, copied verbatim, and copies the hypothesis's ledger onto that wording. Since the hypothesis stated the same claim, each copied verdict keeps its audit status and counts as a verdict on the new wording in every later decision, including withdrawal. The promoted

memory keeps an edge to every hypothesis it was promoted from or merged over, each hypothesis keeps its edge to the episode that proposed it, and every *support* or *refute* verdict adds an edge to the episode it came from. These are the provenance edges of Section 3.1. Every committed change increases a version counter, which dates each wording and each verdict (Appendix A.3).

A.6 Revision: Refinement, Split, Merge and the Periodic Review

A wording is revised in three ways, and every revision follows one pattern: a model call proposes a new wording, the assessor judges the proposal again against the episodes behind the old one, and the proposal commits only if these new verdicts support it. The ways differ in what triggers them, which episodes the proposal is judged against, and what the new wording keeps from the old ledger.

Refinement and split after a refutation. A refuting verdict against a context's target (Appendix A.2) triggers a revision of that target. Neither facet takes precedence: when the two contexts have different targets and both are refuted, each target is revised separately. The editor, a model call (Appendix A.14.4), reads the refuting verdict, the memory's current wording and conditions, and the supporting verdicts in its ledger. It proposes a refinement, which rewrites the memory with stricter conditions, a split into two or more children, or nothing. The proposal is then judged again against every episode in the memory's current ledger and against the refuting episode.

A refinement commits only if every pair is resolved, at least one episode other than the refuting one supports it, and the refuting episode no longer refutes it. The new wording must thus survive the episode that refuted the old one while keeping support from another episode. Its ledger is exactly these new verdicts. A split commits only if every pair is resolved, every child keeps a supporting episode, the refuting episode supports at least one child, and no two children have the same set of supporting episodes. Each child is thus a distinct part of the old claim with evidence of its own. Each child starts with its own ledger and an edge to the parent, which is superseded. A grounded procedure is kept after a refinement if it still contains a word the new claim lacks, and it is always dropped after a split or a merge.

Merge. Merges are proposed only when a memory is newly promoted, never after a refutation or at the periodic review. The merge judge, a model call (Appendix A.14.5), reads the new memory and up to k_{merge} of the most similar promoted memories that have the same source outcome and are neither withdrawn nor superseded. It returns one group of memories that state the same claim, or nothing. The merged wording is judged again against every episode that any memory of the group was judged against. It commits only if every pair is resolved, each memory of the group has at least one episode that now supports the merged wording, and no episode refutes it. No memory is thus merged away unless some of its own evidence supports the merged wording. The memories of the group are then superseded, and the merged memory keeps an edge to each.

Periodic review. A refutation may never arrive, yet a wording can still be broader than the evidence behind it. The periodic review therefore revisits memories every T episodes without waiting for a refutation. It queues the promoted memories that are neither withdrawn nor superseded and have gathered at least k_{new} ledger entries since their last review. It ranks them by the number of these new entries, adding two for each new refutation and two if the demotion flag is raised, and reviews the top k_{sw}. The editor receives each reviewed memory with its whole ledger, reduced for each entry to the episode, the verdict, the goal, the reason and the outcome, and proposes nothing, a refinement or a split.

Only n_s supporting episodes are judged again, spread evenly over the supporting episodes sorted by episode identifier. The entries not sampled pass to the new wording with their verdicts unchanged; each is marked as inherited and keeps a reference to the wording it was recorded against. A refinement commits if no episode judged again refutes it and at least one supports it. A split judges n_s episodes again per cluster of supporting episodes and needs one support per child. Each cluster's unsampled entries pass to its child in the same way, and an episode in no cluster is carried over to no child.

What inherited verdicts may decide. Only the periodic review carries verdicts over; a revision after a refutation and a merge keep only the verdicts made when the proposal was judged again.

An inherited verdict counts in s_r and q_r like any other audited verdict, and so in ℓ_r, in the refute share that withdrawal reads and in the *disputed* label. It judged an earlier wording, however, so it is excluded from the two counts that concern the current wording itself. The count q_r^{fr} of Eqn. (6) takes only refutations recorded against the current wording, so a memory is never withdrawn on inherited refutations alone. The count s_r^{fr} of Eqn. (8) takes only support recorded after the wording was written, so a memory is never labelled well-tested on inherited support alone. Inherited evidence thus still shapes the beliefs about a memory, but it can neither withdraw the memory nor vouch for its new wording.

A.7 Withdrawal

Withdrawal takes a memory out of use once the evidence has turned against it. The condition of Eqn. (6) is checked at each episode end for every promoted memory that received a verdict at that episode end, and it reads audited verdicts only; s_r and q_r count distinct episodes. In words, a memory is withdrawn once at least n_{wd} episodes have decided on it, at least a share ρ_{wd} of them refuted it, and at least one refutation concerns its current wording. The last condition keeps a freshly revised memory from being withdrawn for the failures of the wording it replaced. A withdrawn memory is no longer recalled at any step, but nothing is deleted: its ledger stays, and it keeps receiving verdicts through the pairs of Appendix A.2. A memory replaced by a split or a merge is marked superseded instead. The log keeps the two apart, and only withdrawn memories were judged by Eqn. (6).

A.8 Recall and the Two Judges

At each step, the read operator first recalls a pool of candidate memories by semantic similarity (Eqn. (7)), and then two model calls decide whether, and in what words, each candidate is delivered. This subsection gives the inputs of these calls and the checks around them; Appendix A.9 gives the rules that combine their answers.

Search keys. The situation is written as numbered lines: the goal, the current observation and the reply to the last action, each cut to 1,200 characters, followed by the last w actions with their replies. The query formulator, a model call (Appendix A.14.7), reads these lines and returns a search key with four fields: the current situation, the decision the actor faces, the observable constraints and the effect the next action should have. It also returns a one-sentence reflection on the recent steps. When the reflection names a problem, such as effort that yields no new information, a second, *reflective* key describes that pattern in place of the situation; the first key is the *progress* key. If the answer still cannot be parsed after one correction, nothing is delivered at this step, not even procedural memory.

Similarity and recall. One frozen text-embedding model, BGE-M3 (Chen et al., 2024a), embeds every facet and every search key. Every semantic similarity in this appendix is the cosine similarity of two such embeddings. The pool of Eqn. (7) is drawn from the promoted memories that are neither withdrawn nor superseded, have at least one supporting verdict, and carry no grounded procedure in their current wording. For such a memory, $\sigma_r^{\mathrm{claim}}$ is the similarity between the key and the memory's claim facet, the embedding of its claim, source outcome and conditions. This recall is applied once per search key, first to the progress key and then to the reflective key, and the two lists are joined in that order. A memory that the reflective key recalls again is not added twice and keeps the similarities of its first recall. The floor η is applied to $\sigma_r^{\mathrm{claim}}$ alone, whereas the blend σ_r decides the order and which k_{norm} memories are kept. Among memories whose claims are equally similar to the key, the one whose conditions better match the situation therefore comes first.

Memories with a grounded procedure. A memory whose current wording carries a grounded procedure is recalled differently, once per episode and against the goal instead of a search key; such memories form the term $\mathcal{R}_i^{\psi}(g)$ of Eqn. (7). They must pass the same floor η on the similarity between the goal and their claim facet, which for these memories also embeds the grounded procedure. They are then ranked by the similarity between the goal and their grounded procedure, which serves as their σ_r, and the top k_{proc} are kept. Every candidate therefore has a σ_r for the later stages, even one that no search key recalled.

The applicability judge. The applicability judge, a model call (Appendix A.14.8), reads one candidate at a time: its claim, effect, conditions and grounded procedure, together with the search keys of this step and the situation lines. A memory is *contested* when its audited ledger holds at least one refutation, the same condition under which the grade gives *disputed*; the applicability judge then also sees the memory's audited support and refute counts. It answers either that the claim applies as written, returning a compact restatement in the claim's own words for delivery, or that the memory's episodes must be read first. The counts can change only how a recalled memory is delivered, since recall has already happened.

The evidence judge. When the applicability judge asks for the episodes, the episodes judged against the memory's current wording are ranked by the similarity of their evidence facet to the search key. Three are shortlisted, taking first the episodes whose verdicts cite milestone steps and then the others. A candidate with no verdict that cites a milestone step is dropped, and so is one whose cited steps can no longer be found in the stored episodes. The evidence judge, a second model call (Appendix A.14.9), sees the three episodes with their goals and outcomes, the cited milestone steps copied in as text it may quote, and, for a contested memory, the same counts. It either returns a delivery that those steps support, citing at least one of them, or withholds the memory. When the candidate carries a grounded procedure, the mechanism appends the procedure to whatever either judge wrote.

The check on what a judge writes. Both judges rewrite a memory in their own words, so what they write is checked before delivery. The text must not fold a second candidate into it, which would show as the prefix that marks a delivered item appearing more than once. It must also contain none of six phrases that claim more support than the verdicts show: `many episodes`, `all episodes`, `numerous episodes`, `always succeeds`, `always works` and `guaranteed`. A delivery that fails the check is refused and its candidate is dropped at this step; a refused answer of the applicability judge is not passed on to the evidence judge. An answer that cannot be parsed is treated the same way. A withholding or a refusal is not a verdict and never enters the ledger.

A.9 The Read Operator in Full

After recall, the read operator μ applies four rules in turn. It decides the form in which each candidate is delivered, removes repeated deliveries, decides what to deliver when nothing is left, and assembles the block. The two judges of Appendix A.8 settle the first rule; the other three are fixed rules.

The delivered form. A candidate recalled from a similar situation may still not apply as written, and its episodes may say more, or less, than its wording. Each candidate is therefore delivered in at most one form:

$$\mathrm{deliver}(r) = \begin{cases} \textit{Trust}(r) & A(r) \wedge P(r), \\ \textit{Trace}(r) & \neg A(r) \wedge K(r) \wedge P(r), \\ \varnothing & \text{otherwise}, \end{cases} \tag{11}$$

where $\mathrm{deliver}(r)$ is what candidate r yields at this step, $\textit{Trust}(r)$ is a compact restatement of r's claim, $\textit{Trace}(r)$ is a sentence grounded in the milestone steps its verdicts cite, and $\varnothing$ is no delivery. Here $A(r)$ holds when the applicability judge finds r applicable as written, $K(r)$ when the evidence judge keeps r on the milestone steps its verdicts cite, and $P(r)$ when the delivery passes the check of Appendix A.8. In words, a memory the applicability judge accepts is delivered as a short restatement, and one it doubts is delivered only if its own cited steps support it. A doubted memory is therefore dropped at this step when none of its verdicts cites a milestone step or when the evidence judge withholds it, and any memory is dropped when the check refuses its text.

The procedure-only form. The execution belief adjusts the form alone. A memory with a grounded procedure is delivered as that procedure alone, in either form, once at least n_{gap} of its adopted deliveries have a known execution result and the refused share ν_r among them reaches ρ_{gap}. The actor has then repeatedly failed to turn the memory's general wording into an action the environment accepts, whereas the procedure quotes actions that an episode actually took (Appendix A.4).

Removing repeats. Each judge sees one candidate at a time, so neither can notice that two candidates say the same thing, or that one memory has been delivered for several steps in a row. What the judges kept is therefore thinned:

$$\begin{aligned} \mathcal{J}_t &= \{\, r \in \mathcal{C}_t \;:\; \mathrm{deliver}(r) \neq \varnothing \,\}, \\ \mathrm{thin}(r) \iff\; & r \text{ was delivered at each of the last } k_{\mathrm{con}} \\ & \quad \text{steps of this episode} \\ & \vee\ \ \sigma_{r,r'} \geq \eta_{\mathrm{dup}} \ \text{ for some } r' \in \mathcal{J}_t \\ & \quad \text{with } \sigma_{r'} > \sigma_r, \\ \mathcal{D}_t &= \{\, \mathrm{deliver}(r) \;:\; r \in \mathcal{J}_t,\ \neg\,\mathrm{thin}(r) \,\}, \\ \mathcal{F}_t &= \mathcal{C}_t \setminus \{\, r \in \mathcal{J}_t \;:\; \mathrm{thin}(r) \,\}, \end{aligned} \tag{12}$$

where $\mathcal{J}_t$ holds the candidates the judges kept, $\mathrm{thin}(r)$ marks those removed at this step, and $\sigma_{r,r'}$ is the semantic similarity between the claim facets of r and r'. The first condition makes a memory give way after k_{con} consecutive steps of delivery, counted whether or not the actor acted on it. The second keeps, of two memories whose stored claims are near-duplicates, only the one with the higher σ_r, that is, the one closer to the situation; it compares the stored claims, not the texts the judges wrote. The delivery set $\mathcal{D}_t$ holds what survives. The fallback pool $\mathcal{F}_t$ holds every candidate that the thinning did not remove, including those a judge withheld. A memory removed as a repeat is thus barred from the fallback too, which is why a step can end with both sets empty.

The fallback. When $\mathcal{D}_t$ is empty, the step falls back to:

$$\mathrm{fallback}(t) = \begin{cases} \mathrm{narratives}(\mathcal{E}_i) & \mathcal{C}_t = \varnothing, \\ \{\, \mathrm{weak}(r^\star) \,\} & \mathcal{F}_t \neq \varnothing, \\ \varnothing & \text{otherwise}, \end{cases} \tag{13}$$

where $r^\star = \arg\max_{r \in \mathcal{F}_t} \sigma_r$ is the fallback pick. The operator weak delivers the pick's stored wording under *weak match*, the one reliance label that no ledger decides, and $\mathrm{narratives}$ returns the recorded narratives of past episodes, under no label. In words, when nothing was recalled, the actor is provided with what happened in the most similar past episodes; when candidates were recalled but none survived, it is provided with the most similar one, marked as a match that no judge approved. The pick may be a candidate a judge declined at this step, since a labelled weak lead is preferred to an empty block. In either case no memory approved by a judge is in front of the actor, which then explores under its own policy π; this is the *Explore* form.

How the fallback items are built. When no candidate was recalled, as in the early episodes of a stream before any memory is promoted, the k_{cold} episodes whose evidence facet is most similar to the first search key contribute the narratives of their milestone steps. Each narrative is delivered as a plain line, with no reliance label and no usage record. The memory starts empty, holding nothing in any of the six parts of Eqn. (1), so in the first episode of the stream there is no episode to draw on yet.

The weak-match item is built from the stored wording of $r^\star$ by a fixed template: the claim, whether this pattern led toward success or toward failure in past episodes, the grounded procedure if there is one, and the conditions under which the memory was recorded. It carries the weak-match label, worded differently when the pick has a grounded procedure (Appendix A.10), and a usage record is written for it. A candidate recalled on the goal competes for the pick with the similarity between the goal and its claim facet instead of its σ_r. If the fallback pool is empty too, only procedural memory is delivered.

The block. Finally, the block is assembled. The reliance label orders it, since similarity alone would put the memory from the most similar situation ahead of the best-supported one. After the ranked memories, the block carries the part of procedural memory that matches the goal, $\mathcal{P}_i(g)$, namely its workflows and index leads (Appendix A.11). The block μ returns is:

$$\begin{aligned} m_t &= \mathrm{ord}_{\,\gamma(r) \succ \varepsilon_r \succ \sigma_r}\big(\mathcal{B}_t\big) \,\|\, \mathcal{P}_i(g), \\ \mathcal{B}_t &= \begin{cases} \mathcal{D}_t & \mathcal{D}_t \neq \varnothing, \\ \mathrm{fallback}(t) & \text{otherwise}, \end{cases} \end{aligned} \tag{14}$$

Table 3: The fixed prefix that begins each kind of delivered line.

Delivered line	Prefix
well-tested memory	`Memory [well-tested]:`
tentative memory	`Memory [tentative --- limited evidence; treat as a hypothesis to test]:`
disputed memory	`Memory [disputed --- past episodes have contradicted this; weigh it against what you observe]:`
weak match	`Memory [weak match --- low confidence, for reference only]:`
weak match with a grounded procedure	`Memory [unverified rule --- concrete steps quoted from a past episode]:`
narrative of a past episode	`Memory:`
workflow	`Memory [recipe from a past episode]:`
index leads	`Observation index (counts from earlier episodes in this run, not a statement about the current room):`

where ord sorts the items of $\mathcal{B}_t$ by the memory behind each: first by reliance label, in the order *well-tested*, *tentative*, *disputed*, *weak match*, then by the adoption belief ε_r and then by σ_r. Narratives, which have no memory behind them, keep their order, and $\|$ appends. In words, the best-supported memories come first; among memories with the same label, those the actor has adopted more readily come first; and among those, the ones closest to the situation come first. Apart from this order, the actor is provided with only the delivered text of each memory and its reliance label.

A.10 The Well-Tested Check and the Block Format

The *well-tested* label tells the actor that the evidence covers what it is reading. The check c_r of Eqn. (8) verifies this in three ways:

$$c_r = x_r \wedge \omega_r \wedge \neg\, b_r, \tag{15}$$

where x_r is the action-taken mark, ω_r the wording check and b_r the demotion flag of Appendix A.4. The mark x_r holds when a supporting verdict cites a milestone step whose action was taken, that is, a step whose recorded text holds both an action and the environment's reply to it. The check ω_r holds when every content word of the text to be delivered occurs in the memory's own text, the goal or the situation lines. In words, the action must have been seen taken, the words delivered must not go beyond what was judged, and the memory's conditions must not have often failed where it was delivered.

The wording check. The memory's own text is its claim, grounded procedure and conditions, and words are matched tolerantly of inflection. A delivery that passes names nothing that was not already in the memory or in front of the actor; one that fails makes c_r of Eqn. (15) false and is delivered under a weaker reliance label. The wording check and the demotion flag catch different faults: ω_r reads the text of this step, and b_r the record of earlier steps.

Order and length. The order of Eqn. (14) breaks any remaining tie by the memory's identifier. Deliveries are added in that order while their total length stays within the budget n_{tok}; one that does not fit is skipped, and the deliveries after it are still tried. The workflow lines are added under the same budget.

Prefixes. Each delivered line begins with a fixed prefix that the mechanism writes (Table 3). The prefixes of *tentative* and *disputed* memories carry a caution, and a weak match that carries a grounded procedure is marked as an unverified rule whose steps were quoted from a past episode. The block opens with `Memory context (from completed episodes):` and separates items by a line `---`. It reads `Memory: (none yet)` only when the read returns no line at all, not even from procedural memory.

A.11 PROCEDURAL MEMORY

Procedural memory keeps concrete, action-level knowledge that the claims of $\mathcal{R}_i$ do not state. It holds three kinds of content, each with its own key: the grounded procedures of promoted memories, workflows for recurring sub-goals, and an observation index of the actions that revealed each word. The workflows and index leads that match the goal are appended to the block after the ranked memories (Eqn. (14)), each labelled with where its text came from rather than with a reliance label.

Grounded procedures. The grounded procedures are those admitted by Eqn. (10). Each is embedded on its own, and a promoted memory that carries one is ranked by that embedding against the goal, rather than recalled by a search key (Appendix A.8).

Workflows. A workflow is a short sequence of actions the agent took for a recurring sub-goal: a header naming the sub-goal and at least two lines, each an action and the reply it received. In every check on workflows, the list of function words also holds the words of the workflow format itself, such as `step` and `goal`. The workflow extractor, a model call (Appendix A.14.6), is called once per episode, for successful and failed episodes alike, on the trajectory written as such lines with each reply cut to 150 characters, and returns at most n_R workflows.

A workflow is admitted on two conditions. First, its coverage cov_R, the share of its step words that occur anywhere in the episode's own actions and replies, must reach κ_R. The share is counted over content words, after every placeholder the workflow extractor was asked to fill in is deleted and every bare number dropped. Second, its text, compared with case and spacing normalized, must not duplicate any stored workflow, including one admitted earlier from the same episode. A workflow written entirely in placeholders and function words has no word to check and is refused.

When a workflow stops being delivered. Two counters follow each stored workflow. A workflow delivered in an episode counts as *taken* when all its informative steps were taken in order at or after its first delivery; a step is informative when it still names a content word after its instance-specific parts are removed. For a taken workflow, a successful episode adds one to its confirmations and a failed one to its refutations. A confirmation, however, needs at least two informative steps, so that one action the actor had its own reason to take cannot confirm a one-step workflow. A workflow stops being delivered once its refutations reach one, for a workflow drawn from a failed episode, or two, for one drawn from a successful episode, and are at least twice its confirmations. Like the other one-way gates, this rule can only stop a workflow from being delivered.

The observation index. The index records which actions revealed each word. Here a word counts only if it is alphabetic, at least three characters long and not a function word. When a word first appears in an environment reply during an episode, it is credited, once per episode, to the generalized action after which it appeared; a word already visible in an earlier observation is credited to no action. An action is generalized by folding case and deleting every whitespace-separated run of digits, so that the instance numbers an environment attaches to otherwise identical objects collapse. The credited actions are the index's *leads*: `go to shelf 1` and `go to shelf 2` become one lead. Beside the index, a goal-word table counts, once per goal, each content word of each goal. Both are counted by fixed rules at the episode end, before the workflow extractor is called.

What procedural memory delivers. At read time, the index answers with the leads for the words of the current goal. Goal words are taken with a wider list of function words, so that a preposition in a goal cannot become a key. For each goal word, the top k_{idx} leads are delivered, each with at least n_{idx} episodes behind it. Once n_{goal} goals have been seen, a word that appears in a share ρ_{goal} or more of them is dropped, since such a word separates nothing. The k_{rec} workflows most similar to the goal are delivered, with the second place reserved for a workflow that offers a step verb the actor has not used and the first workflow lacks, provided its own similarity reaches η_{div} and lies within Δ_{div} of the first. The index line and the workflow lines are delivered as lines of their own, labelled as Appendix A.10 lists. Since workflows record actions the agent took and the index records words it has seen, procedural memory offers nothing for a sub-goal the agent has never reached.

A.12 THE LOOP AS AN ALGORITHM

Algorithm 1: MCMEM: reading memory within an episode and maintaining it at the episode end

Require: task stream $\tau_1, \ldots, \tau_N$; frozen model f_θ; thresholds $(\rho, \rho_{\mathrm{wd}}, n_{\mathrm{wd}}, \ell_{\min}, n_{\min}, \rho_{\mathrm{gap}}, n_{\mathrm{gap}}, T)$
Ensure: memory M_{N+1}

Initialize $M_1 \leftarrow (\mathcal{E}_1, \mathcal{R}_1, \Lambda_1, \mathcal{U}_1, \mathcal{H}_1, \mathcal{P}_1)$, all empty
for $i = 1$ **to** N **do**
  // Read operator μ at every step (Section 3.2.3)
  for each step t of episode ξ_i until τ_i ends **do**
    $Q_t \leftarrow \textsc{QueryFormulator}(g, o_t, \xi_t^w)$
    $\mathcal{C}_t \leftarrow \textsc{Recall}(Q_t, g, M_i)$ // Eqn. (7)
    $\mathcal{D}_t \leftarrow \varnothing$
    for all $r \in \mathcal{C}_t$ **do**
      if $\textsc{ApplicabilityJudge}(r, g, o_t, \xi_t^w)$ finds r applicable as written **then**
        $\mathcal{D}_t \leftarrow \mathcal{D}_t \cup \{\textit{Trust}(r)\}$
      else if $\textsc{EvidenceJudge}(r)$ keeps r **then**
        $\mathcal{D}_t \leftarrow \mathcal{D}_t \cup \{\textit{Trace}(r)\}$
      end if
    end for
    deliver its grounded procedure alone for each r with one, $n_r^{\mathrm{ex}} \geq n_{\mathrm{gap}}$ and $\nu_r \geq \rho_{\mathrm{gap}}$
    remove the deliveries thin marks // Eqn. (12)
    if $\mathcal{D}_t = \varnothing$ **then**
      $\mathcal{D}_t \leftarrow \mathrm{fallback}(t)$ // Eqn. (13); the actor explores
    end if
    $m_t \leftarrow \mathrm{ord}(\mathcal{D}_t) \,\|\, \mathcal{P}_i(g)$ // Eqn. (8), Eqn. (14)
    $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup \{\text{a usage record per memory in } \mathcal{D}_t\}$
    $a_t \sim \pi(\cdot \mid g, o_t, \xi^{<t}, m_t)$
  end for
  // Write operator: $M_i^+ \leftarrow \phi(M_i, \xi_i, y_i)$
  add $\textsc{Summarizer}(\xi_i)$ to $\mathcal{E}_i$ and $\textsc{HypothesisGenerator}(\xi_i, y_i)$ to $\mathcal{H}_i$
  add $\textsc{WorkflowExtractor}(\xi_i)$ to $\mathcal{P}_i$; update its counters and index from (ξ_i, y_i)
  // Maintenance operator: $M_{i+1} \leftarrow \zeta(M_i^+)$ **(Section 3.2.1, Section 3.2.2)**
  for all pairs (c, e) of a stored claim and a semantically similar episode **do**
    $\Lambda_i \leftarrow \Lambda_i \cup \{v(c, e)\}$ // assessor, Eqn. (3)
  end for
  for all $h \in \mathcal{H}_i$ with $\mathrm{promote}(h)$ **do**
    move h into $\mathcal{R}_i$ // Eqn. (5)
  end for
  for all r refuted by ξ_i, and every T episodes r chosen by the periodic review **do**
    commit $\textsc{Editor}(r)$ if the assessor, judging it again, finds it supported
  end for
  for all r promoted at this episode end **do**
    commit $\textsc{MergeJudge}(r)$ if the assessor, judging it again, finds it supported
  end for
  for all $r \in \mathcal{R}_i$ with $\mathrm{withdraw}(r)$ **do**
    withdraw r // Eqn. (6)
  end for
  mark each usage record of ξ_i with $\textsc{AdoptionJudge}(\xi_i)$ and whether the action was accepted
  re-file as *not tested* this episode's refutations of each memory whose every adopted action in ξ_i was refused
end for
return M_{N+1}

Algorithm 1 gathers the whole loop over the task stream: the read operator μ at every step of an episode, then the write operator ϕ and the maintenance operator ζ at its end. Each helper is a model call or a fixed rule described in Section 3 or in this appendix, and a comment names the display a line applies, where there is one.

A.13 COST

Every model call of the memory mechanism goes to the same frozen model the actor runs on, at temperature zero. A call whose answer cannot be parsed is retried once, and these retries are not counted in what follows. At each step, the mechanism makes one query-formulator call, one

applicability call per candidate, and one evidence call per candidate that was sent to its episodes and has a citable step there; a step that recalls nothing costs the query call alone. A step has at most k_{norm} candidates per search key, plus k_{proc} recalled on the goal. It embeds one search key, or two when a reflective key was written, and reuses one goal embedding per episode.

At each episode end, the mechanism makes one summarizer call, one hypothesis-generator call, one workflow-extractor call, one assessor call per pair, and one adoption-judge call for every eight memories delivered in the episode. A refutation of a target adds one editor call and one assessor call per pair judged again, and a promotion adds one merge-judge call with the same re-judging. Every T episodes, the periodic review adds up to k_{sw} editor calls, each followed by its re-judging. The episode end embeds two facets per record, up to three per new wording and one per workflow. Nothing in the memory is ever deleted: episodes, hypotheses, ledger entries, edges, usage records, workflows and index entries all accumulate, and a promoted memory that is withdrawn or superseded changes a flag, not its record.

A.14 PROMPTS

The prompts of the calls made at the episode end come first, followed by those of the calls made at every step. Each is reproduced exactly as the mechanism sends it, with two flaws left in place: the periodic review's prompt breaks off in the middle of a sentence, and the assessor's prompt asks for a sixth answer field that its output format does not list.

The prompts use the implementation's names, which map to the paper's as follows. An episode's *key evidence* is its milestone steps; a *recipe* is a workflow; an *abstraction* or *rule* is a promoted memory; and a *sweep* is the periodic review. The first-wave and second-wave judges are the applicability judge and the evidence judge. Their *direct* and *keep* answers give the *Trust* and *Trace* forms of Eqn. (11), and a *traceback* answer sends a candidate to the evidence judge. The editor's *refine*, *split* and *noop* are a refinement, a split and no change. A number that appears in a prompt is the value of the constant the mechanism passes into the call. Each call requires a fixed output format, and when an answer does not follow it, the correction call quotes that format back to the model.

A.14.1 EPISODE SUMMARIZER

```
You are an Episode Summarizer.
Your mission is to extract key steps from a completed agent trajectory
     into a structured Episode record.

## Guidelines
- Output format: a single JSON object with these fields:
  {
    "key_evidence": [
      {
        "step_refs": ["<step_id>", ...],
        "content": "<string>"
      },
      ...
    ]
  }

- step_refs: a list of IDs of the key steps that mark the milestones
    of the trajectory, in trajectory order.
- content: a concise statement that illustrates how these key steps
    gradually lead to the terminal outcome of the provided episode

## Constraints
- Use only events from the supplied trajectory for key_evidence.
```

A.14.2 HYPOTHESIS GENERATOR, SUCCESS FORM

```
You are a Hypothesis Generator.
Your job is to distill from this successful trajectory potentially
    reusable candidate memory that could be later validated into a
    general memory.
```

```
Each hypothesis must declare one of two kinds:
- "workflow": a reusable sequence of action roles that, when followed, tends to produce a class of result.
- "strategy_pattern": a reusable decision once the agent tried certain routes or reaches a certain kind of situation.
## Guidelines
- Output format: a single JSON object with these fields:
  {
    "hypotheses": [
      {
        "kind": "workflow" | "strategy_pattern",
        "grounded_procedure": "<string, see constraints; empty string for any other kind>",
        "claim": "<string, a tentative reusable claim in general roles/categories>",
        "applicability_conditions": ["<string>", ...] - at most 3 concise, GENERAL observable conditions (roles/categories) under which this claim applies; do not enumerate every absent object, location, or unperformed action,
        "evidence": [
          {
            "step_ref": "<string, e.g. step_0001, one step ID from this trajectory>"
          },
          ...
        ]
      },
      ...
    ]
  }

## Constraints
- Generality rule: claim and applicability_conditions must describe GENERAL, observable roles, categories, or relations - never the specific object names, creature names, or instance-specific values of THIS episode.
- Generate at most 4 distinct hypotheses. Aim for a mix of both kinds when the trajectory supports it; otherwise emit only the kinds it supports.
- Each evidence item must reference one real step_ref (e.g. step_0001) that actually appears in this trajectory; do not invent step ids.
- Do not repeat a step_ref within one hypothesis.
- Each hypothesis must use at most 3 applicability_conditions and at most 3 evidence items.
- Each output claim must be semantically different from the others.
- Do not invent steps, actions, or outcomes that were not actually observed in this trajectory.
- grounded_procedure is REQUIRED and must be non-empty for every hypothesis of kind "workflow". The generality rule above does NOT apply to this one field: it is the only place where the episode's own concrete names are kept, and a generalized grounded_procedure is discarded.
- How to write it for kind "workflow": first look at the steps you listed in evidence, and copy their action text word-for-word; then join those copied phrases with as few words of your own as the sentence needs. Keep every object name, location name and number exactly as the step wrote it. Do not name an object, place, or outcome that does not appear in the cited steps or in the goal. A restatement built mostly out of your own wording is discarded.
- For every other kind, grounded_procedure must be the empty string.
```

The failure form differs only in the kinds it may emit and in the field that must carry the grounded procedure:

```
You are a Hypothesis Generator.
```

```
Your job is to distill from this failed trajectory potentially reusable candidate memory that could be later validated into a general memory.
Each hypothesis must declare one of two kinds:
- "failure_lesson": a step (or a sequence of steps) that appeared unproblematic at the time but actually contributed to the eventual failure.
- "guardrail": a reusable rule "once a certain kind of situation appears, do NOT do X afterward" - both the triggering situation and the discouraged action stated as general roles/categories.
[the Guidelines block above, with "kind": "failure_lesson" | "guardrail"]
[the Constraints block above, with "guardrail" in place of "workflow"]
```

A.14.3 EVIDENCE ASSESSOR

```
You are an Evidence Assessor.
Your mission is to judge whether one completed Episode supports, refutes, or does not test one subject claim, by comparing the claim against the Episode's causal evidence path (the key steps that led to its terminal outcome), using only observable facts from that Episode.

## Guidelines
- Answer the sub-questions in order, then derive the verdict mechanically from q5_match.
- Output format: a single JSON object with these fields:
  {
    "q1_claim_action": "<the concrete action or procedure the claim is about, and whether the claim RECOMMENDS it or DENIES it>",
    "q2_claim_prediction": "<the observable result the claim predicts when that action is taken under its stated conditions>",
    "q3_path_steps": ["<step_ref>", ...],
    "q4_observed": "<what actually happened at those steps, quoted>",
    "q5_match": "consistent" | "contradicted" | "not_tested",
    "verdict": "supports" | "refutes" | "neither",
    "evidence_refs": ["<step_ref>", ...],
    "reason": "<concise explanation grounded in the cited steps>"
  }
- q3_path_steps: the steps where the claim's action is actually executed under its stated conditions; [] if that never happens.
- q5_match: "not_tested" if q3_path_steps is []; "consistent" if the observed result matches q2; "contradicted" if it conflicts with q2.
- verdict follows q5_match mechanically: consistent -> "supports", contradicted -> "refutes", not_tested -> "neither".

## Constraints
- Merely holding an object or being in a matching situation does not test a claim about an action; without execution the verdict is "neither".
- Steps anywhere on the causal path count, not only the terminal step.
- Outcome-mediated test: if the claim DENIES that an action can accomplish the goal and the Episode SUCCEEDED with that action as a key step, q5_match MUST be "contradicted" and the verdict MUST be "refutes", even when no single observation shows the denied information directly --- the successful terminal outcome itself is the test result.
- Judge only from the quoted steps; world knowledge about what should work is forbidden.
- "supports" or "refutes" must cite step refs in evidence_refs; "neither" must have an empty evidence_refs array.
- q6_not_tested_reason: when q5_match is "not_tested", say which kind of non-test this was. "conditions_not_met": the claim is about this Episode's subject matter, but the applicability conditions it
```

```
        states for itself never held here. "unrelated": the claim is
       about something this Episode is not about at all. When q5_match is
        not "not_tested", answer "not_applicable".
```

A.14.4 EDITOR (REFINEMENT AND SPLIT PROPOSER)

```
1  You are a Memory Graph Editor.
2  Your mission is to decide whether a target Abstraction that was just
       refuted by the current Episode should be refined (REFINE_CANDIDATE
       ), split into child abstractions (SPLIT_CANDIDATE), or left
       unchanged (NOOP), by comparing the older supporting evidence
       against the new refuting evidence.
3
4  ## Guidelines
5  - Output format: a single JSON object with these fields:
6    {
7      "operation": "REFINE_CANDIDATE" | "SPLIT_CANDIDATE" | "NOOP",
8      "proposal": { --- required only for REFINE_CANDIDATE. Its
           source_outcome is fixed to the target abstraction's current
           source_outcome; do not restate it.
9        "claim": "<string, the revised conditional rule statement>",
10       "applicability_conditions": ["<string>", ...] --- narrowed
             observable conditions that the current refuting Episode no
             longer satisfies
11     },
12     "children": [ --- required only for SPLIT_CANDIDATE, at least 2
           items. Each child's source_outcome is fixed to the parent
           target's current source_outcome; do not restate it.
13       {
14         "claim": "<string, the child rule statement>",
15         "applicability_conditions": ["<string>", ...] --- observable
               conditions specific to this child
16       },
17       ...
18     ],
19     "reason": "<string, why this operation and these edits were chosen
           over the alternatives>"
20   }
21 - How to choose the operation (all conditions must satisfy the
       Generality rule below):
22   - REFINE_CANDIDATE: narrow the applicability_conditions by adding a
         condition that was present in supporting Episodes but absent or
         contradicted in the current refuting Episode. After refinement,
         the current Episode must assess as "neither" or "supports",
         never "refutes".
23   - SPLIT_CANDIDATE: partition the target into at least two children
         such that the current Episode SUPPORTS at least one child (the
         sub-pattern that still holds), while another child isolates the
         refuted sub-pattern. Every child must keep at least one
         supporting Episode from the ledger, and no two children may
         share the same supporting Episode set.
24   - NOOP: use only when the visible Episode facts are too weak to name
          a revised condition or child split --- prefer a concrete REFINE
          or SPLIT whenever a named observable distinction separates
         support from refutation.
25 - Factual evidence must come from Episodes.
26
27 ## Constraints
28 - Generality rule: every applicability_condition, in the proposal or
       in any child, must be a GENERAL observable role, category, or
       relation - never the specific object names, creature names, or
       instance-specific values of THIS episode. Narrow the rule's scope
       without dropping to episode-specific vocabulary.
```

```
- applicability_conditions are matching predicates evaluated MID-EPISODE, BEFORE acting: each must be verifiable as true/false from the agent's current observations at decision time.
- Write conditions as observable STATE, never as success/failure judgments. In-episode feedback the agent can see (an error message, an ambiguity prompt, an unchanged observation after an action) IS a legitimate observable; the episode's final result is NOT available at decision time and must never appear as a condition. Rephrase judgments into state: 'the action failed' -> 'the environment returned an error message for the action'; 'successfully placed X into Y' -> 'X is inside Y'.
- Never put task family, dataset name, or action aliases into any field.
- Do NOT invent or output action procedures. Abstractions are conditional rules, not executable plans.
- Do NOT output prose, markdown, or any text outside the JSON object.
```

The periodic review sends the same role a different mission, over one promoted memory and its whole ledger rather than over one refutation:

```
You are a Memory Graph Editor.
Your mission is to review one Abstraction together with its accumulated evidence ledger and decide whether the rule should be split into condition-incompatible children (SPLIT_CANDIDATE), tightened into a strictly narrower rule that still covers all of its supporting Episodes (REFINE_CANDIDATE), or left unchanged (NONE).

## Guidelines
- Output format: a single JSON object with these fields:
  {
    "action": "NONE" | "REFINE_CANDIDATE" | "SPLIT_CANDIDATE",
    "proposal": {
      "claim": "<string>", - the tightened conditional rule statement
      "applicability_conditions": ["<string>", ...] --- a strictly narrower set of observable conditions that every supporting Episode still satisfies
    },
    "clusters": [
      {
        "episode_ids": ["<episode_id>", ...], --- at least two episode ids, each taken ONLY from evidence_ledger entries whose verdict is "supports"; never reuse an id in more than one cluster
        "claim": "<string>", - the child rule statement
        "applicability_conditions": ["<string>", ...] --- observable conditions specific to this cluster
      },
      ...
    ],
    "reason": "<string> - why this action and these edits were chosen over the alternatives"
  }
- "proposal": required only for REFINE_CANDIDATE.
- "clusters": required only for SPLIT_CANDIDATE, at least 2 items.
- "claim" is the rule's causal statement - what behavior pattern leads to what consequence;
- "applicability_conditions" are matching predicates evaluated MID-EPISODE, BEFORE acting: each must be verifiable as true/false from the agent's current observations at decision time.Write conditions as observable STATE, never as success/failure judgments. In-episode feedback the agent can see IS a legitimate observable.
- How to choose the action:
  - SPLIT_CANDIDATE: choose it when the supporting Episodes separate into two or more clusters that satisfy mutually incompatible
```

```
      observable conditions, so a single rule blurs distinct sub-
      patterns. Each cluster must draw at least two episode ids that
      appear in the evidence_ledger with verdict "supports" (ignore "
      refutes" and "neither" entries when picking ids), and no two
      clusters may share an episode id.
  - REFINE_CANDIDATE: choose it when ALL supporting Episodes actually
      satisfy a strictly narrower observable condition than the
      current claim states, so the rule can be tightened without
      dropping any support.
  - NONE: choose it when neither SPLIT_CANDIDATE nor REFINE_CANDIDATE
      is suitable.

## Constraints
- Generality rule: every applicability_condition, in the proposal or
    in any cluster, must stay a GENERAL observable role, category, or
    relation - never the specific object names, creature names, or -
    Never put task family, dataset name, or action aliases into any
    field.
- Do NOT invent or output action procedures. Abstractions are
    conditional rules, not executable plans.
- Do NOT output prose, markdown, or any text outside the JSON object.
- Output discipline: keep "reason" to at most 3 sentences and put no
    step-by-step deliberation inside any field. Emit only the final
    decision so the JSON object stays short and complete.
- Prefer NONE over inventing a distinction the ledger does not support
    .
```

A.14.5 MERGE JUDGE

```
You are a Consolidation Judge.
Your mission is to decide whether the current Abstraction and any of
    its retrieved candidates express the same conditional rule and
    should be merged into one.

## Guidelines
- Output format: a single JSON object with these fields:
  {
    "operation": "MERGE" | "NOOP" --- whether to merge or not,
    "merge_groups": [ --- required for MERGE; for NOOP use []
      {
        "source_abstraction_ids": ["<id>", ...] --- IDs selected ONLY
            from the supplied current + candidate list, must include
            current_abstraction.id and at least one other,
        "merged_abstraction": { --- its source_outcome is fixed to the
             shared source_outcome of the sources; do not restate it.
          "claim": "<string, the unified rule that subsumes all
              sources>",
          "applicability_conditions": ["<string>", ...] --- shared
              observable conditions
        },
        "reason": "<string, why these sources express the same rule>"
      }
    ],
    "reason": "<string, overall rationale for MERGE or NOOP>"
  }
- Return MERGE only when a subset (including the current Abstraction)
    expresses the same conditional rule with compatible observable
    conditions. Sources must share the same source_outcome.
- Return NOOP when the abstractions are genuinely different rules.

## Constraints
- Do NOT output prose, markdown, or any text outside the JSON object.
- Vector similarity only proposes candidates --- you must judge
    semantic equivalence of the rules themselves, not surface word
    overlap.
```

A.14.6 WORKFLOW EXTRACTOR

```
You read a recorded trajectory of an agent and write down the reusable sub-routines it contains. You copy wording from the trajectory instead of inventing your own.
Below is one recorded trajectory. Each numbered line is an action the agent issued and the reply it received.

Task goal: {goal}

Trajectory:
{trajectory}

Episode outcome: {outcome}

Write down at most {max_recipes} reusable sub-routines contained in this trajectory.

Rules:
- Each sub-routine must achieve a distinct sub-goal, and the sub-goals must not overlap.
- Each sub-routine must contain at least two steps.
- Keep the invariant elements exactly as they appear above: the verbs of the actions, and the names of the fixed facilities and fixed places the agent interacted with. Copy them character for character.
- Replace the parts that are specific to this particular instance, such as the particular items involved and their numeric identifiers, with a short descriptive variable name inside braces, for example {the target item}.
- Do not invent steps or capabilities that the trajectory does not show. A sub-routine may be taken from a part of the trajectory that worked even if the episode as a whole did not.

Write only the sub-routines, in exactly this format and nothing else:

[Recipe] Goal: <what this sub-routine achieves>
1. <action pattern> -> <what this step is for>
2. <action pattern> -> <what this step is for>
```

A.14.7 QUERY FORMULATOR

```
You are a Query Formulator with meta-cognitive duties.
Your mission is to turn the actor's situation into retrieval queries that will be embedded and used to search relevant memory.  A good query set covers what the actor might be getting wrong.

## Meta-cognitive principles
Before writing queries, examine the goal, the current observation, and the actor's own recent action/result history, and ask yourself:
1. Which thread of progress is worth advancing next, given the goal and what has already been accomplished?
2. Does the recent exploration itself show signs of a problem --- e.g. effort that yields no new information, feedback that contradicts expectations, or a possibly wrong assumption?  Judge this from the history; do not assume a problem exists if the exploration looks productive.

## Output format
A single JSON object:
  {
    "progress_query": {
      "current_situation": "<one short sentence: what is directly observable now>",
      "decision_need": "<one short sentence: the action choice the actor faces right now>",
```

```
      "observable_constraints": ["<string>", ...],
      "desired_effect": "<one short sentence: the effect the next action should produce>"
    },
    "reflection": "<one short sentence: what in the recent exploration looks problematic or uncertain, or exactly 'none'>",
    "reflective_query": <null if reflection is 'none', else same 4-field object, where current_situation describes the problematic pattern in the actor's own recent behavior, decision_need names what the actor should reconsider, and desired_effect states what escaping/resolving the problem would look like>
  }

## Constraints
- Base every judgment only on the goal, observations, action feedback, and action/result history given to you.
- Do NOT output prose, markdown, or any text outside the JSON object.
- Keep each field concise --- queries are search keys, not analyses.
- Do NOT include a suggested_action field.
```

A.14.8 APPLICABILITY JUDGE

```
You are a Retrieval First-Wave Judge.
Your mission is to decide whether one candidate abstraction is directly applicable to the actor's current situation, or requires deeper inspection of its supporting episode evidence.

## Guidelines
- Output format: a single JSON object with these fields:
  {
    "query_id": "<string, the query identifier>",
    "abstraction_id": "<string, the candidate abstraction ID>",
    "active_revision": <int, the revision number of the candidate>,
    "decision": "DIRECT" | "TRACEBACK" --- whether the abstraction applies as-is or needs evidence inspection,
    "actor_evidence": "<string, compact reworded guidance> | null --- required for DIRECT, must be null for TRACEBACK,
    "reason": "<string, brief explanation of the decision>"
  }
- decision DIRECT: the abstraction's claim and effect apply to the current situation as written. actor_evidence must be a compact reworded string preserving the original effect and semantic direction. Select and compress only grounded content from the abstraction; do not invent new applicability conditions, do not inflate support, and do not merge multiple abstractions into one evidence string.
- decision TRACEBACK: the abstraction is potentially relevant but you need to inspect the supporting or refuting episode evidence before deciding. actor_evidence must be null.
- A candidate may carry a "contested" field with the number of past Episodes that supported and refuted it. Treat it as a reliability signal about the claim, not as evidence about the current situation: the more it has been refuted relative to its support, the more caution the decision warrants.

## Constraints
- Do NOT use any reward, progress rate, subgoal completion, difficulty, gold path, or evaluator-only metric.
- Do NOT output prose, markdown, or any text outside the JSON object.
- Do NOT include any field other than the six listed above.
```

A.14.9 EVIDENCE JUDGE

```
You are a Retrieval Second-Wave Judge.
```

```
Your mission is to decide whether one candidate abstraction, backed by resolved episode evidence, should be kept as actionable guidance for the actor.

## Guidelines
- Output format: a single JSON object with these fields:
  {
    "query_id": "<string, the query identifier>",
    "abstraction_id": "<string, the candidate abstraction ID>",
    "active_revision": <int, the revision number of the candidate>,
    "decision": "KEEP" | "DROP" --- whether to keep the abstraction as actor guidance,
    "actor_evidence": "<string, compact reworded guidance> | null --- required for KEEP, must be null for DROP,
    "grounding_refs": [{"episode_id": "<string>", "step_refs": ["<string>", ...]}, ...] --- required for KEEP, each entry must be drawn from the legal_grounding_refs set,
    "reason": "<string, brief explanation of the decision>"
  }
- The resolved episode evidence is grounding provenance (where the abstraction came from), NOT a validation test you must pass. Judge applicability primarily against the current raw_situation. Use episode evidence only to confirm the abstraction is grounded in real steps, not to fact-check its claim. Minor discrepancies between applicability_conditions and episode evidence should NOT cause DROP if the core guidance still applies.
- Prefer KEEP when the abstraction provides ANY grounded procedural or evidential guidance relevant to the query's situation, decision need, or desired effect. Only DROP when the abstraction is entirely unrelated to the query or would actively mislead the actor.
- decision KEEP: the abstraction is applicable given the resolved episode evidence. actor_evidence must be a compact reworded string preserving the original effect and semantic direction. At least one grounding_ref must point at a resolved (episode_id, step_ref) pair from the legal set.
- decision DROP: the abstraction is inapplicable, conflicting, or needs a REFINE or SPLIT. actor_evidence must be null and the reason must explain why.
- A candidate may carry a "contested" field with the number of past Episodes that supported and refuted it. Treat it as a reliability signal about the claim, not as evidence about the current situation: the more it has been refuted relative to its support, the more caution the decision warrants.
- DROP is reserved for abstractions that are clearly unrelated to the current situation, or that would actively mislead the actor. Nothing else.
- If the abstraction is relevant but thinly tested, or only partially matches the situation, choose KEEP. The system labels each kept item with its own reliability level before the actor sees it, so you do not need DROP to express doubt.

## Constraints
- Do NOT use any reward, progress rate, subgoal completion, difficulty, gold path, or evaluator-only metric.
- Do NOT output prose, markdown, or any text outside the JSON object.
- Select and compress only grounded content. Do not invent new applicability conditions, do not inflate support, and do not merge multiple abstractions into one evidence string.
- Do NOT include any field other than the seven listed above.
```

B EXPERIMENTAL DETAILS

This appendix follows the order of Section 4, from the configuration to the mechanism analysis, and closes with two experiments the main text only names: planted wrong memories and robustness.

B.1 CONFIGURATION AND CONSTANTS

Every MCMEM run in the paper uses one configuration and the constants of Table 4. No constant is tuned per benchmark, actor size or variant; a variant changes which parts of the mechanism run, never the value at which a part is set. The ablation, the planted-memory runs, the order experiment and the task-shift experiment each run this configuration beside their other variants. Each is read against its own runs, never against a cell of Table 1, except the AW full-mechanism bar of Figure 3, which is that table's AW 9B cell.

B.2 ABLATION

Each variant of Figure 3 removes one part of the configuration, and with it the parts that depend on it (Table 5). In the fifth variant, every delivered memory carries the plain prefix `Memory:` in place of its reliance label, so the actor sees no reliability wording, while the beliefs are still estimated. No variant removes procedural memory.

Table 6 shows how the actor uses memory under each variant, with every count taken per 10 steps. Deliveries count the memory lines handed to the actor. An adoption event is a delivered line whose action the actor takes within five steps, never having taken that action before the delivery. The actor would take some of these actions without any memory. Adoption is therefore counted net of chance: the same delivered lines are replayed onto a no-memory run of the same tasks, and the matches found there are subtracted. A stuck event is two consecutive steps whose actions the environment refused. More net adoption means the actor follows memory more often, and fewer stuck events are better.

Every variant hands the actor about as many lines as the full mechanism: 38.1 to 44.6 per 10 steps against 42.0 on SW, and 39.8 to 48.0 against 45.0 on AW. The success gaps of Figure 3 are therefore not a matter of handing over more or less memory. What changes is how the actor uses the lines it receives. Without the reliance labels, no delivered line carries a label set from the memory's ledgers, and the actor adopts less, 0.06 against 0.15 on SW and 0.59 against 0.83 on AW. It also gets stuck more often, 0.65 against 0.55 on SW and 0.49 against 0.34 on AW. With the plain prefix only, the same holds on SW, at 0.11 against 0.15 adoption and 0.58 against 0.55 stuck events. The label the actor sees thus changes how it uses memory, which fits the reading of Figure 3.

B.3 MECHANISM ANALYSIS

Does the reliance label tell the actor which memory is safe to follow, and where do each episode's verdicts fall? Figure 4 reads MCMEM's runs of Table 1, pooled per actor size. A delivered line counts as adopted when the actor's action matches it word for word. Failure feedback means that the environment refuses the adopted action or the one after it, so a lower share marks a safer line.

At every actor size, an adopted well-tested line meets failure feedback less often than an adopted weak-match line. The shares are 0.18 against 0.28 at 4B, 0.04 against 0.29 at 9B and 0.06 against 0.12 at 27B. Sorted by retrieval similarity instead, the same lines barely differ: the three terciles lie within 0.04 of one another at every size. A line is well-tested only when the agent's own verdicts have supported its memory in enough episodes and refuted it in none, while a weak-match line is picked by similarity alone. The label, not similarity, thus tells which memory is safe to follow.

In panel (b), support is the most common verdict at every actor size. The one-way gate that a success never confirms a claim drawn from a failure removes at most 4% of the supports. Set-aside verdicts, where the memory's own conditions never held, and not-tested verdicts never count toward a memory's standing. The standing thus rests only on episodes that took the memory's action under its conditions. We attribute part of MCMEM's lead over MemRL to this routing, since MemRL updates every retrieved memory's value from the task's reward, whether its conditions held or not.

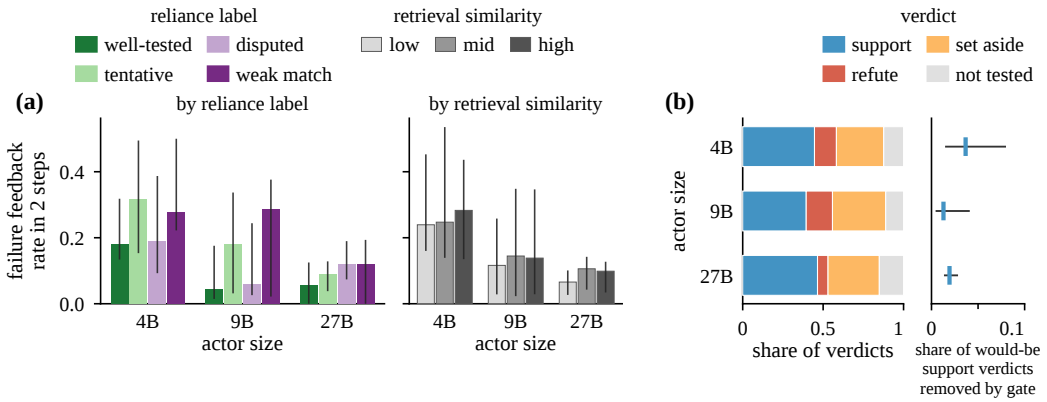

Figure 4: MCMEM's runs of Table 1 per actor size, 27B pooling BabyAI, PDDL and SW; whiskers span environments. (a) Share of adopted lines followed by failure feedback within two steps, by reliance label and by similarity tercile. (b) Share of each verdict, and of supports removed because no success confirms a claim drawn from a failure.

These results support Contribution 2: the agent's verdicts on its own experience set a label that tells the actor how far to rely on each memory.

B.4 PLANTED WRONG MEMORIES

If wrong memories get into an agent's memory, does the method hand them to the agent, and does the agent act on them? And can MCMEM tell a wrong memory from a correct one? This is the second challenge of Section 1: a memory's correctness cannot be observed directly, so a wrong memory that looks well supported is the hardest case for belief estimation. We plant wrong memories into a memory the agent has built and follow what happens to them.

All runs use SW at 9B, with the 90 tasks ordered from easy to hard. Before the 30th task, five wrong memories were added to each method's memory. In MCMEM, each planted memory was made to look well established: it is a promoted memory with six supporting verdicts, enough for the well-tested label. Its text is neutral, so nothing in it marks it as planted. The methods are MCMEM, Memora, AWM, MemRL and A-Mem, with three runs each, and each method is compared with its own runs without planted memories.

Three measures, each counted over the steps after planting, follow a planted memory from the memory to the agent's actions (Table 7); lower is better on all three. The first is the share of steps at which the agent is handed at least one planted memory. Since methods differ in how much they hand over, the second, the delivery ratio, divides the planted memories' share of what is handed over by their share of the memory. At 1 they are handed over at their share, above 1 they are favoured, and below 1 they are held back. The third counts the steps at which the agent takes a planted memory's wrong action, minus the same count in the method's runs without planting. The subtraction removes the times an agent takes such an action on its own. It covers the four memories whose wrong action shows in a single step.

MCMEM hands a planted memory to its agent at 5.0% of steps, against 17.6% for A-Mem, the next lowest, and 37.0% to 49.9% for MemRL, Memora and AWM. It is the only method whose delivery ratio lies below 1, at 0.47. MemRL hands planted memories over slightly beyond their share (1.10), Memora at twice their share (2.02), and AWM and A-Mem at about five times (4.96 and 5.01). MCMEM's agent also takes the fewest extra wrong actions, 6.7 steps, against 11.7 for A-Mem, 15.3 for AWM, 21.3 for MemRL and 51.7 for Memora.

We attribute the gap to what decides delivery. The four baselines hand over the memories most relevant to the situation, so a planted memory that matches the tasks stays relevant however often it misleads. Memora does attach a number to each memory, but it is a relevance score to the current situation, recomputed at every lookup. In MCMEM, similarity only admits a candidate, and the applicability judge delivers a memory only where it applies as written (Section 3.2.3).

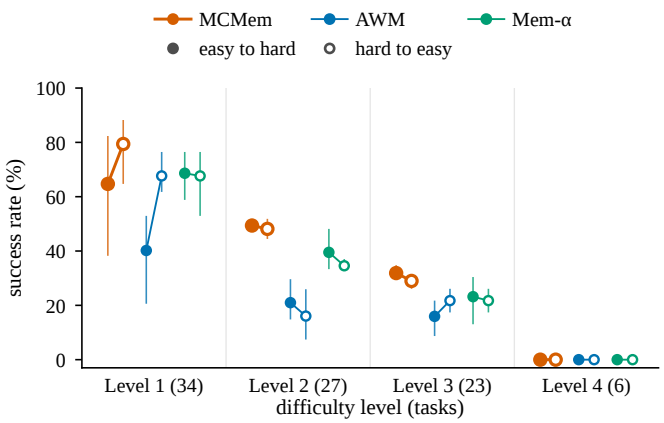

Figure 5: Success rate per difficulty level (task counts in parentheses) on SW at 9B, under the easy-to-hard (filled) and hard-to-easy (hollow) order, for MCMEM, AWM and Mem-α; points are means of three repeats, whiskers span the repeats.

Handing a memory over less often does not show that MCMEM tells a wrong memory from a correct one, so the second part asks whether its own verdicts do. Each of 30 wrong memories was also written in a correct twin with the same conditions and the same planted support. Each of three MCMEM runs planted one version of every memory, chosen at random, with memories that restate one another always in the same version. Language-model judges then checked each correct version against the claim, the recorded actions and the environment's replies, never against a run's outcome or MCMEM's verdicts, and kept 22 of the 30. A planted memory counts as decided once MCMEM's verdicts have supported or refuted it, and as judged wrong when its refutations outnumber its supports.

On the 22 kept memories, MCMEM's verdicts judge the wrong version wrong in 83% of decided cases (10 of 12), against 14% for the correct version (2 of 14) (Table 8). The wrong version is judged wrong more often in each of the three runs, and on all 30 memories the shares are 86% against 11%. We attribute this to what a verdict checks: whether the result a claim predicts followed where its action was taken, not whether the task succeeded (Section 3.2.1). A wrong claim predicts a result that the environment then does not return.

These results support Contribution 2 on the second challenge. Where every baseline hands planted wrong memories to its agent beyond their share of memory, MCMEM hands them over below it, and its agent takes the fewest extra wrong actions. Its own verdicts, which judge a memory against the steps where it applied, judge a wrong memory wrong far more often than its correct twin.

B.5 Robustness

Two experiments extend the main comparison: whether MCMEM's result depends on task order and difficulty, and how it fares when a stream changes task type.

B.5.1 Task Order and Difficulty

Does MCMEM's result depend on the easy-to-hard order, and does its lead hold on the hardest tasks, where Section 1 places the risk of trust fixed at extraction? The same 90 SW tasks were run in both orders at 9B for MCMEM, AWM and Mem-α, three repeats per method and order. AWM is the contrast because its workflows are kept and delivered as induced and never re-rated. Success is read per difficulty level of the benchmark's task-to-level mapping, which holds the same tasks in both orders, so levels are paired per task (Figure 5, Table 9).

MCMEM solves 47.4% of the tasks in the easy-to-hard order and 51.9% in the reverse, ahead of AWM by 21.8 and 16.0 points and of Mem-α by 3.7 and 10.4 points. Level 4 holds six tasks that no method solves under either order, so level 3 is the hardest any method solves. At levels 2 and 3,

MCMEM is ahead of both baselines under both orders. At level 3 it solves 31.9% and 29.0%, against 15.9% and 21.7% for AWM and 23.2% and 21.7% for Mem-α. We attribute this lead to the trust MCMEM re-estimates from later tasks, where AWM keeps each workflow as induced. These results support Contribution 3 under both orders: MCMEM's success does not depend on the easy-to-hard order, and its lead over AWM and Mem-α holds at the hardest level any method solves.

B.5.2 TWO-OBJECT TASKS AFTER SINGLE-OBJECT TASKS

Reordering one benchmark's tasks by difficulty keeps the task types mixed, so it cannot show what memory built on one task type does once the stream moves to another. At 9B, a stream of the 24 single-object AW tasks that open the easy-to-hard order was followed by all 17 two-object tasks in task order. A second stream presented the 17 two-object tasks alone, from an empty memory. MCMEM and five memory baselines, A-Mem, AWM, Dynamic Cheatsheet, Memora and MemRL, ran both streams and the no-memory agent the first, three repeats each with the settings of Table 1. Success is read on the 17 two-object tasks.

On the first stream, MCMEM is compared with each baseline by net flips: tasks where its success rate over three repeats is above the baseline's, minus tasks where it is below. The floor is the larger of the two methods' bands, a band being the largest net flips between two repeats of one method. MCMEM counts as above a baseline when its net flips exceed the floor and two further conditions hold. Each of the six leave-one-out nets, dropping one repeat of either method, exceeds the floor with the same sign. MCMEM's lowest repeat is also above the baseline's highest.

On the first stream MCMEM solves 12.0 of the 17 two-object tasks on average, against 2.0 to 8.0 for the five memory baselines (Table 10). All three conditions hold against every baseline, each against a floor of 3. The net flips are 10 against A-Mem, 16 against AWM, 7 against Dynamic Cheatsheet, 13 against Memora and 16 against MemRL, with leave-one-out nets of 8–10, 14–16, 4–7, 12 and 14–16. From an empty memory the methods solve 5.33 to 9.33 on average, and the no-memory agent solves 7.0 on the first stream. After the single-object tasks, MCMEM thus solves more two-object tasks than each of the five memory baselines, supporting Contribution 3 on a subset of Table 1's AW tasks at 9B.

Table 4: Every constant that the method (Section 3, Appendix A) names without a value, with what it fixes and its value, grouped by the part of the mechanism that uses it.

Symbol	Meaning	Value
Episode end: promotion, revision, withdrawal		
k_h	Hypotheses abstracted from one finished episode	4
k_{ep}	Most semantically similar past episodes drawn per facet when the episode end forms pairs to judge	3
k_{abs}	Most semantically similar promoted memories drawn per facet in the same draw, withdrawn ones ranked separately	6
k_{hist}	Drawn episodes one maintenance context keeps as its historical episodes	3
k_{focus}	Focus hypotheses one maintenance context keeps	6
κ	Share of a grounded procedure's content words that must be lifted from its own episode	0.50
ρ	Refuting share of a hypothesis's ledger above which neither promotion route may fire	0.34
T	Episodes between periodic reviews	5
k_{new}	New ledger entries that queue a promoted memory for the periodic review	3
k_{sw}	Promoted memories reviewed in one periodic review	10
n_s	Supporting episodes judged again against a proposed wording	3
k_{merge}	Promoted memories offered to the merge judge beside a newly promoted one	4
n_{wd}	Supporting or refuting verdicts a promoted memory must hold before it may be withdrawn	3
ρ_{wd}	Refuting share of the ledger at which a promoted memory is withdrawn	0.50
Beliefs		
z	Level of the lower confidence bound that gives the correctness belief	1.96
n_{price}	Decided usage records under which the adoption belief stays full	8
n_{use}	Verified usage records a promoted memory must hold before its demotion flag may be raised	5
n_{gap}	Adopted deliveries whose action was accepted or refused before the execution belief may change the delivery form	5
ρ_{gap}	Refused share of those deliveries at which a grounded procedure is delivered alone	0.50
ℓ_{use}	Line the lower confidence bound on the share of usage records whose conditions held must fall below	0.50
Read operator: recall, grade, slots		
w	Steps of the trajectory the read is given	8
η	Semantic similarity a candidate must reach to enter the pool	0.20
k_{norm}	Candidates entering the pool per search key	3
β	Weight of the condition similarity in the order of the pool	0.50
k_{proc}	Promoted memories with a grounded procedure recalled on the goal, once per episode	2
k_{cold}	Episodes whose narratives are delivered when the pool is empty from the start	4
ℓ_{min}	Correctness-belief line a promoted memory must clear for the well-tested reliance label	0.50
n_{min}	Supporting or refuting verdicts a promoted memory must hold for the well-tested reliance label	3
η_{dup}	Semantic similarity at which two memories count as near-duplicates and share one slot	0.90
k_{con}	Consecutive steps of delivery after which a memory yields its slot	3
n_{tok}	Length of one step's delivered block, in tokens of the actor's tokenizer	2048
Procedural memory		
κ_R	Share of a workflow's content words that must be lifted from its own episode	0.50
n_R	Workflows drawn from one episode	3
k_{rec}	Workflows delivered per step	2
η_{div}	Semantic similarity a workflow must reach to take the verb-diversity slot	0.45
Δ_{div}	Gap behind the most similar workflow that the same slot still allows	0.20
k_{idx}	Index leads delivered per goal word	3
n_{idx}	Episodes a lead needs behind it before it is delivered	2
ρ_{goal}	Share of goals a word may appear in before it is dropped as a key	0.60
n_{goal}	Goals seen before that frequency filter starts to apply	5

Table 5: What each variant of Figure 3 removes, and what that removal takes with it. In the first two variants procedural memory, the workflows and the observation index, is still delivered at the full mechanism's rate.

Variant	Removed	Removed with it
Full mechanism	–	–
Without the correctness belief	the correctness belief	withdrawal; the reliance labels but weak match; $\mathcal{R}_i^{\psi}(g)$, the memories whose grounded procedure is most similar to the goal; the refutation cap ρ
Without the reliance labels	every reliance label but weak match, which no ledger decides	$\mathcal{R}_i^{\psi}(g)$
Without the usage ledger	the usage ledger	the adoption belief's place in the block's order; the demotion flag b_r
Without one gate and two workflow rules	the one-way gate that no success confirms a claim drawn from a failure; the rules that stop delivering a refuted workflow or one whose header names an action none of its steps takes	–
Plain prefix only	every reliance label and fallback prefix, replaced by the plain prefix `Memory:`	the reliance label's place in the block's order

Table 6: How the actor uses memory under the variants of Figure 3 at 9B: all five variants on SW, the first three on AW. Means over three repeats, per 10 steps. *Adoption*: adoption events net of chance. *Stuck*: two consecutive refused steps.

	Variant	Deliveries	Adoption	Stuck
SW	Full mechanism	42.0	0.15	0.55
	Without the correctness belief	40.2	0.14	0.57
	Without the reliance labels	38.1	0.06	0.65
	Without the usage ledger	44.6	0.12	0.60
	Without one gate and two workflow rules	41.8	0.17	0.52
	Plain prefix only	43.0	0.11	0.58
AW	Full mechanism	45.0	0.83	0.34
	Without the correctness belief	41.8	1.03	0.38
	Without the reliance labels	39.8	0.59	0.49
	Without the usage ledger	48.0	0.98	0.29

Table 7: Five wrong memories planted before the 30th task on SW at 9B; means over three runs per method. *Steps*: share of steps after planting at which the agent is handed a planted memory. *Delivery ratio*: the planted memories' share of what is handed to the agent over their share of memory. *Extra wrong actions*: steps taking a planted wrong action, minus the method's runs without planting.

Method	Steps (%)	Delivery ratio	Extra wrong actions
MCMEM	5.0	0.47	6.7
Memora	44.8	2.02	51.7
AWM	49.9	4.96	15.3
MemRL	37.0	1.10	21.3
A-Mem	17.6	5.01	11.7

Table 8: Thirty memories planted into MCMEM on SW at 9B, each in a wrong or a correct version; three runs. *Planted*: planted memories summed over the runs. *Decided*: supported or refuted by MCMEM's verdicts by the 80th task. *Judged wrong*: share of decided memories whose refutations outnumber their supports.

Version	Planted	Decided	Judged wrong (%)
The 22 memories kept by the independent check			
Wrong	32	12	83
Correct	34	14	14
All 30 memories			
Wrong	44	14	86
Correct	46	19	11

Table 9: The runs of Figure 5, three repeats per method and order: success rate (%) per difficulty level (task counts in the heading) and overall, means over the repeats, under the easy-to-hard (E→H) and hard-to-easy (H→E) orders.

		Success rate per level (%)				
Method	Order	Level 1 (34)	Level 2 (27)	Level 3 (23)	Level 4 (6)	All (%)
AWM	E→H	40.2	21.0	15.9	0.0	25.6
	H→E	67.6	16.0	21.7	0.0	35.9
Mem-α	E→H	68.6	39.5	23.2	0.0	43.7
	H→E	67.6	34.6	21.7	0.0	41.5
MCMEM	E→H	64.7	49.4	31.9	0.0	47.4
	H→E	79.4	48.1	29.0	0.0	51.9

Table 10: Two-object AW tasks solved, of 17, at 9B, mean over three repeats, for the stream that presents the 17 tasks after 24 single-object tasks and for the stream that presents them alone from an empty memory. –: not run.

Method	After 24 single-object tasks	From empty memory
No memory	7.0	–
A-Mem	6.67	6.0
AWM	3.33	7.0
Dynamic Cheatsheet	8.0	5.33
Memora	5.0	5.33
MemRL	2.0	6.0
MCMEM	12.0	9.33