CURATORMEM: METADATA-GROUNDED CURATION OF ACCUMULATING AGENT MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents act in environments that keep changing, where tools, code and user goals move on between encounters, so their memory fills with successive attempts at related problems that overlap. To deal with this situation, patch-based memory is introduced to manage these attempts: each attempt is recorded as a patch, and the state it supersedes is kept as history, enabling cross-patch retrieval. However, a store built to persist is written once and never revised, so it decays as it grows: redundant entries accumulate and split the retriever's focus. In a patch-based system, superseded attempts and the ones that replaced them sit side by side, and there is no record to evaluate their worth. For this reason, we propose CuratorMem, a memory layer over an append-only, patch-based store: a manifest carries version lineage and per-entry statistics, an additive curation operator refines and enriches what is stored based on these heuristics, and retrieval is scoped to a task chain and ordered by that lineage. The key idea is to decouple evidence from opinion: curated content is written by a model, but an entry may be endorsed only on a recorded outcome, grounded outside the model that wrote it. Across GAIA, GAIA2, LoCoMo and τ^2-bench with multiple backbones, CuratorMem improves on an uncurated patch store and two external memory baselines (A-Mem, Mem0). Every experiment in this paper is backed by full trajectories, anonymously provided in Appendix A, and the system will be released as a memory-layer SDK after publication.

1 INTRODUCTION

Conditions of building Large language model (LLM) agents are changing: new tools are created, codebases are updated, and user questions evolve. Most of the scaffolding developed for earlier agent frameworks has since been discarded because it compensated for limitations that newer models no longer have. Memory management, however, remains necessary (Hatalis et al., 2023). Models may be able to read instructions and code, but they cannot remember which call a specific deployment accepted last week unless that information is stored. Greater model capability cannot recover missing records. For this reason, agents are given memory that persists beyond a discrete episode (Wang et al., 2023; Zhang et al., 2026a). However, experience-based memory accumulates successive and often conflicting attempts at overlapping problems, so the store the agent reads from also changes over time. Such a store must retain information that may become relevant again while revising outdated content, which forms the stability–plasticity dilemma of agent memory. Retaining everything does not resolve this dilemma, since relevant information must still be found among all retained entries (Liu et al., 2024), yet each deletion risks discarding information needed for a future task.

Existing frameworks typically prioritize one side of this trade-off. Accumulative memories append every entry to the store. As a result, superseded records accumulate, making relevant memories harder to find among them (Xu et al., 2025b; Chhikara et al., 2025). Compacting memories actively prune entries to maintain a smaller store. In a non-stationary environment, entries that a later rollback or release makes relevant again may already have been pruned and discarded (Wei et al., 2026a; Zhao et al., 2026). Patch-based memory (Wang, 2025; Zhou et al., 2026b) retains each superseded entry as history instead of overwriting it, preserving entries that may prove useful later. Once a patch is committed, however, it is not revised, so new entries accumulate without further review. A-Mem responds to this problem by updating stored notes as related ones are added, and memory-OS designs move content between stores with different levels of durability (Xu et al., 2025b; Kang et al., 2025).

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.

The chain, as the store held it

`v1`	`create_table(t2, schema)`	ok
	catalog 2.0: `partition_spec` becomes mandatory	
`v2`	`create_table(t2, schema)`	**fail**
`v3`	`create_table(..., partition_spec)`	ok

What the next task on this chain is served

Accumulative	both signatures, but could not adequately distinguish which the request needs
Compacting	drops or merges the old form, so the served call fits neither catalog
Patch-based memory	serves the later signature, so a request that still needs `v1` receives `v3`
CuratorMem	the lineage and the reason; the other version stays on the chain

Figure 1: **Why a store that only keeps history is not enough.** A production catalog change (Appendix G). `v1` is valid before the change; `v2` is that call after it; `v3` is the new signature. Bottom: what each design serves. Only the last row keeps both and records why each was written.

Both approaches revise stored content through destructive updates. The previous state is removed, and a model rereading an entry it has just edited can compound those errors.

A catalog upgrade in a production deployment illustrates why retaining history alone is insufficient (Figure 1; Appendix G provides the full example). The upgrade changed the signature of a stored API call. The new catalog rejects the old signature, while the old catalog rejects the new one. Both signatures therefore remain in use, and a later request may require either. An accumulative store returns both but cannot reliably determine which signature the request requires. A compacting store discards the old signature or merges it with the new one, producing a call that neither catalog accepts. Patch-based memory retains both versions but serves the newer signature, even when the request requires the older one. Selecting the appropriate version requires the reason for each patch to be available during retrieval; CuratorMem records that reason on the lineage.

To handle situations where additional information is required to distinguish among patches, and where iterative environmental feedback is given to refine them, we propose CuratorMem, a memory layer that maintains version lineage and statistical metadata for every entry separately from the model's assessment of it, and serves that record with the retrieved lineage. Our contributions are as follows:

- We propose CuratorMem, a memory layer over a single append-only store organized by a metadata manifest, similar to that which database systems use to query immutable history. Curation is additive: refine, critique, and enrich operators improve stored experience without overwriting earlier entries. Retrieval is chain-partitioned and lineage-ordered: read cost is a near-linear function of the queried chain (Tables 5 and 6).
- We propose a metadata-based certification system that separates evidence from opinion. A model writes the curated text, but an entry may be presented as a successful attempt only when its outcome was recorded by the benchmark's scorer or the environment, rather than by the model being evaluated. We identify the factors that determine whether a curated entry is worth serving and use them to guide both storage and retrieval.
- We conduct comprehensive evaluations of our system across three backbones on four benchmarks, and report drift-corrected paired effects with bootstrap intervals and detailed statistical analysis.

2 Related Work

Under the operational tests of Table 15 in Appendix E.3, CuratorMem is the only system fully marked on curated entries, non-destructive storage, and outcome-ranked retrieval together.

2.1 Memory for LLM Agents

Agent memory extends the information available to an agent beyond its current episode (Yao et al., 2023; Shinn et al., 2023). Persistent memory takes many forms: distilled insights (Zhao et al., 2024), paged context (Packer et al., 2023; Kang et al., 2025), graphs indexed for associative or temporal

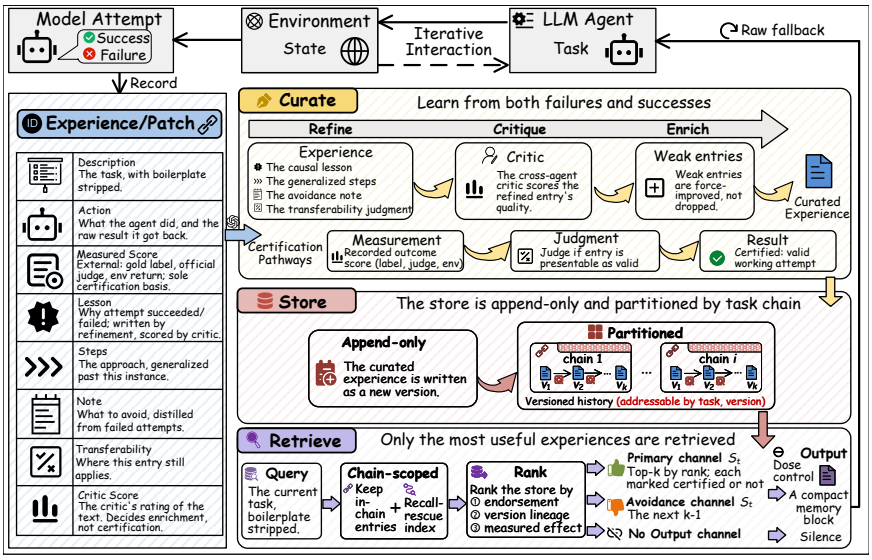

Figure 2: The workflow of CuratorMem.

recall (Zhong et al., 2023), procedures abstracted from past trajectories (Fang et al., 2025; Wang et al., 2023), or parameters updated at test time (Behrouz et al., 2025). Beyond retaining information, a memory system must distinguish which of its entries a later request should see, and refine those entries when later feedback arrives. Both require evidence that is not in the stored text alone.

Our comparison focuses on how a growing store updates its contents and whether those updates preserve earlier versions. One line of work learns to improve the memory itself, rewriting notes as related ones are added (Xu et al., 2025b; Zhou et al., 2026b; Ouyang et al., 2026; Zhang et al., 2026a; Yu et al., 2026). A second line manages memory under a storage budget, pruning or compacting stale entries (Chhikara et al., 2025; Wei et al., 2026a; Feng et al., 2026). Other approaches control how memory is revised or accessed: they rewrite entries in place (Wu et al., 2026), organize memory into tiers to bound read latency (Zhang et al., 2026b; Zhu et al., 2026), or type dialogue into profile, event and source-grounded records (Liao et al., 2026a).

Updating memory raises two distinct problems: the preservation of earlier states and reliability assessments of what is retained. In-place edits remove the previous state, while retaining entries without reviewing them allows stale or potentially ineffective contents to accumulate. Xiong et al. (2026) report that unvetted records can propagate errors to agents that retrieve them, and that downstream outcomes provide labels for stored entries at no additional cost. Addressing both problems requires a store that revises without discarding the predecessor and that keeps per-version evidence.

2.2 Memory Provenance and Revision

Recent work addresses memory provenance and history through entry verification, version control, and policies governing memory use. These approaches verify entries before admission (Xu et al., 2025a; Yang et al., 2026), incorporate semantic version control into the memory representation (Park, 2026; Su et al., 2026), or define policies that determine when a memory may be used (Zhang et al., 2026c; Hu et al., 2025). Stores designed specifically for agents also make history part of the memory representation (Zhou et al., 2026c). Our focus is on versioning individual entries and certifying each from its own recorded outcome, rather than versioning or consolidating the store as a whole. Closest to our design, patch-based memory (Wang, 2025; Zhou et al., 2026b) records each update as a patch and retains every superseded state. A later request still has to choose among those patches from the stored text alone. We keep that append-only write: the original entry stays as written, and what we add is manifest heuristics that distinguish among patches together with a non-destructive curation pass that writes a successor beside them. ESAA (Brito dos Santos Filho, 2026) similarly separates mechanical recording from judgment-based curation, without certifying on recorded outcomes.

Fan et al. (2026) note that recall-based benchmarks are generally concerned with the traceability of stored results. Current validation mechanisms share that concern. Neither says whether a stored outcome still holds: a result that can be traced may already have been superseded.

3 METHOD

3.1 PRELIMINARIES

Our method builds on the Stateful Reflective Decision Process of Wang (2025), which divides each task attempt into two operations, interpreted together as one step of policy iteration: WRITE records the attempt and its outcome, and READ uses that record to inform the next attempt. Our memory layer implements these two operations. Let t index attempts and M_{t-1} the store just before attempt t. A completed episode is $e_t = (x_t, \zeta_t, o_t)$, where x_t is the task, ζ_t the trajectory of actions and observations, and o_t the recorded outcome. $\textsc{Write}(M_{t-1}, e_t)$ stores that episode. $\textsc{Read}(M_{t-1}, x_t)$ returns a block of text prepended to the prompt for the current task x_t. This block is the agent's only input from the memory layer (Figure 2). The underlying agent policy and tools remain unchanged. Across a chain of T tasks $x_1, \ldots, x_T$, the experimental arms therefore use the same agent configuration; the memory layer determines the additional context provided to the agent.

The layer stores experience as patches (Wang, 2025; Zhou et al., 2026b): an updater U sets $M_t = U(M_{t-1}, x_t)$ and records every non-additive change as

$$p_t = \big(\tau_t,\ C_t^-,\ C_t^+,\ r_t,\ z_t,\ e_t\big), \tag{1}$$

where τ_t is the time of the change, C_t^- and C_t^+ the content before and after it, r_t the rationale, z_t a summary, and e_t the supporting episode. Our store preserves this tuple and adds metadata fields.

SRDP also provides a framework for identifying memory failures through its bound on the value gap between a memory-conditioned policy and the optimal policy:

$$\sup_s \big|V^{\pi^*}(s) - V^{\pi_M}(s)\big| \ \le\ \frac{2R_{\max}}{(1-\gamma)^2}\big(\varepsilon_{\mathrm{LLM}}(r_M) + \delta_M\big), \tag{2}$$

where s is a state, $V^\pi(s)$ its value under policy π, π^* the optimum, π_M the store-reading policy, $R_{\max}$ the maximum reward and γ the discount factor. The bound distinguishes three sources of error: a relevant entry may never be stored, which the coverage radius r_M measures; it may be stored but not returned, which the retrieval error δ_M measures; or, once returned, it may be used incorrectly, which $\varepsilon_{\mathrm{LLM}}(r_M)$ measures.

Those terms, however, assume a discounted MDP with a fixed retrieval map and a stationary horizon. Since a language agent on a changing benchmark does not satisfy these assumptions, we take the three failure modes as design targets rather than as a theoretical guarantee (Appendix I). Accordingly, CuratorMem addresses them in the same order: first, the store is append-only, so that no curation step removes an entry a later task may need (Section 3.2); second, retrieval is scoped to a task chain and ordered by version lineage, so that a superseded entry ranks behind its successor rather than merging into it (Section 3.3); and third, curation improves the presentation of stored experience without changing the agent's instructions (Section 3.4).

3.2 STORE: NEVER DELETE

We use gradient-based training as an analogy to the store–retrieve–curate methodology. Storing is similar to the initialization of a weight for every patch. WRITE is the step that performs that initialization, through a maintenance map $\mathcal{M}$, and prior systems implement this map destructively by editing fields in place or merging entries, making an entry's earlier state unrecoverable after revision (Xu et al., 2025b; Chhikara et al., 2025; Zhou et al., 2026b). In comparison, our system adds a new entry, so initializing c_t does not replace the weight already assigned to its predecessor:

$$\mathcal{M}(M_{t-1}, c_t) \ =\ M_{t-1} \cup \{c_t\}, \tag{3}$$

where c_t is the new entry: it has its own identifier $\mathrm{id}(c_t)$, a version $\mathrm{ver}(c_t) = \mathrm{ver}(e) + 1$ relative to the task's latest entry e, and a pointer to $\mathrm{id}(e)$. This pointer records that c_t supersedes e without

altering it. The store preserves both content and provenance. Content is retained without loss: every stored entry remains retrievable unchanged under its own identifier,

$$\forall\, e \in M_t \;\; \exists\, \mathrm{id}(e) :\; A[\mathrm{id}(e)] = e \quad \text{for all } t' > t, \tag{4}$$

where A is the store's address map from identifiers to entry contents and t' is any later attempt. Consequently, curation adds a successor and $|M_t| \geq |M_{t-1}|$. Provenance, in turn, is recorded for every field and thereby determines how that field may be used. In particular, the harness records most ranking metadata directly from execution, without consulting a model or a reference answer (Table 1), just as gradient-based training reads the loss off the step that was just run. Accordingly, the recorded outcome v_c is that loss, π_c names its source (ENV, GOLD, or SELF), and H_c records the versions c supersedes.

Table 1: **Stored metadata, its provenance, and its permitted uses.**

Field	Written by	Available	Gold-derived	Affects
γ_c chain key	harness	at write	no	eligibility (Eq. (6))
s_c cleaned situation	harness	at write	no	eligibility (overlap floor)
$\mathrm{ver}(c)$, H_c lineage	store	at write	no	ranking, first key
v_c recorded outcome	scorer	after the attempt is scored	**yes**	ranking, endorsement partition
π_c provenance of v_c	harness	with v_c	no	whether v_c may be read at all
$\bar{\Delta}_c$ reuse co-occurrence	harness	after each later attempt	inherits π_c	ranking, second key
q critic quality	critic model	at write	no	rendering only (enrichment)

Endorsement, by contrast, determines whether an entry may be presented as a successful attempt, just as gradient-based training counts an example as a success only when the observed loss is trusted. We call this endorsement a certificate and display it as a checkmark in the rendered context block:

$$\textsc{Endorse}(c) \iff \pi_c \neq \text{self} \;\wedge\; \big(v_c \geq \tau_s \;\vee\; v_c > v^0(x_c)\big) \tag{5}$$

The first condition therefore restricts the source of the outcome to environmental feedback (ENV) or an externally recorded score (GOLD), forbidding the acting model's self-assessment (SELF), which can reproduce the errors in its own attempt. The second condition then sets the required performance: v_c must meet the benchmark's success threshold τ_s or exceed $v^0(x_c)$, which is the first recorded score of task x_c before a later entry on that task is served. Because the evaluation protocol records v^0 at iteration 0 with an empty store, a first visit displays no certificate.

Separating authorship from certification therefore allows a model to write an entry without certifying its own work, just as a model may produce the forward pass while the loss that updates it is not its own grade. When that separation is dropped, the store underperforms both the memory-free baseline and the external memory layers (Table 7; Appendix D.7).

3.3 Retrieval: One Chain at a Time, Newest Version First

Retrieval is similar to the forward pass in gradient-based training, where the current weights decide which stored examples are read out for this input. CuratorMem restricts retrieval to the current task chain. Equation (6) is the base heuristic, using the chain key to select a partition of the store:

$$\mathcal{C}_t = \{\, c \;\mid\; \gamma_c = \gamma_t,\;\; \mathrm{sim}(s_t, s_c) \geq \theta \,\}, \tag{6}$$

where c is a stored entry. Its chain key γ_c must match the current task's key γ_t (a partition identifier, not the discount in Eq. (2)). The cleaned situations $s_t = \textsc{Core}(x_t)$ and $s_c = \textsc{Core}(x_c)$ are the two task texts with run-specific wording stripped, and sim is their stopword-filtered Jaccard overlap (Eq. (16)). Algorithm 1 then falls back: exact-task entries are taken alone, like a lookup of the same training example, and θ, similar to a kernel bandwidth, is waived if it would empty a chain with history. Candidates are ranked using recorded metadata: endorsement takes priority, followed by version lineage and reuse statistics:

$$c \succ c' \iff \big(\textsc{Endorse}(c),\; \mathrm{ver}(c),\; \bar{\Delta}_c\big) \;>_{\mathrm{lex}}\; \big(\textsc{Endorse}(c'),\; \mathrm{ver}(c'),\; \bar{\Delta}_{c'}\big), \tag{7}$$

under lexicographic order $>_{\mathrm{lex}}$. An endorsed entry always ranks above an unendorsed entry. Among entries with the same endorsement status, the later version ranks first; the reuse statistic breaks any remaining ties, for reasons similar to those that order a weighted readout by trusted loss, then by the latest update, then by running weight. This statistic summarizes outcomes from later reads R_c that included c in the rendered block and received an externally recorded score. For each such read r,

v_r is the scored outcome of task x_r and π_r is that score's provenance; $\delta_r = v_r - v^0(x_r)$ measures improvement over the same task's memory-free score:

$$\bar{\Delta}_c = \begin{cases} 0 & \text{if } R_c = \emptyset, \\ \dfrac{1}{|R_c|} \displaystyle\sum_{r \in R_c} \delta_r & \text{otherwise,} \end{cases} \qquad R_c = \{r : \ c \in \text{served}(r), \ \pi_r \in \{\text{ENV}, \text{GOLD}\}\}. \tag{8}$$

Here served(r) is the set of entries placed in the rendered block for read r, including any that ranking selected but the budget L then truncated. Self-assessed outcomes therefore never count as evidence that an entry helped. The statistic measures co-occurrence rather than causal contribution, so it is used only as the final ranking key (Appendix I).

READ renders the highest-ranked candidates as a primary group of k entries and a secondary group of up to $k-1$ entries:

$$\text{READ}(M_{t-1}, x_t) = \text{RENDER}\big(S_t, \overline{S}_t\big), \qquad S_t = \mathcal{C}_t^k, \quad \overline{S}_t = \mathcal{C}_t^{2k-1} \setminus S_t, \tag{9}$$

where k is the number of primary entries a read may serve, $\mathcal{C}_t^k$ those first k candidates under $\succ$, RENDER writes both groups as text, S_t is the primary group and $\overline{S}_t$ the avoidance group. The rendered block is limited to L characters because it shares the prompt with the task. Certified entries in S_t are marked according to Eq. (5); the avoidance group holds the next-ranked attempts, typically failures. A chain returns no context if it contains neither a certified entry nor an attempt with a score above zero. Operating values of k, L and θ, and the remaining eligibility checks, are in Appendix I.

3.4 CURATION: REFINE, CRITIQUE, ENRICH

Curation improves stored experience through three additive stages on the write path, just as a parameter update in gradient-based training applies an increment computed from the observed loss to the current weights. Refinement extracts reusable guidance from a trace and stores it alongside the original execution record. An independent critic therefore scores the result, since generated guidance can contain errors. If the score is low, enrichment proceeds and the original entry is preserved (Eq. (4)).

A completed attempt is converted into an experience $e = \text{ANALYZE}(x, a, h, o)$, where x and o are the task and outcome of the episode and (a, h) unpack the trajectory ζ: a the actions and h the interaction trace. Refinement uses the version history $H(e) = H_c$ of that task — the predecessor entries in the same chain — to produce reusable guidance while preserving the original record:

$$\tilde{e} = \text{REFINE}\big(e, H(e)\big) \tag{10}$$

where $\tilde{e}$ is the refined experience. The external critic scores the result out of ten, and a low score routes the entry to enrichment rather than deletion:

$$e^\star = \rho(e) = \begin{cases} \text{ENRICH}(\tilde{e}) & q < \tau_c, \\ \tilde{e} & \text{otherwise} \end{cases} \tag{11}$$

where ρ is the curation map, $e^\star$ the curated experience, q the critic's score and τ_c the threshold for enrichment. Both branches retain all existing entries. The critic's score determines how the guidance is written, but does not determine whether the entry may be presented as a successful attempt.

4 EXPERIMENTS

Under repeated-task reuse, the curated store improves on an uncurated patch store and on the two external memory layers; a more detailed protocol and analysis are in Appendices D.1 and D.3.[1]

4.1 EXPERIMENTAL SETUP

Datasets. We evaluate on four benchmarks covering question answering and interactive tool use. GAIA (Mialon et al., 2024) requires multi-step reasoning to produce a short answer. We report exact match both alone and with an LLM judge consulted when exact match fails. GAIA2 (Froger

[1]Three attempts per item, two on τ^2-bench; the gate reads an externally recorded outcome (Table 11).

et al., 2026) evaluates sequences of tool calls that interact with live application state, using the benchmark's own judge. LoCoMo (Maharana et al., 2024) evaluates memory over long, multi-session conversations using token-level F1. τ^2-bench (Barres et al., 2025) evaluates actions in a simulated domain, with rewards based on checks of the resulting database state. This is environmental feedback of the kind available in deployment (Table 11). GAIA and LoCoMo evaluate question answering; GAIA2 and τ^2-bench evaluate interactive execution.

Baselines. We compare CuratorMem with four baselines to distinguish the benefits of persistent memory, curation, and the choice of memory layer. Raw uses no persistent memory. Patched stores and retrieves uncurated patches using the same underlying store, retrieval framework, and context budget as CuratorMem, but without curation or endorsement; the two arms also use different overlap thresholds, as specified below. A-Mem (Xu et al., 2025b) maintains linked notes and updates them as related notes are added, while Mem0 (Chhikara et al., 2025) extracts and consolidates facts into a vector store. Both retain their native retrieval, storage formats, and write-time prompts, so these comparisons evaluate complete memory layers. Within each benchmark–backbone condition, all arms use the same task agent and backbone. The memory layers share a `record`/`inject` interface and the same external model for their internal LLM calls. Appendix E.2 provides further details.

Implementation details. We evaluate repeated-task reuse with three attempts per task, except for τ^2-bench, which uses two. Memory updates occur after an attempt is scored and before the next attempt; Raw performs no updates. On GAIA, GAIA2, and τ^2-bench, a chain contains one task's attempt sequence. On LoCoMo, a chain contains one conversation, allowing later questions to retrieve previously stored experience from that conversation. Between iterations, an LLM rewords each task while preserving its facts and expected answer; all arms and backbones use the same cached variants. We use HY3 with high reasoning effort, DeepSeek-v4 (`deepseek-v4-pro`), and GPT-5.5 with low reasoning effort, all at temperature 0. Table 2 reports the twelve benchmark–backbone combinations using each benchmark's native metric. Table 19 specifies sample sizes and per-benchmark protocols, including the separate native-protocol LoCoMo evaluation. For the two patch-based arms, retrieval uses k=3 and a 900-character context budget, with overlap thresholds of 0.05 for Patched and 0.08 for CuratorMem. The critic threshold is τ_c=5 out of 10, and the endorsement threshold is τ_s=0.5. The endorsement gate uses the recorded outcome source specified for each benchmark in Table 11; the critic's quality score does not certify success. Appendix I.3 details truncation, the lineage footer, and fallbacks; Appendix J the rest of the protocol.

Table 2: **Main results.** The curated store has the highest final-round point estimate in every benchmark–backbone cell, with one tie against Patched.[2]

Benchmark	Model	Scoring	Raw	Patched	A-Mem	Mem0	CuratorMem
GAIA	HY3 (high)	exact / judge	19.79 / 25.83	19.19 / 24.24	24.49 / 27.45	20.00 / 22.00	**26.80 / 30.93**
	DeepSeek-v4	exact / judge	53.00 / 56.00	51.00 / 56.00	53.00 / 55.80	61.00 / 64.80	**63.00 / 67.00**
	GPT-5.5 (low)	exact / judge	25.51 / 32.65	23.00 / 30.80	23.00 / 28.99	27.00 / 34.00	**32.32 / 39.39**
GAIA2	HY3 (high)	official judge	33.48	31.35	29.01	33.54	**42.00**
	DeepSeek-v4	official judge	29.91	36.72	33.65	37.09	**53.42**
	GPT-5.5 (low)	official judge	31.57	41.01	19.96	37.18	**41.99**
LoCoMo	HY3 (high)	token-F1	9.46	14.50	13.11	25.80	**28.04**
	DeepSeek-v4	token-F1	35.55	32.21	35.35	35.85	**36.92**
	GPT-5.5 (low)	token-F1	16.95	41.21	50.58	51.22	**53.45**
τ^2-bench	HY3 (high)	reward	67.50	72.50	71.25	68.75	**77.50**
	DeepSeek-v4	reward	76.25	77.50	73.75	82.50	**86.25**
	GPT-5.5 (low)	reward	68.75	**86.25**	79.75	76.25	**86.25**

4.2 Empirical Results

Performance Analysis. The curated store has the highest final score in every benchmark–backbone combination, including one tie with Patched (Table 2). After pooling across backbones and correcting for empty-store offsets, the effect is positive in all sixteen comparisons (Table 10).

[2]Reworded reuse of the same item. Printed scores are unadjusted; empty-store offsets are removed in Table 10. Run-to-run spread: Table 20. Protocol: Appendix D.

The ablations in §4.3 attribute more of the gain to endorsement on external evidence than to retrieval.

Table 3: **LoCoMo native protocol**: distinct questions on three conversations, each asked once, not a rewording of the same item that preserves the earlier expected answer. LLM-judge correctness J (%), n=497, with the best score for each row and answering model in **bold**. The extraction model and judge are fixed across all conditions.

		Answerer: HY3 (high)					Answerer: GPT-5.5 (low)				
Question type	n	Raw	Patched	A-Mem	Mem0	CuratorMem	Raw	Patched	A-Mem	Mem0	CuratorMem
Multi-hop	74	4.1	28.4	14.9	39.2	**41.9**	8.1	32.4	25.7	44.6	**47.3**
Temporal	90	3.3	3.3	42.2	70.0	**78.9**	1.1	7.8	73.3	63.3	**82.2**
Open-domain	21	38.1	47.6	47.6	52.4	**61.9**	57.1	**71.4**	66.7	66.7	66.7
Single-hop	200	13.0	58.5	36.5	**65.5**	65.0	17.5	67.0	61.0	**76.0**	73.5
Incidental detail	112	8.9	12.5	**17.0**	12.5	14.3	21.4	44.6	**56.2**	42.9	42.9
All	497	10.1	33.2	30.4	49.9	**52.5**	15.7	46.3	57.1	61.2	**64.0**
token-F1	497	9.5	14.5	13.1	25.8	**28.0**	17.0	41.2	50.6	51.2	**53.5**

Native-Protocol Analysis. We also evaluate LoCoMo using the protocol for which A-Mem and Mem0 were designed: ingest each session, retrieve context for each question, and answer using only that context (Table 3). A later question is a new item on the same conversation, not a rewording that keeps the earlier expected answer. The curated store outperforms both baselines overall with both answering models, with the largest gain on temporal questions a version chain can record.

Cost Analysis. Retrieved context can reduce the number of tokens the agent generates. The curated block is generally shorter than the uncurated block, or at most slightly larger (Table 16). Where the store serves context, generation falls while accuracy is maintained (Appendix I).

Table 4: **End-to-end inject time** (200 queries, GAIA history, no LLM in the loop). Each layer runs its own retrieval; the curated median is over queries that serve context.

Layer	p50 (ms)	p95 (ms)	p99 (ms)
A-Mem	0.06	0.07	0.08
CuratorMem	1.31	1.52	1.86
Mem0	28.96	38.81	42.49

Table 5: **Read cost against chain length** (store held at 10,000 patches; medians over 50 operations; write p50 excludes curation LLM calls)

Entries per chain	Chains	Read p50 (ms)	Write p50 (ms)
10	1,000	0.20	18.3
100	100	1.27	18.7
1,000	10	11.13	17.1
10,000	1	120.11	18.4

Read cost is a near-linear function of the queried chain via chain-partitioned manifest operations: latency scales with chain length rather than total store size (Tables 5 and 6). Table 4 reports `inject` time; A-Mem, Mem0 and CuratorMem perform different retrieval operations, and the curated median is taken over queries that serve context (Appendix I.6).

Table 6: **Read and write cost at scale**, with the queried chain held fixed.

	queried chain held fixed			grows with the store			
Store (patches)	**Read p50 (ms)**	**Append (µs)**	**Time-travel (ms)**	**Index build (ms)**	**Save+load (ms)**	**Metadata (MB)**	**With embed. (MB)**
10^3	0.10	0.2	0.02	2	6	1.8	3.3
10^4	0.10	0.3	0.02	21	101	16.4	31.0
10^5	0.10	0.3	0.03	238	1353	164.4	310.9

Increasing the rest of the store has little effect on read latency when the queried chain is held fixed (Table 6), because each read selects a single chain before ranking. Index construction time and memory footprint increase with store size. The index is constructed once at load time and retained in memory for subsequent reads. In conclusion, CuratorMem is a store where accumulated history does not inflate the next read, and where curated context reduces generation while accuracy is maintained.

Robustness Analysis. On GAIA, scores are similar on the easy tier and differ more on the two harder tiers (Figure 4). On τ^2 the gate reads the simulated domain's final database state rather than a benchmark label (Table 11). On disagreed τ^2 tasks it leads all four arms (Appendix D.7).

4.3 Ablation Study

Tables 12 and 17 show that the margin over Patched stays positive over seven iterations, and that accuracy does not fall with evidence depth.

Table 7: **Effect of the certification signal.** Both panels vary only the evidence used to certify a stored attempt as successful. Differences are in percentage points; panel (a) re-scores recorded attempts.

		effect within benchmark		
Certified by	**Backbone**	**No mem.**	**Score**	**Δ**
the agent's own opinion	GPT-5.5	36.52	31.04	−5.49
	HY3	33.48	28.05	−5.44
metadata-grounded curation	GPT-5.5	31.57	**33.73**	+2.16
	DeepSeek	29.91	**33.56**	+3.66

(a) Self-certification underperforms the memory-free baseline; metadata-grounded curation improves on it. GAIA2. Separate windows; 31.57 matches Table 2.

	CuratorMem's margin over each		
Certified by	**A-Mem**	**Mem0**	**Patched**
the agent's own opinion	−4.53	−3.68	+0.00
no checkmark at all	+0.12	−0.23	+3.57
metadata-grounded curation	+5.31	+3.96	+8.42

(b) The certification signal affects performance relative to external memory layers. GAIA/HY3

To isolate the endorsement signal, we vary the evidence required for certification while holding the other components fixed (Table 7). Self-certification underperforms both the memory-free arm and the external layers, consistent with the risk that a model misclassifies its own failures as successes and then repeats them. Without certificates the arm makes no success claims, and performance is similar to that of the external layers. Requiring an independently recorded outcome produces the strongest results. The benefit depends on the evidence required to present an entry as successful.

Table 8: **Ablation on individual components of CuratorMem**, in drift-corrected points with a 10^4-resample paired bootstrap. Each panel reports one concurrent sweep, with a full benchmark run for each arm. Given the limited sample sizes, we report *certainty*, defined as the proportion of resamples with the same effect direction as the estimate.

Component	**Effect**	**Certainty**
Read policy (GAIA2/HY3, n=98)		
stricter overlap floor	−2.78 [−10.75, +4.95]	75.8%
lineage ordering over similarity	+1.41 [−7.55, +10.83]	61.6%
curated text over the raw patch	+0.63 [−7.57, +8.85]	55.7%
ranking on $\bar{\Delta}_c$	−3.98 [−13.02, +4.82]	80.8%
Certification (GAIA/HY3, n=162; only π_c differs)		
a grounded outcome over none at all	+7.41 [+0.00, +14.81]	96.9%
a grounded outcome over self-assessment	+4.32 [−3.70, +12.35]	82.9%
a self-assessed outcome over none at all	+3.09 [−4.94, +11.73]	74.0%
curated text, no certificate, vs. no memory	−4.24 [−10.91, +2.42]	91.3%

We ablate individual components by restoring each to Patched's setting while retaining the other CuratorMem settings (Table 8). The clearest effect concerns certification: the grounded-outcome condition exceeds the no-certificate condition by 7.41 points, with a reported interval whose lower bound rounds to zero. Self-assessment recovers part of this difference. Read-policy effects are less conclusive, including a near-zero estimate for curated text over the raw patch. Overall, the ablation evidence is stronger for certification than for any individual read-policy choice.

5 Conclusion

In this work, we introduced CuratorMem, a memory layer for agents that repeatedly attempt related tasks. Following the read–write decomposition of the Stateful Reflective Decision Process, the system maintains an append-only patch-based memory history: curation adds successor entries while preserving their predecessors, and an endorsement gate presents an entry as successful only when the outcome is recorded independently of the acting agent. Through evaluations across three backbones and four benchmarks, we demonstrated an advantage over an uncurated patch store and over external memory layers, under reworded reuse of the same item and on distinct questions on three LoCoMo conversations. Ablation studies further validate the effectiveness of endorsement, and these results highlight the importance of recording the evidence that supports a memory, alongside preserving its history and making it retrievable.

REFERENCES

Apache Software Foundation. Apache iceberg: The open table format for analytic datasets. https://iceberg.apache.org, 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2025.

Elzo Brito dos Santos Filho. ESAA-Conversational: An event-sourced memory layer for continuity, handoff, and curation across heterogeneous LLM coding agents. *arXiv preprint arXiv:2606.23752*, 2026.

Prateek Chhikara et al. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Xinyi Fan, Miri Liu, Ruozhen Yang, Siru Ouyang, and Jiawei Han. Can agent memory systems track evolving state? *arXiv preprint arXiv:2608.19652*, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025.

Tao Feng, Chongrui Ye, Tianyang Luo, Jingjun Xu, Xueqiang Xu, Haozhen Zhang, Ge Liu, and Jiaxuan You. ElasticMem: Latent memory as a learnable resource for LLM agents. *arXiv preprint arXiv:2605.30690*, 2026.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. In *International Conference on Learning Representations (ICLR)*, 2026. Oral. arXiv:2602.11964.

Kostas Hatalis, Despina Christou, Joshua Myers, Steven Jones, Keith Lambert, Adam Amos-Binks, Zohreh Dannenhauer, and Dustin Dannenhauer. Memory matters: The need to improve long-term memory in LLM-agents. In *Proceedings of the AAAI Symposium Series*, volume 2, pp. 277–280, 2023.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *International Conference on Learning Representations (ICLR)*, 2026.

Yuyang Hu, Shichun Liu, Yanwei Yue, et al. Memory in the age of AI agents. *arXiv preprint arXiv:2512.13564*, 2025.

Jiazheng Kang et al. Memory os of ai agent. *arXiv preprint arXiv:2506.06326*, 2025.

Yuxin Liao, Le Wu, Min Hou, Hao Liu, Han Wu, and Zishu Wang. LeanMem: Simple and efficient long-term memory for LLM agents. *arXiv preprint arXiv:2608.03463*, 2026a.

Yuxin Liao, Le Wu, Min Hou, Yu Wang, Han Wu, and Meng Wang. From atom to community: Structured and evolving agent memory for user behavior modeling. *arXiv preprint arXiv:2601.16872*, 2026b.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. In *Transactions of the Association for Computational Linguistics (TACL)*, 2024.

Enze Ma, Yufan Zhou, Wei-Chieh Huang, Jie Yang, Huanhuan Ma, Zixuan Wang, Chengze Li, Chunyu Miao, Philip S. Yu, and Zhen Wang. MemProbe: Probing long-term agent memory via hidden user-state recovery. *arXiv preprint arXiv:2606.24595*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*, 2024.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations (ICLR)*, 2024.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Chen-Yu Lee, Jiawei Han, and Tomas Pfister. Skillos: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*, 2026.

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.

Young Bin Park. Graph-native cognitive memory for ai agents: Formal belief revision semantics for versioned memory architectures. *arXiv preprint arXiv:2603.17244*, 2026.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

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.

Quentin Spencer. Ground truth first: A longitudinal evaluation instrument for agent memory, and the tenure crossover in memory-architecture rankings. *arXiv preprint arXiv:2607.21962*, 2026.

Yongye Su, Wujiang Xu, Chaoji Zuo, and Elisa Bertino. ChronoMem: Version control and semantic rollback for large language model agent memory. *arXiv preprint arXiv:2607.27773*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Jun Wang. Memento 2: Learning by stateful reflective memory. *arXiv preprint arXiv:2512.22716*, 2025.

Lei Wei, Xiao Peng, Xu Dong, Niantao Xie, and Bin Wang. Fademem: Biologically-inspired forgetting for efficient agent memory. *arXiv preprint arXiv:2601.18642*, 2026a.

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*, 2026b.

Zhaofen Wu, Hanrong Zhang, Fulin Lin, Wujiang Xu, Xinran Xu, Yankai Chen, Henry Peng Zou, Shaowen Chen, Weizhi Zhang, Xue Liu, Philip S. Yu, and Hongwei Wang. Gam: Hierarchical graph-based agentic memory for llm agents. *arXiv preprint arXiv:2604.12285*, 2026.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts LLM agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2026.

Haoran Xu, Jiacong Hu, Ke Zhang, Lei Yu, Yuxin Tang, Xinyuan Song, Yiqun Duan, Lynn Ai, and Bill Shi. SEDM: Scalable self-evolving distributed memory for agents. *arXiv preprint arXiv:2509.09498*, 2025a.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025b.

Tianyu Yang, Sudipta Paul, Vijay Srinivasan, Vivek Kulkarni, and Srinivas Chappidi. Trustmem: Learning trustworthy memory consolidation for llm agents with long-term memory. *arXiv preprint arXiv:2606.25161*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*, 2026.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, and Wenya Wang. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

Ningning Zhang, Xingxing Yang, Zhizhong Tan, Weiping Deng, and Wenyong Wang. Himem: Hierarchical long-term memory for llm long-horizon agents. *arXiv preprint arXiv:2601.06377*, 2026b.

Yuxiang Zhang, Jiangming Shu, Ye Ma, Xueyuan Lin, Shangxi Wu, and Jitao Sang. Memory as action: Autonomous context curation for long-horizon agentic tasks. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026c. arXiv:2510.12635.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

Zicheng Zhao, Xinyang Guo, Luyao Lv, Menghan Wang, Ming Li, and Shuaicheng Li. Accurate and efficient long-term memory for LLM agents. *arXiv preprint arXiv:2607.16211*, 2026.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng, Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026a.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning LLM agents without fine-tuning LLMs. *arXiv preprint arXiv:2508.16153*, 2025.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026b.

Wei Zhou, Xuanhe Zhou, Shaokun Han, Hongming Xu, Guoliang Li, Zhiyu Li, Feiyu Xiong, and Fan Wu. Are we ready for an agent-native memory system? *arXiv preprint arXiv:2606.24775*, 2026c.

Pengyuan Zhu and Ming Wu. Agent zero memory: Provenance-aware long-term memory for LLM agents. *arXiv preprint arXiv:2608.29606*, 2026.

Qiming Zhu, Shunian Chen, Rui Yu, Zhehao Wu, and Benyou Wang. From lossy to verified: A provenance-aware tiered memory for agents. *arXiv preprint arXiv:2602.17913*, 2026.

List of Appendices

A Reproducibility Statement . . . 14
B Ethics Statement . . . 14
C AI Use Statement . . . 14
D Experimental Details and Credibility . . . 14
D.1 Evaluation Metrics . . . 15
D.2 Weak Backbones . . . 15
D.3 Credibility Report . . . 16
D.4 The Gate and the Curation . . . 17
D.5 What the Read Path Serves . . . 18
D.6 How Much the Partition Depends on Its Key . . . 19
D.7 Running the Loop Without Gold . . . 19
E Detailed Comparison with Prior Work . . . 20
E.1 Patch-Based Memory . . . 20
E.2 Detailed Baselines Introduction . . . 21
E.3 Capability Comparison . . . 21
F Limitations and Future Directions . . . 23
G A Worked Example: CuratorMem in a Production Data-Lake Agent . . . 24
H Broader Impact . . . 24
I Cost, Scaling, Method and Implementation Details . . . 25
I.1 Scaling Analysis . . . 26
I.2 Store-Level Diagnostics and Controls . . . 26
I.3 Retrieval: Chain Scoping and Lineage Ordering . . . 27
I.4 What a Successor Inherits, Field by Field . . . 28
I.5 Curation: Refine, Critique, Enrich . . . 29
I.6 Implementation Overview . . . 29
I.7 Read–Write Loop . . . 29
I.8 The Two Objects Taken from Prior Work . . . 29
I.9 Prompts and an Injected Block . . . 30
I.10 Evaluation Harness . . . 31
I.11 Curation Module and the Three Arms . . . 31
I.12 Per-Agent Integration . . . 31
J Experimental Settings . . . 32
J.1 Run-to-Run Variance . . . 32

A Reproducibility Statement

Every number printed in this paper recomputes from a trace file released with it. The complete source code for the harness, the per-arm traces and the scripts that regenerate each table are available anonymously at `https://anonymous.4open.science/r/CuratorMem-8EFE`. That repository has two directories. `sourcecode/` holds the memory layer, the harness and the runners, with `REPRODUCING.md` describing how a sweep is launched against any OpenAI-compatible endpoint. `experiments_results/` holds the traces and the checkers: `by_table/` keeps one trace file per cell of Table 2, named by benchmark, backbone and arm, and `_paper/` keeps the scripts, of which `verify.py` recomputes every printed value from those files in a single pass. Appendix D describes which run produced each cell, the run-to-run spread the intervals should be read against, and what survives pooling. Appendix J gives the backbones, decoding settings and per-benchmark protocol; Appendix I.6 gives the read–write loop and the arm definitions in enough detail to reimplement them.

B Ethics Statement

This work involves no human subjects and no personally identifying data. Every benchmark is a public release used under its own licence, and no data was collected for this study. The one account of a deployment, in Appendix G, is given at the level of an API change and reproduces no customer content. We therefore identify no ethical concerns arising from the work itself. Two properties of the design that bear on responsible deployment, erasure from an append-only store and the propagation of an outcome recorded in error, are discussed in Appendices H and F. The authors declare no competing interests beyond the industrial deployment already described.

C AI Use Statement

In this work, we used generative AI tools to design and give feedback on the experimental protocol, to implement methods, to support qualitative and thematic data analysis, and to assist with translation. We have not used generative AI tools to generate synthetic data sets, to develop theoretical models or conceptual frameworks, to formulate mathematical claims, to provide critical ingredients for proving them, to assist in the writing of proofs, to propose or refine hypotheses, to clean or reformat datasets, or to interpret results.

Additionally, we used generative AI tools to summarise and analyse existing literature, to identify relevant literature, to format references, to create and edit software code, to create figures from data we had already produced, to draft parts of this paper, and to edit it for readability. We did not use them to suggest a structure for it, to propose a title or keywords, to formulate questions for surveys or interviews, to suggest experimental parameters, to transcribe recordings of research material, to create artifacts, to discover research topics or identify gaps, to brainstorm, or to source information.

We have reviewed all AI-assisted work. AI-assisted analysis code was checked by regenerating every reported number from the archived traces rather than by inspection alone, and the regeneration scripts are released with the traces (Appendix D.3). Every citation was verified against a bibliographic record, and suggestions that did not resolve to one were discarded. The authors formulated the research questions, decided the experimental design, and interpreted the results. Generative models also appear *inside* the system under study, as the refiner and critic of §3.4; this is a property of the method being evaluated, not of the writing of this paper. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

D Experimental Details and Credibility

This appendix answers four questions. First, §D.1–§D.2 say how each number is produced and on which backbones. Next, §D.3 says how far a single number can be trusted; run-to-run spread is in §J.1. Then §D.4–§D.6 say which part of the design produces the effect. Finally, §D.7, Table 17, and Appendices I.9 and D.2 say how far the method extends beyond Table 2.

D.1 Evaluation Metrics

Table 2 prints four scores. A cell cannot be recomputed from the traces without the formula that produced it. This section defines those four quantities as the harness computes them, in table order: GAIA exact match with an optional judge, the GAIA2 official judge, LoCoMo token-F1, and the τ^2-bench reward.

GAIA: exact match and LLM judge. Exact match is the better of the extracted short answer and the full response, $e = \max\big(\mathrm{EM}(\hat{a}, a^*), \mathrm{EM}(r, a^*)\big)$, so a correct full response is not lost to a failed extraction. The judge is called only when exact match is 0. If its score is at least τ_j=0.8, that score is used; below 0.8 the task stays 0:

$$\text{score} = \begin{cases} e & e > 0 \\ j & e = 0 \text{ and } j \geq \tau_j \\ 0 & \text{otherwise,} \end{cases} \qquad \tau_j = 0.8, \tag{12}$$

where $j \in [0, 1]$ is the judge's verdict. Table 2 prints e and this score as two GAIA numbers, so the ordering can be read with the judge and without it.

GAIA2: official judge. The judge is GAIA2's own: the same extraction of oracle and agent write-actions, the same dependency-ordered matching, and the same LLM adjudication of whether an agent action satisfies an oracle one. The reported score is the matched fraction of the oracle's write-events, the actions that change the environment rather than read it:

$$\text{score} = \frac{|\{\text{oracle write-events matched by the agent}\}|}{|\{\text{oracle write-events}\}|}. \tag{13}$$

Every arm is scored by this rule.

LoCoMo: token-F1. Both answers are lowercased and split into alphanumeric tokens. Overlap counts a repeated token with multiplicity: for a prediction $\hat{a}$ and gold answer g, $o = \sum_t \min\big(c_{\hat{a}}(t), c_g(t)\big)$, where $c_z(t)$ is the number of times token t occurs in z. Then

$$\text{precision} = \frac{o}{|\hat{a}|}, \qquad \text{recall} = \frac{o}{|g|}, \qquad \text{F1} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \tag{14}$$

with $\text{F1} = 0$ whenever $o = 0$ or either answer is empty. Token-F1 is a ratio of token counts, so it falls when an answer grows past the gold. Table 3 uses a different protocol and reports LLM-judge correctness J as well as token-F1; the two can rank the same pair of systems differently.

τ^2-bench: reward. The benchmark compares a simulation's final database state and the actions it took against the task's expected outcome and returns 1 or 0. A cell is the mean over tasks and simulations, and a simulation that issues no tool call scores 0 instead of the reward an untouched database would earn, to prevent inaction from earning full credit in some cases. The rule applies to every arm.

D.2 Weak Backbones

Table 2 uses backbones that can issue well-formed tool calls. This section asks whether the same ordering survives when the backbone cannot. We ran three sub-3B models under that protocol, served locally from vLLM in `float16` on two V100s (the card has no `bfloat16`). Every arm is scored by one external model. Only LoCoMo produces attempts the memory layer can still change; Table 9 reports it by question type.

GAIA, GAIA2 and τ^2 score an attempt on the actions it takes. That needs a well-formed tool call (name, argument names, JSON) on every turn. At this scale the models do not hold that format: a run collapses into missing or malformed calls before memory can matter, and every arm hits the same floor. A score collected there measures whether the backbone can call tools, not whether curation helps. LoCoMo asks for a short free-text answer and needs no tool call, so it is the one of the four benchmarks where a sub-3B backbone produces attempts the memory layer can still change.

Table 9: **LoCoMo per-task-type breakdown on sub-3B backbones**: token-F1 (%)

Backbone	Memory	Single 200	Incid. 112	Temporal 90	Multi 74	Open 21	**All** 497
Llama-3.2-1B	Raw	7.3	7.2	0.3	1.5	**9.6**	5.2
	A-Mem	13.4	13.8	12.4	5.9	4.6	11.8
	Mem0	26.0	12.7	44.6	16.9	4.1	24.1
	CuratorMem	**27.2**	**16.3**	**45.8**	**22.5**	3.1	**26.4**
Qwen2.5-1.5B	Raw	11.8	13.9	2.2	10.5	**20.9**	10.7
	A-Mem	22.4	22.4	21.7	18.5	10.8	21.2
	Mem0	**42.4**	25.4	50.2	**38.0**	18.4	38.3
	CuratorMem	41.0	**25.5**	**60.1**	34.9	15.4	**39.0**
Llama-3.2-3B	Raw	12.3	11.7	2.1	3.4	9.7	8.9
	A-Mem	18.1	14.6	11.1	10.9	15.4	14.9
	Mem0	46.5	**23.0**	**55.8**	**39.2**	**22.3**	40.8
	CuratorMem	**49.6**	20.5	55.7	37.8	20.6	**41.1**

CuratorMem is first in aggregate on all three backbones, and the per-type pattern matches Table 3 on the strong backbones. The gain is largest on *temporal* questions, where a version chain records the supersession, and where the memory-free arm is weakest (0.3, 2.2, and 2.1). Two types go the other way. *Open-domain* favours the memory-free agent on two of three backbones, and *incidental detail* favours Mem0 on the largest one. In both cases the answer is tied to no entity and no chain, so version history has nothing to supersede, and the curated block is extra text the agent has to read past.

D.3 Credibility Report

Table 2 reports unadjusted arm means. A live endpoint can drift during a run, so those means still contain the empty-store gap between arms. This section subtracts that gap task by task (Eq. (15)) and reports the residual in Table 10. Table 11 names the recorded outcome the gate used on each benchmark. Table 12 splits LoCoMo by how far the annotated evidence sits.

Table 10: **Drift-corrected effect, pooled across backbones**, in points (Eq. (15)), with a 10^4-resample paired bootstrap over tasks.

Bench	n	Iter-0	vs Raw	vs Patched	vs Mem0	vs A-Mem
GAIA	288	-2.1	$+9.7\,[+3.00,+16.32]$	$+10.4\,[+3.78,+17.02]$	$+6.1\,[+0.07,+12.09]$	$+11.6\,[+5.28,+17.72]$
GAIA2	296	$+7.3$	$+4.6\,[-0.54,+9.83]$	$+3.7\,[-1.80,+9.37]$	$+3.7\,[-1.85,+9.29]$	$+12.0\,[+6.87,+17.27]$
LoCoMo	100	$+2.6$	$+10.7\,[+2.00,+19.70]$	$+8.5\,[-1.90,+19.20]$	$+10.8\,[+2.30,+19.80]$	$+2.7\,[-6.00,+11.40]$
τ^2	239	$+0.9$	$+10.5\,[+2.51,+17.99]$	$+5.9\,[-1.26,+12.97]$	$+6.3\,[-0.84,+13.39]$	$+6.7\,[-1.26,+14.64]$

Served endpoints are non-stationary: load varies and providers refresh models during a run. To isolate the memory effect from endpoint drift in Table 2, we subtract the between-arm difference at iteration 0, where the store is empty and no memory is injected, so that the remaining difference appears only after the store has been written.

Table 10 applies that correction to the traces of Table 2. The subtraction is a paired contrast, so it is defined only when those traces share a task, a backbone, and a score on the same scale. Three protocol facts make them so. First, each cell of Table 2 is backed by one trace file per arm, and a released provenance manifest records the run that produced it. Second, tasks, variant, backbone and experimental setting are identical across arms. Third, after the per-benchmark scoring rule is applied, every task is on $[0, 1]$: exact match with the LLM-judge tie-break on GAIA and LoCoMo, native soft recall on GAIA2, and environment reward on τ^2-bench.

For each task t and comparison arm X, we subtract the drift defined as:

$$\mathrm{DiD}(t) \;=\; \big[C_K(t) - X_K(t)\big] \;-\; \big[C_0(t) - X_0(t)\big], \tag{15}$$

which cancels the run-level offset task by task. Each comparison column subtracts its own pair-specific empty-store gap, so after subtraction, the Iter-0 column is the mean of those four gaps on the benchmark, not the quantity subtracted from every column. Here n is the number of tasks scored in all four comparisons, summed across backbones that have a complete case, so the four columns of a row use a common task set. On LoCoMo the complete case is 100 questions from one conversation (one backbone). Run-to-run variation is reported in Appendix J.1.

When the iteration-0 offset is comparable to the final-iteration difference in Table 2, that difference is the run-level offset.

The residual in Table 10 is a difference of evaluation scores. The endorsement gate does not read those scores; it marks an entry as successful from the recorded outcome in Table 11.

Table 11: **What v_c is, per benchmark.**

	Signal Eq. (5) reads	**Produced by**	**Deployment has it?**
GAIA	benchmark gold label	the dataset	no
GAIA2	official judge	GAIA2's own harness	no
LoCoMo	benchmark gold label	the dataset	no
τ^2	task reward from the final database state	the environment	**yes**

Eq. (5) uses v_c only if the acting model did not write it; every row of Table 11 meets that test. On GAIA and LoCoMo the gate reads the dataset gold label, and J scores Table 2 later. On GAIA2 the gate and the evaluation share the official scorer, which is not produced by the backbone or the store. The exact-match row of Table 2 uses no judge.

A-Mem and Mem0 keep their native write path and never consult v_c. The vs-Mem0 and vs-A-Mem columns of Table 10 therefore compare a gated store to layers that write without that record. To separate that architecture from the gold label, we rerun the gate with no label on the write path, the curator, or the certificate (§D.7).

The four rows of Table 10 also differ in chain construction, which changes where a residual can arise. On GAIA, a fixed gold answer makes a chain a sequence of attempts at one *solution path*; most chains never produce a verified success. On GAIA2, the prompt already contains an application context three orders of magnitude larger than the injected block, and the measured effects are smaller. Figure 4 provides a separate difficulty-stratified view of the GAIA window.

The intervals in Table 10 resample tasks. Questions on the same memory chain are correlated, so we also resample whole chains. On τ^2, GAIA and GAIA2 a chain is one task, and the two procedures coincide. Resampling chains instead of tasks moves a reported interval by a median of 1.79 points and at most 3.63 points; no estimate changes sign, and every interval keeps the same relation to zero. On LoCoMo one chain is one conversation, and all 100 questions come from that conversation. Table 3 is that comparison: CuratorMem leads in each of the three conversations against the memory-free arm and against A-Mem.

Table 10 does not split questions by how far the evidence sits. The chained protocol cannot do that split either. Table 12 does it under LoCoMo's native protocol, by how many sessions separate a question from its annotated evidence. Accuracy of the curated store does not decline as that distance increases; accuracy of A-Mem does. The memory-free arm shows that the depth buckets are of comparable difficulty.

Table 12: **Accuracy against evidence depth** under LoCoMo's native protocol, by how many sessions before the conversation's end the question's annotated evidence lies (n=151/129/215) and on questions whose evidence spans several sessions.

	HY3				**GPT-5.5**			
System	recent	mid	deep	multi	recent	mid	deep	multi
No memory	9.3	10.1	10.2	6.5	15.2	19.4	13.0	9.1
Raw store	26.5	28.7	40.0	27.3	39.1	46.5	50.7	33.8
A-Mem	35.8	24.8	29.3	14.3	54.3	52.7	61.4	23.4
Mem0	43.7	**53.5**	51.6	36.4	54.3	60.5	66.0	42.9
CuratorMem	**52.3**	52.7	**52.1**	**40.3**	**57.6**	**61.2**	**69.8**	**45.5**

D.4 The Gate and the Curation

After a task is scored, curation rewrites the attempt into a stored record, and the endorsement gate (Eq. (5)) decides from v_c whether a later read may present that record as successful (§3). This section records three operating choices that section leaves open: which fields the rewriter may change, the threshold τ_s, and what a later correction of v_c does.

Which fields the rewriter may touch. The rewriter is an LLM call that could edit every model-written field. A read already stays inside one chain, so the next attempt is about the same entities and those names have to survive. An earlier setting generalized every field, including the causal lesson; 82% of served lessons then lacked the name the next attempt needed. We therefore rewrite only the

step list. Ranking never reads that list, and the renderer does not inject it. The lesson keeps the real names. Every curated arm in this paper uses this rule.

Where τ_s sits. The gate presents an entry as successful only if the recorded outcome reaches τ_s or beats the chain's first attempt. We sweep τ_s on the 32,922 scored attempts archived with this paper. At the operating point 0.5, 31.0% of attempts clear the gate and their mean outcome is 0.97. At 0.1, coverage rises to 38.7% and the mean falls to 0.83, because attempts below the midpoint now pass. At 0.9 the mean is 1.00 and coverage is 28.9%. We keep 0.5: lowering it further buys little coverage for the quality it costs. Each benchmark shows the same shape.

If v_c is wrong. A false recorded outcome is not the same failure as a missing one. The store never writes the gate's decision as a field. Eq. (5) is recomputed on every read from v_c and π_c. If a poisoned outcome is later corrected, the next read drops every endorsement that depended on it; no rewrite or migration is required. The correction is a new version next to the old record, so the poisoned entry remains auditable.

This does not detect the attack. Anyone who can write the environment's feedback channel can create a passing v_c, and the read path cannot tell it from a real one. The attacker must forge a recorded outcome, not merely a persuasive memory an LLM critic might accept. A deployment should therefore treat that provenance channel as security-relevant.

D.5 What the Read Path Serves

A later task calls `inject`. The store may already hold several versions of that chain. The read can return nothing, the newest version, an older version, or a version with a checkmark. The measurements below say which of those happened.

How often it returns anything. Figure 3 is the share of CuratorMem reads that returned any text. A read is empty when the chain has no certified entry and no attempt that ever scored above zero. On an empty store every read is empty. As chains then collect certifiable work that share rises, and by the last iteration it lies between 32% and 100%, depending on the cell. LoCoMo is already above zero at iteration 0: a chain is a conversation, so earlier questions are already stored when later ones are asked.

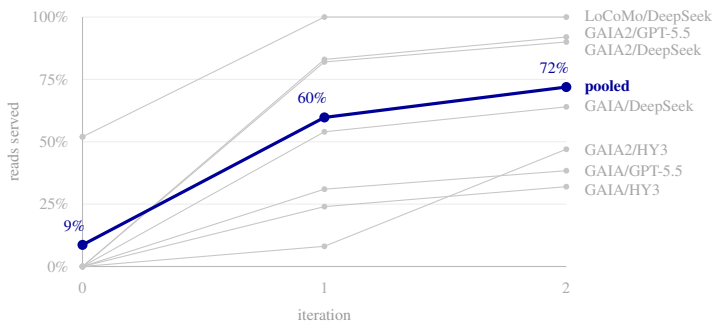

Figure 3: **Share of CuratorMem reads that returned any text** (*reads served*).

Which version it returns. Eq. (4) keeps a superseded entry addressable. Eq. (6) sorts a chain newest-first. We label each served entry by version. On a LoCoMo/HY3 window at n=100, every final-iteration read served the newest available version of the lineage it touched (57 of 57). Half of those reads also carried the immediate predecessor. The budget decides how far down the order a read reaches. A superseded entry ranks below its successor and remains reachable.

Whether the certificate filters. The certificate is a sort key (Eq. (7)). No entry is dropped from the candidate pool for lacking one. An uncertified entry is served if it reaches top-k. Silence is stricter than a missing certificate: if nothing is certified and nothing ever scored above zero, the read returns nothing. That is the silence in Figure 3.

Silence is not where the gain comes from. If CuratorMem won only on silent chains, the gain would be abstention, not curation. We split the GAIA/DeepSeek-v4 chains by whether anything was served at the final iteration. It leads in both halves. On the 64 chains where it spoke it scored 82.50, against Patched's 78.12 and Raw's 75.00. On the 36 silent chains it scored 30.56, against Raw's 22.22; Patched, which has no gate and still injects, scored 16.67. Curation still ran on the write path of those silent chains. The rescue-to-spoil ratio is positive in both halves: $8/5$ where it spoke and $7/2$ where it did not.

D.6 How Much the Partition Depends on Its Key

Eq. (6) restricts a read to the chain γ_t. In Table 2 that identifier is supplied by the harness; at deployment it is not. The residual reported there could therefore depend on an oracle key. This section measures that dependence on LoCoMo, where one chain holds a hundred distinct questions, under two degradations of γ_t: a corrupted key and a withheld key.

Table 13 reports both. Each panel is one LoCoMo/HY3 window run with all of its arms. *Effect* is the drift-corrected gain over that panel's own memory-free arm (Eq. (15)); *Parity* is the iteration-0 gap that correction subtracts; *Served* is the share of final-iteration reads that returned text. As in Table 8, each panel is a full re-run of every arm, so the windows are sized to detect a reordering rather than to resolve a small effect.

A corrupted key is produced by hashing the task id and relocating a fixed fraction of tasks, the same tasks on every iteration, from their own chain into one shared chain. Those tasks can no longer retrieve their own history and now sit among unrelated tasks. The corrupted panel is the complete-case intersection of the three key conditions (n=200). Relocating a quarter of the tasks leaves the residual essentially unchanged, from $+1.50$ to $+1.60$. Relocating half reduces it to $+0.00$ and lowers coverage from 99.5% of final-iteration reads to 62.8%. Iteration-0 parity remains near zero across the three key conditions.

A withheld key removes γ_t from the read. The resolver scores the cleaned situation against every chain already in the store and opens a new chain when no score clears a fixed threshold. The withheld panel is a separate window (n=100). Against a control that keeps the harness key on the same window, tasks, and variants, the resolver answers every final-iteration read, with residual $+4.00$ versus $+0.00$ for the control. Thus, the residual survives without a harness-supplied name when related attempts are assigned to the same chain.

Table 13: **Chain-key ablation on LoCoMo/HY3.** Corrupted panel: complete-case intersection of three key conditions (n=200). Withheld panel: a separate window (n=100).

Chain key	Effect	Parity	Served
Corrupted, n=200			
correct key	$+1.50$	-0.50	99.5%
a quarter moved	$+1.60$	-1.60	100%
half moved	$+0.00$	$+0.00$	62.8%
Withheld, n=100			
supplied by the harness	$+0.00$	$+1.00$	100%
inferred by the read	$+4.00$	-3.00	100%

The same resolver can be scored without running the loop, because LoCoMo's native release already provides ten conversations as ground-truth chains. We score each of 1,986 questions against the chains built from all the others and record the top-scoring chain. It is the true conversation 86.9% of the time. A further 6.7% fall below the threshold and open a new chain, so coverage drops but nothing is attached to the wrong conversation. The remaining 6.4% are filed under a different conversation. That misfiling is not produced by a loose threshold: only 0.04% of cross-conversation question pairs in the corpus clear it. That 6.4% is several times smaller than the share of tasks the upper panel of Table 13 can relocate without moving the residual.

D.7 Running the Loop Without Gold

The endorsement rule requires an outcome from a source independent of the acting model (Eq. (5)). For GAIA, GAIA2, and LoCoMo, this source is the benchmark evaluator; for τ^2, it is the environ-

ment reward (Table 11). Because the former is unavailable at deployment, this section identifies the label-dependent components of the write path and evaluates a label-free alternative.

What the record contains. A read ranks entries by version lineage, chain key, cleaned situation, and $\bar{\Delta}_c$. These execution-derived fields contain no reference answer. Only v_c records the outcome of an attempt. The rule does not require that a benchmark produce v_c; it requires $\pi_c \neq \textsc{self}$. Thus τ^2 applies the same rule to an environment return.

What reaches the curation prompts. The curation prompts receive the task description, recorded outcome, scalar score, executed steps, and version history, but never the reference answer. *Missing steps* and *failure reason* are derived from GAIA2's oracle-event list and are therefore populated only for GAIA2. Thus, curation itself remains label-free even when the gate's outcome originates from a benchmark evaluator.

The τ^2 reward. On τ^2 the signal is the simulated environment's reward, not a label. It is taken from the final database state. A simulation that calls no tool leaves that state unchanged and would collect the reward of the simulation that did the work. We score inaction as zero and record the tool-call count on every row.

Table 14: **Label-free gate signal on answer-scored benchmarks.** The signal is within-chain agreement, evaluated by re-scoring stored attempts.

	Really correct	Gate certifies	Of those, correct	Of correct, found
GAIA / DeepSeek-v4	60.5	33.5 (67)	**95.5**	52.9
GAIA / GPT-5.5	37.7	17.1 (34)	**97.1**	44.0
GAIA / HY3	25.9	17.8 (35)	45.7	31.4
LoCoMo / DeepSeek-v4	36.0	91.5 (183)	35.5	90.3
LoCoMo / HY3	33.5	89.5 (179)	34.1	91.0

Removing the label from the runs. On τ^2, no reference answer enters the write path, curator, or certificate because the gate already uses the environment return (§4.2). To probe the answer-scored benchmarks, Table 14 instead re-scores stored attempts with within-chain agreement. The substitute certifies agreement with the previous attempt, not correctness against the benchmark. Its certification count and precision are therefore a property of the candidate signal, not an estimate of a deployed gate. GAIA2 is absent because agreement certifies only one of 598 pooled attempts.

We then ran three label-free windows end to end. Two use within-chain agreement; the GAIA window issued seven certificates and served five. The third uses an independent, memory-free re-sample of the same task, preventing injected text from entering its own certificate; that signal admits 23% of final attempts. These windows contain 40 GAIA tasks and 100 LoCoMo questions, so they establish execution feasibility rather than an effect-size estimate. On disputed tasks, the curated store leads Patched 3:2 on GAIA; it takes nine of ten disputes against the memory-free control in the two LoCoMo self-agreement windows; and it leads Patched, Mem0, and A-Mem 4:2, 2:1, and 4:2, respectively, in the re-sample window.

E Detailed Comparison with Prior Work

§2 gives the high-level comparison; this appendix supplies the criteria needed to interpret it. It first distinguishes patch-based versioning from whole-store versioning, then explains why A-Mem and Mem0 are compared as complete layers, and finally defines the six tests used in Table 15.

E.1 Patch-Based Memory

Our closest design class records an update as a patch while retaining the state it replaced. Patch-based memory uses this unit of history: each patch records the pre-change and post-change state, its reasoning, and its motivating evidence.

Zhou et al.'s Memento (Zhou et al., 2025) is the case-based precursor: a frozen LLM adapts by writing and retrieving episodic cases over a memory-augmented MDP, and case selection is similarity search or a learned Q-function. Memento 2 and Memento-Skills (Wang, 2025; Zhou et al., 2026b)

keep a non-destructive case bank and retrieve by similarity under a value prior. patch-based memory instead retrieves along a task's version chain and stores every state as written. The Stateful Reflective Decision Process of Memento 2 (Wang, 2025) supplies the bound we build on (§3.1), whereas the earlier paper supplies the continual-adaptation framing, not that bound.

ChronoMem (Su et al., 2026) also preserves version history, but at a different granularity. It commits a snapshot of the whole memory at each write and resolves a natural-language undo request to one snapshot by lexical and semantic retrieval. Its unit of history is therefore the store, and its operation is rollback.

In contrast, we version an entry. A successor is appended beside its predecessor with an incremented version, so both remain addressable under their own $(\text{task}, \text{version})$ key and nothing is rolled back. A superseded entry is ranked below its successor and remains servable if the successor's next attempt fails.

The difference in unit also determines the gate: ChronoMem selects a snapshot to restore, whereas CuratorMem marks an entry as having worked from its recorded outcome and provenance (Eq. (5)). Both avoid in-place overwrite, but they use version history for different decisions.

Our Patched arm is that claim at entry granularity: the same append-only lineage and the same retriever code path as CuratorMem. The two arms still differ in written text, the endorsement gate, and the overlap floor (Table 19), so the main-table gap is their joint effect, not curated wording alone. A whole-memory snapshot scheme cannot separate that quantity: it has no per-entry successor to curate and no per-entry outcome to certify. Our comparison is with the class of design ChronoMem represents, at the granularity our store operates on.

E.2 Detailed Baselines Introduction

Table 2 compares against A-Mem and Mem0 as complete layers, not as a shared read policy on their stores. This section says how those layers were attached, and why the endorsement rate and the ranking key cannot be transferred onto them.

A-Mem (Xu et al., 2025b) builds interlinked notes and evolves them as related ones arrive. Mem0 (Chhikara et al., 2025) extracts and consolidates facts into a vector store. Both layers run behind the same `record`/`inject` interface as CuratorMem and are given the same external model for their internal LLM calls. A-Mem and Mem0 keep their native retrieval, so the comparison is of complete layers rather than of a shared read policy.

The endorsement gate is not what separates the layers. Over every entry each arm actually served, the fraction clearing our endorsement predicate falls within six points across the four arms: 31.0% for CuratorMem, 31.3% for Patched, 36.7% for A-Mem and 35.5% for Mem0. The gate therefore admits a comparable slice of every arm's output; what differs is the content of that slice.

Transferring the rest of the read path is not possible either, because Eq. (7) orders candidates by a version, a tool-sequence delta and a recorded co-occurrence term, and neither a note graph nor a compacted fact store records any of the three (Appendix D.7). A structured conflict-aware store such as MOSAIC (Zhao et al., 2026) organises contradictory entries on a typed graph and deletes the loser. That write is not an add, so it is not a drop-in behind the same `record`/`inject` interface, and the comparison in Table 15 is at the design level.

E.3 Capability Comparison

This is the table §2 cites as Table 15. It places CuratorMem beside memory and skill-learning systems on six capabilities that matter for an agent in a changing environment. Each mark is scored from that system's own description against the operational tests below, not as a ranking.

What each column means. A mark is ✓ only if the operational test holds. *Persistent store*: the system retains information across episodes, not only inside one context window. *Curated entries*: after a write, a stored item is rewritten, typed or enriched rather than left as the raw trace; a ○ is a one-shot rewrite of the current answer or a note link that is not a post-hoc curator. *Non-destructive*: every write is an add; no stored item is deleted, overwritten or compacted away (Eq. (4)); a ○ means some record types are immutable while others are updated in place. *Outcome-ranked*: retrieval orders can-

Table 15: **Capability comparison with prior work** on six operational tests (✓ the test is met, ○ a related mechanism that fails the test, ✗ none). The middle block holds structured, provenance-aware, conflict-aware and learned-budget systems. Marks follow the column tests in the text.

Method	Persist. store	Curated entries	Non-destruct.	Outcome-ranked	Evolution aware	Indep. critic
Reflexion (Shinn et al., 2023)	✓	✗	✓	✗	✗	✗
Voyager (Wang et al., 2023)	✓	○	✓	✗	✗	✗
A-Mem (Xu et al., 2025b)	✓	○	✗	✗	○	✗
Mem0 (Chhikara et al., 2025)	✓	○	✗	✗	✗	✗
Self-Refine (Madaan et al., 2023)	✗	○	✗	✗	✗	✗
Memento (Zhou et al., 2025)	✓	✗	✓	○	✗	✗
Memento 2 (Wang, 2025)	✓	✗	✓	○	✗	✗
SkillOS (Ouyang et al., 2026)	✓	✓	✗	○	○	✗
FadeMem (Wei et al., 2026a)	✓	✓	✗	○	✗	✗
GAM (Wu et al., 2026)	✓	✓	✗	✗	○	✗
HiMem (Zhang et al., 2026b)	✓	✓	✗	✗	○	✗
SEDM (Xu et al., 2025a)	✓	○	○	✗	✗	✓
TierMem (Zhu et al., 2026)	✓	✓	✓	✗	✗	○
Kumiho (Park, 2026)	✓	○	✓	✗	✓	✗
ChronoMem (Su et al., 2026)	✓	✗	✓	✗	✓	✗
LeanMem (Liao et al., 2026a)	✓	✓	○	✗	○	✗
Agent Zero (Zhu & Wu, 2026)	✓	✓	○	✗	✓	✗
AgeMem (Yu et al., 2026)	✓	○	✗	○	○	✗
MOSAIC (Zhao et al., 2026)	✓	✓	✗	✗	✓	✗
ElasticMem (Feng et al., 2026)	✓	✗	○	○	✗	✗
STEAM (Liao et al., 2026b)	✓	✓	✗	✗	✓	✗
CuratorMem (ours)	✓	✓	✓	✓	✓	✓

didates by a recorded attempt outcome (score, success, or the reuse delta of Eq. (8)), not by surface similarity, citation lock or a learned token budget; a ○ is a soft value prior or a downstream RL reward that is not stored on the entry. *Evolution-aware*: the design handles stale or superseded content by versioning, conflict detection, validity intervals or typed maintenance of changing records; a ○ is note evolution triggered by related writes without an explicit stale-handling rule. *Independent critic*: a writer-independent role scores the stored text, and that score never certifies the attempt (Eq. (5)); a citation lock, a same-policy RL reward, or the actor grading its own write is not this test.

Prior systems. The criteria distinguish three recurring design choices: similarity retrieval rather than outcome ranking, consolidation or deletion rather than full retention, and writer-side assessment rather than independent outcome evidence. The following systems instantiate those choices. Skill libraries such as Voyager (Wang et al., 2023) grow a reusable repertoire and verify new skills, but they retrieve by similarity and never re-score an entry as the world moves. Reflexion (Shinn et al., 2023) keeps verbal feedback across episodes, and that feedback is raw and self-produced. A-Mem (Xu et al., 2025b) and Mem0 (Chhikara et al., 2025) consolidate and link notes. Mem0 also compacts to stay within budget, so their writes can rewrite or drop earlier content. Self-Refine (Madaan et al., 2023) likewise improves the current answer but keeps no store between tasks. Both Memento (Zhou et al., 2025) and Memento 2 (Wang, 2025) keep a non-destructive case bank with a soft value signal. They store cases as produced rather than curating them or passing them through an independent critic. LeanMem (Liao et al., 2026a), separately, types dialogue into profile, event and source-grounded records, updates only evolving events, and allocates a query-adaptive retrieval budget on LoCoMo. Records stay source-grounded, but ranking is by query type rather than recorded attempt outcome. Agent Zero Memory (Zhu & Wu, 2026) makes origin, timestamp and an evidence pointer first-class and answers only under a citation lock. That is source attribution rather than outcome-ranked retrieval. AgeMem (Yu et al., 2026) learns write, retrieve, update and forget as tool actions, a learned alternative to the fixed rules in Eqs. (5)–(7), and those actions include deletion. MOSAIC (Zhao et al., 2026) organises a typed graph and deletes or updates a neighbor on conflict, so superseded content is removed rather than ranked below a successor. ElasticMem (Feng et al., 2026) spends a learned latent token budget per retrieved chunk. The bank can be frozen, but ranking is not a stored outcome. STEAM (Liao et al., 2026b) evolves atomic preference units from longitudinal user traces and consolidates them into communities. That is user-specific drift rather than a task-chain partition. Closest in spirit among the budgeted systems are SkillOS (Ouyang et al., 2026) and FadeMem (Wei et al., 2026a), which raise quality by merging or forgetting.

Position of CuratorMem. Under these definitions, CuratorMem is the only row fully marked for curated entries, non-destructive storage, and outcome-ranked retrieval together. This does not make the remaining systems incomplete implementations of our design: LeanMem and Agent Zero

Memory curate structured stores but rank by query type or citation-locked evidence. MOSAIC and STEAM handle contradiction and preference drift by updating or deleting neighbors. AgeMem and ElasticMem learn write, retrieve or budget actions instead of applying a fixed endorsement rule. The Memento line keeps a non-destructive case or patch bank without post-hoc curation of stored entries. Each of our patches is refined, critiqued and enriched by an independent critic, and retrieval is ordered by recorded version lineage with the outcome delta as a tie-break, so nothing previously retrievable is lost. Compaction helps under a tight budget; the same selection is still reachable by reversible re-weighting, which writes nothing irreversibly and leaves the store auditable (Appendix F). In the terms of §3, curation targets the retrieval-error term a decayed store inflates, without touching what the agent already learned.

Surveys, benchmarks and retrieval policy. The table compares store operations, not the broader evaluation landscape. Surveys map the surrounding space of memory, skills, protocols, and harness engineering (Zhou et al., 2026a; Hatalis et al., 2023). Position work then asks what a store built for agents rather than adapted to them would require (Zhou et al., 2026c). A learned or graph-walking read is orthogonal to the write-path comparison we measure. On the benchmark side, Evo-Memory (Wei et al., 2026b) benchmarks test-time learning against a self-evolving store, long interactive histories probe what survives many sessions (Hu et al., 2026), and others plant facts carrying validity intervals and state reversals (Spencer, 2026). LeanMem reports LoCoMo under a query-adaptive token budget, whereas our LoCoMo arm uses the same character budget L as the other layers (Table 2). MemProbe (Ma et al., 2026) audits whether a store recovers hidden user state after ordinary assistance, which is a complementary test of preference persistence that the repeated-attempt protocol does not score.

F Limitations and Future Directions

The results rely on a recorded outcome, an append-only store, a chain-scoped retriever, and a harness-supplied key. The following four limitations identify where each assumption ceases to hold and the extension it would require.

It needs a signal that an attempt went well. The gate reads a recorded outcome. If a chain never produces an attempt it can certify, the read falls back to the uncurated store and CuratorMem behaves like the raw patch baseline. Partial credit softens this: a chain that beats its own first attempt earns a certificate without clearing the absolute bar. Under a binary metric only the absolute bar is available. This is why a weak backbone is a different setting rather than a smaller version of a strong one (Appendix D.2).

It never deletes. The store keeps every version, so it grows with use. The prompt does not, since a read injects only the top k entries, but the disk does. We measure that cost at 10^5 patches in §I.1. The manifest can take a reversible compaction layer without changing the read path, but nothing in the current system reclaims space on its own.

The retriever and the critic are simple. Retrieval uses chain-scoped word overlap with lineage ordering, and the critic is a prompted LLM. It consequently does not merge disconnected chains. For a query that requires evidence across such chains, CuratorMem would drop $\gamma_c = \gamma_t$ in Equation (6), keep the overlap test, and apply Equation (7) on the union; the withheld-key resolver in Table 13 is the first step. A learned retriever (Zhou et al., 2026b) or critic is untested here.

The main table is repeated-task reuse. Table 2 revisits one item under rewording with a harness-supplied key. The other reported protocols address separate questions: distinct questions on three LoCoMo conversations, an inferred or corrupted key, and the gate signal appear in Tables 3, 11 and 13. Figure 1 is the production catalog example that motivates versioned retention. Calendar time is held fixed, so those tables do not score preference drift over real time. MemProbe (Ma et al., 2026) scores the store as a recoverable user-state artifact rather than as downstream task success; that is the audit for shifting preferences that STEAM (Liao et al., 2026b) studies on longitudinal traces. Spencer (2026) attach validity intervals and volatility classes to facts before any text is written, which is the harder version of this problem.

Where this goes next. The reported protocols cover reworded reuse of one task and distinct questions from three conversations. Extending them to real-time preference drift requires the operations that immutable code and data histories provide: commits and lineage, chain-scoped reads, recovery of earlier states, and historical queries (Apache Software Foundation, 2025). Table 6 reports the cost of those operations at 10^5 patches.

G A Worked Example: CuratorMem in a Production Data-Lake Agent

Figure 1 in the main text is a catalog change that left both API signatures in use. This section is the deployment that figure came from: the agent loop, the stored entry, and what the next read served after the change.

We ran a fleet of agents on a large-scale industrial cloud data platform. Each agent built a customer's data pipeline, registered the tables it created in the platform's catalog, and checked them with a query, reusing calls that had worked on earlier tasks. A catalog change then made one of those calls fail. The store kept returning the old call, and after every release an engineer added the new form by hand. Figure 1 is that chain. The agent's own loop is unchanged; the layer supplies the context and takes the outcome back:

```
ctx = mem.inject(task) + task.prompt
plan = agent.run(ctx)
score = validate(plan)    # table materialized + check query
passed?
mem.record(task, plan, score)
```

The write-time record of one attempt, and the curated entry it became after the catalog change, carrying its own outcome history instead of replacing it:

```
did:  catalog.create_table(t2, schema); ran validation SQL;
row counts matched
outcome:  success

approach:  register the table in the catalog, then run the
row-count validation
lesson:  create_table needs a partition_spec since catalog
2.0
avoid:  date_add is exclusive-bound since engine 3.x;
catalog write needs a scoped role
outcome history:  ok(1.x) -> create-fail(2.0) ->
ok(partition) -> ok(role+bounds)
```

At the iteration against catalog 3.x, the read is limited to this chain and serves that entry ahead of the request, so the two lines the migration invalidated arrive as constraints rather than as an approach to repeat.

H Broader Impact

An append-only, gated store changes both what an agent retains and what an operator can later inspect or erase. This section derives the resulting deployment obligations before separating them from the claims the benchmarks establish.

Where this applies. CuratorMem changes only the store's write and subsequent read, so it can be added to a running agent without a new training phase. It is relevant where experience accumulates against a moving target: a codebase, an internal process, a long-running assistant relationship, or a large-scale industrial agent deployment. In these settings, an incorrect deletion can affect later tasks and cannot be undone. The method does not apply to a one-off question against a fixed corpus, which has no chain to improve. Because the store is append-only and each change keeps its version history, rationale and triggering evidence, what the agent remembers stays inspectable: a developer

can trace a failure back to the patch that caused it and see which knowledge is current and which superseded. A store that compacts itself cannot be audited for what it removed.

Risks and mitigations. Retention creates two immediate obligations. First, a store that helps an agent retain a strategy across sessions can also help an adversarial agent retain one; access control and tenant isolation are therefore prerequisites. Second, an append-only store needs a deletion path rather than merely an unlisting operation.

Tenant isolation is a partitioning requirement. The chain's key must subsume the tenant, so that no read path can cross tenants even in principle. In our deployments the store is per agent fleet.

Poisoning is narrowed by the gate but not closed by it. Recorded provenance is therefore security-relevant metadata that a deployment must protect (Appendix D.4).

Long-horizon capability strengthens misuse by the same mechanism that strengthens use, so those safeguards sit at the agent level rather than in the store.

Deletion rights are the remaining issue, because unlisting an entry does not satisfy a deletion request. The append-only deletion primitive is crypto-shredding: per-principal encryption of patch content under destroyable keys. Destroying a key makes every patch it covers unreadable while the manifest's lineage stays intact. Because the key is scoped to the principal rather than the chain, one operation covers that principal on every chain their content reached. Embeddings, task keys, outcome values and lineage edges remain after shredding; the residue paragraph below is the deletion surface. A lineage query over an erased range still resolves, returning the shape of the history with unreadable content.

That leaves a residue to be judged: version numbers and supersedes edges, the outcome score, the tool-sequence deltas, the provenance tag, the chain key and the reuse statistics. Two of these can carry information about a principal. A chain key built from a task identifier may identify one, so a principal-linked workload must key its chains by an opaque handle. Entry embeddings are derived from the plaintext and are not covered by the payload key. Embedding inversion is a live attack, so key destruction bounds who can read a patch rather than what can be inferred from it. A deployment with real obligations would need shredding together with encrypted or discarded embeddings, minimised manifest metadata, access control and tenant isolation, and a retention policy for entries touching more than one principal. What the append-only design contributes is that each of those decisions stays auditable.

Scope of the evidence. Our benchmarks support a claim about curated, grounded, append-only memory under reworded reuse of the same item and distinct questions on a LoCoMo conversation. They do not establish the privacy and isolation properties above, which are design commitments a deployment must verify for itself.

I Cost, Scaling, Method and Implementation Details

Tables 4–6 report latency and storage at a high level. This appendix supplies the implementation details needed to interpret those measurements: prompt cost, longer chains, store growth, controls, and the read–write procedure.

What a read displaces. An injected block adds prompt tokens, but it can reduce the response needed to complete a task. On GAIA/DeepSeek-v4, CuratorMem prepends 71.8 tokens per task against Patched's 146.8 (Table 16). Relative to each task's iteration 0, responses on the 64 tasks that received memory fall from $1{,}500$ to 956 tokens, and fall on 43 of the 64, while exact match is 84.4% versus 82.5%. We therefore report the token trade-off over served tasks; the 36 silent tasks move in the opposite direction on both measures.

How to read Table 16. A layer that stays silent on a task is charged zero tokens for it. The three left-hand columns give each layer's whole block. The three on the right split CuratorMem's block into the primary channel, the avoidance channel and the header, averaged over served reads only. The two denominators therefore differ. The split columns sum to what a served read receives, not to the whole-block figure beside them. Against Patched, the curated block is smaller on GAIA and

GAIA2 and larger on LoCoMo. Neither arm's block is exactly L, because the budget bounds the curated text and the lineage footer follows it.

Table 16: **The memory block each layer prepends**, mean `cl100k` tokens per task over every scored row of one run (DeepSeek-v4).

	whole block				CuratorMem splits into		
	GAIA	GAIA2	LoCoMo		primary	avoid.	header
Patched	146.8	139.4	124.1	GAIA	164.0	0.0	11.0
A-Mem	105.5	106.5	292.7	GAIA2	85.5	0.0	143.4
Mem0	111.1	113.0	133.3	LoCoMo	44.0	121.5	40.8
CuratorMem	**71.8**	134.3	177.1				

How to read Table 17. The table extends the main protocol from three to seven iterations on GAIA/HY3 and uses the deduplicated complete-case trace. The memory-free arm was resumed and appended second rows for some tasks; consequently, only 35 of the 40 tasks have one scored row in every arm at every iteration. Every column uses those same 35 tasks.

I.1 Scaling Analysis

Never deleting concerns the store rather than the LLM. We therefore isolate two storage questions without an LLM in the loop: whether the retained history changes space growth and whether unrelated history changes read latency (Table 6). Figure 4 separately reports the GAIA result by the benchmark's difficulty strata.

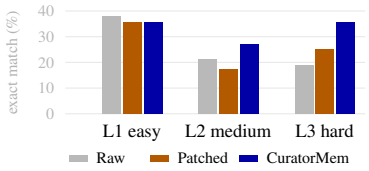

Figure 4: **GAIA exact match by the benchmark's own difficulty level** (HY3, final iteration, $31/52/16$ tasks at L1–L3)

Growth of the store. At matched sizes, the store holds comparable space to the alternatives. It sits slightly above the smallest layer at 10^5 patches because of the version lineage it keeps on top of the vectors. That lineage is what makes an earlier state recoverable at all. Index construction is linear and stays under a quarter of a second at 10^5 patches. Appending an entry and updating the index costs 0.3 microseconds and does not change across two orders of magnitude, because the partition index is a position dictionary and needs no rebalancing. The footprint at 10^5 patches is 311 MB, split near evenly between metadata and embeddings.

Read cost as the store fills. A read is pruned by chain. It consequently scales with that chain's length rather than with unrelated history (Tables 5 and 6). Holding the store fixed and redistributing patches from many short chains to few long ones isolates that dependence. Two design choices produce it. The write embeds each new entry so that no read has to, and the manifest keeps each chain's patches ordered so retrieval does no sorting at read time. Table 4 times inject only, so the three layers run different retrieval and the curated median is taken over the queries it serves.

Write cost. Writing an entry costs $m \approx 2$ to 3 short model completions: refinement and the critic run on every stored attempt, and enrichment runs only when the critic scores below τ_c. All of them happen after the task has been scored, so none is on the read path and none adds to the latency an agent waits for. The write p50 in Table 5 is the local append and embed; it excludes those completions.

I.2 Store-Level Diagnostics and Controls

A between-arm gap is interpretable as a memory-layer difference only when the remaining harness is shared. Table 18 lists each shared control and the alternative explanation it rules out.

Table 17: **Long iteration chains**: same protocol and tasks as the main runs, extended from three iterations to seven (GAIA, HY3).

Iter	Raw	Patched	CuratorMem	CuratorMem−Patched
0	22.86	22.86	25.71	+2.86
1	25.71	17.14	25.71	+8.57
2	22.86	20.00	25.71	+5.71
3	28.57	22.86	25.71	+2.86
4	28.57	20.00	25.71	+5.71
5	22.86	22.86	**31.43**	+8.57
6	22.86	17.14	28.57	+11.43

Table 18: **Controls, and what each one removes.** A-Mem and Mem0 keep native retrieval; the other rows are held identical across arms.

Held fixed	Setting	Removes the explanation
Interface	one `record`/`inject` path	"a different call surface"
Writer model	one external model for every layer's own calls	"a better writer"
Read-time dose	L below Patched's measured block size	"more context in the prompt"
Grader	one external critic and tie-break judge, never a backbone	"graded by itself"
Task text	one cached paraphrase set shared by all arms	"easier task wording"
Empty-store start	iteration-0 parity checked per sweep	"run-level drift"

One control deserves separate mention. Appendix J.1 gives seven repeats of a single condition and the concurrent identical arms alongside them, which together put the replication floor at roughly ± 5.8 points. A gap of a few points between arms in one run is therefore not evidence of anything on its own, which is why every table leads with the drift-corrected paired effect and its interval rather than with arm means.

I.3 Retrieval: Chain Scoping and Lineage Ordering

§3.3 defines the candidate pool and ranking key. This section specifies the resulting read path: eligibility, ordering, and the size of the returned block. It then reports the operating constants and the cases in which each key can decide.

Terms. A *chain* γ_t is the partition the workload declares, not a synonym for a task: GAIA, GAIA2 and τ^2 file each task on its own chain, whereas a LoCoMo chain holds one conversation's hundred distinct questions. The *cleaned situation* $s_t = \textsc{Core}(x_t)$ is the task text with run-specific detail stripped, and it is what the overlap test compares: a stopword-filtered Jaccard overlap on content words,

$$\mathrm{sim}(s_t, s_c) = \frac{\#\,\text{content words in both}}{\#\,\text{content words in either}}, \tag{16}$$

with a floor θ below which a candidate is dropped. The *injection budget* L caps the characters a read may send and k the entries it may draw on. The operating values are k=3, L=900 characters, θ_{B}=0.05 for Patched and θ_{C}=0.08 for CuratorMem. The similarity retriever multiplies overlap by an effectiveness weight $w_c = \mathrm{clip}(1 + 2\bar{\Delta}_c,\, 0.3,\, 1.5)$; that clip is not applied to the third lexicographic key of Eq. (7). The guarded read path of Algorithm 1 uses the lexicographic order and not w_c.

The read policy. Given these terms, resolving x_t takes five steps. *(i)* Candidates are the entries filed under γ_t. *(ii)* If any entry was recorded against x_t itself, those are taken alone. *(iii)* The floor θ is applied, then waived if applying it would empty a chain that has history. *(iv)* Survivors are sorted by Eq. (7) (endorsement, version, $\bar{\Delta}_c$, each descending) by a stable sort, so entries tied on all three keep the retriever's order. *(v)* The first k form S_t and the next $k-1$ form $\overline{S}_t$. The block is truncated to L with S_t written first, so truncation consumes avoidance entries before certified ones. Because Eq. (7) ranks endorsed entries first, $\overline{S}_t$ holds failed attempts whenever fewer than k entries are endorsed. If more than k are endorsed, the overflow endorsed entries occupy $\overline{S}_t$ as well; they remain marked by Eq. (5). A chain holding no certified entry and no attempt that ever scored above zero returns nothing (Algorithm 1). When the chain has uncertified attempts that scored above zero, those attempts are served without a certificate.

Why the policy is shaped this way. The fallback follows the purpose of the overlap floor. θ removes entries from unrelated tasks that share a chain; it is not a test of whether the chain's own his-

tory is useful. A reworded task can fall below the floor against its earlier entry, so enforcing it would erase all available context. The ordering then uses recorded facts rather than model judgements. Version is stored directly, and $\bar{\Delta}_c$ carries its provenance restriction inside the statistic (Eq. (8)), so a delta recorded under SELF is never averaged in. The critic's quality score q enters neither. That is where the design departs from the relevance-times-effectiveness form prioritized experience replay (Schaul et al., 2016) would suggest. Newest-first answers an environment that moves; it does not claim that newer is better. After a rollback the newest version's next attempt fails, that failure is recorded, and the read serves the older version, which is still stored (Appendix G). No entry is dropped for lacking recorded evidence, since that would cost coverage on the short chains with the least to spare. The lineage-ordering-over-similarity arm bounds what the whole ordering is worth (Table 8).

How often each key decides. Of the 403 reads that returned text, 227 contained more than one entry. In 81% of those, versions differ, so lineage fixes their order; only the remaining 19% leave a decision for $\bar{\Delta}_c$. This is expected where one entry is appended per iteration: GAIA, GAIA2 and τ^2 assign each task its own chain and therefore leave the later keys to LoCoMo. Newest-first does not mean newest-only: at the final iteration, predecessors and successors are served 98 and 97 times. Finally, $v_c > v^0$ is inactive under binary scoring because failure is zero; under graded scoring it prevents τ_s from certifying no attempt for a backbone below the absolute threshold.

Fallbacks and constants. Algorithm 1 is the read path: a dead chain returns empty context; an uncertified attempt that scored above zero is still served. The cross-chain fallback is declared from chain-key stability and is active on τ^2 alone. A startup gate rejects any block exceeding $L{+}130$ characters, so a misconfigured budget fails loudly. None of the constants was tuned against a reported number: $\tau_c{=}5$ and $\tau_s{=}0.5$ are the midpoints of the critic's rubric and of the normalised outcome, k and the reuse-weight clip are inherited from the underlying patch store, and L was fixed below the uncurated arm's measured block size so the curated arm cannot win by sending more text (Tables 16 and 19). The two overlap floors are the one value that changed during development, for a stated reason rather than for a score: an irrelevant lesson misleads in a way an irrelevant raw patch does not. Giving CuratorMem Patched's floor moves the outcome by -2.78 points, inside the replication floor (Table 8).

I.4 What a Successor Inherits, Field by Field

Eq. (4) says a successor is appended and the predecessor stays addressable. This section says, field by field, what that successor carries forward and what it may rewrite. Eq. (4) guarantees that every entry stays retrievable under its own identifier. What a curated *successor* carries forward from the predecessor it points to is a separate question, and the answer depends on which field is asked about. Write $\mathrm{content}(e)$ for the recorded fields of Table 1 together with the execution record the entry was built from: trajectory, intermediate states, tool sequence, outcome and provenance, version and patch history.

The execution record is inherited unchanged. Curation never writes to it, because refinement, the critic and enrichment all write into a separate taxonomy block, so the trajectory, tool sequence, intermediate states and recorded outcome are byte-identical before and after ρ. Every quantity the ranking key reads lies in this group, which is what keeps the order a function of what happened rather than of what was later written about it.

The predecessor itself is inherited as an address. A revision is a new entry carrying $\mathrm{ver}(e') = \mathrm{ver}(e) + 1$ and the predecessor's patch history extended by one record of what changed, while the predecessor stays in the store, reachable by its $(\text{task}, \text{version})$ key. That is what lets a read serve an older version after a rollback, and what a time-travel query resolves against.

Only the four model-written prose fields may change, and then only in one direction. Enrichment may rewrite them when the critic rates an entry weak, and the rewrite is accepted only if it is strictly longer than what it replaces, so a field never shrinks or disappears, though a longer rewrite need not contain the shorter one token for token. The ranking key reads none of these fields; they reach only what the served text says.

All three are properties of the store rather than of the read: a retained entry can still fall below top-k or be cut by L, which is why what reaches the policy is measured (Figure 3) rather than inferred.

I.5 Curation: Refine, Critique, Enrich

§3.4 names three write-time stages. This section writes the maps and the critic threshold, and states that a weak entry is enriched rather than dropped.

A finished attempt becomes an experience $e = \textsc{Analyze}(x, a, h, o)$, which the curation operator ρ transforms in the three additive stages of §3.4 before appending. Refinement rewrites the content in light of $H(e)$; the critic then scores actionability, generalizability, correctness and novelty:

$$q \;=\; \textsc{Critic}(\tilde{e}) \;\in\; [0, 10]. \tag{17}$$

A patch the critic finds weak ($q < \tau_c$) is *enriched* with the missing preconditions, failure modes and recovery steps rather than discarded:

$$e^{\star} \;=\; \textsc{Enrich}(\tilde{e}) \quad \text{if } q < \tau_c. \tag{18}$$

The enriched entry is stored without a second critic pass, so a write uses one refine, one critic call, and at most one enrich. Every stage appends rather than replaces and no entry is dropped for lacking evidence, so Eq. (4) secures *retention* by construction: no curation step can put a state the store once held beyond reach. What reaches the policy is a separate question, since the read path still ranks, truncates and may withhold. That quantity and the other two terms of Eq. (2) are engineering hypotheses established by measurement rather than consequences of the bound (§4.2, Figure 3).

I.6 Implementation Overview

The equations leave the online maintenance of the reuse statistic unspecified. This section fixes that choice, including the per-benchmark anchor. CuratorMem wraps the agent's existing patch store and changes only its memory read and write path.

The anchor $v^0(x)$ is always the memory-free iteration-0 score of the task x itself, recorded in the same run before the store held anything, so no counterfactual execution is needed. On GAIA, GAIA2 and τ^2 a chain is one task's iteration sequence, so the task anchor and the chain anchor coincide. On LoCoMo a chain holds a hundred distinct questions, so the two differ: the anchor stays per question, and candidates recorded against the current question order ahead of the rest. A session-mate's high score can therefore neither certify an entry through Eq. (5) nor raise its $\bar{\Delta}_c$.

The set R_c is the reads for which $c \in S_r \cup \overline{S}_r$ and whose outcome carries grounded provenance. Both channels receive the same δ_r. An entry selected into that set stays in R_c even if the character budget later truncates its text: credit follows the selected set, not the surviving characters. A write at iteration k is first available to ranking at $k{+}1$. When $R_c = \emptyset$, Eq. (8) sets $\bar{\Delta}_c = 0$; several such entries therefore tie on the last key and keep the order of the stable sort (Appendix I.3). The same δ_r is written on every member of served(r), so $\bar{\Delta}_c$ records co-occurrence on a jointly rendered block. Eq. (7) uses it only after endorsement and version. It is kept as a running mean; an entry whose recorded co-occurrence falls is re-ordered rather than evicted.

I.7 Read–Write Loop

A reimplementation needs a read procedure and a write procedure. Algorithm 1 is the authoritative read path. After scoring, the write path turns the finished attempt into an experience, refines it, has an independent critic score it, and enriches it when weak. It appends the successor beside its predecessor rather than replacing it; hence Eq. (4) holds. Performing this update after scoring ensures that both curation and the reuse statistic observe an actual outcome.

I.8 The Two Objects Taken from Prior Work

Two objects in the main text are taken from prior work rather than derived here. This section names them and says what we keep from each. The patch tuple of Eq. (1) is taken unchanged from Wang (2025); the fields this store adds around it are its own. That citation is Memento 2, not the 2025

Algorithm 1 CuratorMem read path: what the store hands an agent before it starts a task.

Require: the task x_t, the chain γ_t it belongs to, the store M, and a character budget L for what may be handed back

$s_t \leftarrow \textsc{Core}(x_t)$ ▷ cleaned situation of Eq. (6)
$\mathcal{C} \leftarrow \{c \in M : \gamma_c = \gamma_t\}$ ▷ Eq. (6), chain conjunct
if any $c \in \mathcal{C}$ was recorded against x_t itself **then**
  $\mathcal{C} \leftarrow$ those entries ▷ exact-task singleton
else
  $\mathcal{C} \leftarrow \{c \in \mathcal{C} : \mathrm{sim}(s_t, s_c) \geq \theta\}$ ▷ Eq. (6), overlap conjunct
  if $\mathcal{C} = \emptyset$ and the chain has history **then**
    $\mathcal{C} \leftarrow \{c \in M : \gamma_c = \gamma_t\}$ ▷ floor waived
  end if
end if
if nothing in $\mathcal{C}$ is certified **and** nothing in it ever scored above zero **then**
  return $\varnothing$ ▷ dead chain: empty, not a raw-store fallback
end if
sort $\mathcal{C}$ by Eq. (7), stable on remaining ties
$S_t \leftarrow$ the first k; $\overline{S}_t \leftarrow$ the next $k-1$ ▷ Eq. (9)
return $\textsc{Render}(S_t, \overline{S}_t)$ truncated to L, S_t written first

Memento agent of Zhou et al. (2025): the earlier paper formalises continual adaptation as memory read and write over a frozen LLM (an M-MDP with a case-selection policy), whereas Memento 2 supplies the SRDP operations and the patch fields reused here.

The second is the value-gap bound restated in §3.1. It does not strictly apply to a language agent on a moving benchmark, since it assumes a discounted MDP with bounded rewards, a fixed retrieval map, a policy conditioned on what that map returns, and a stationary horizon. We therefore take from it the separation of the three failure modes, not a derivation.

What Table 4 times. A-Mem retrieves links it built at write time, Mem0 re-embeds the query against its vector store, and CuratorMem embeds, ranks and renders. CuratorMem returns nothing where the chain holds no certified or measured entry, so its median is taken over the queries it serves.

What the gate excludes is a role, not a kind of grader. The external critic is independent of the actor, yet its quality score does not certify an entry either: a critic's judgement is an opinion where the gate reads a recorded outcome. What Eq. (5) excludes is the actor's own reading of its own attempt, whoever else is also consulted.

I.9 Prompts and an Injected Block

The main text names refine, critic, and enrich without quoting the templates. The verbatim strings live in the anonymous release (`https://anonymous.4open.science/r/CuratorMem-8EFE`, `sourcecode/curatormem/refine.py`): `AI_REVIEW_PROMPT`, `CROSS_AGENT_EVAL_PROMPT`, and `CRITIC_REFINE_PROMPT`. This section records only the constraints a reimplementation must keep.

Refine. The prompt receives the task, outcome, score, steps, and version history, and returns JSON with `generalized_steps`, `causal_lesson`, `avoidance_note`, `transferability`, and `evolution_insight`. Placeholders are allowed only in `generalized_steps`; every other field keeps the concrete names. The lesson must name a mechanism rather than restate success, and no step may be dropped or compressed.

Critic. An independent call scores actionability, generalizability, correctness, and novelty on a 0–10 rubric. Tautological lessons, compressed steps, and overfitted identifiers zero the corresponding axis and force `low_confidence`. The critic also emits an `outcome_verdict`; Eq. (5) does not read it.

Enrich. If $q < \tau_c$, a third call must return a strictly longer record: the original steps plus preconditions, failure modes, and recovery actions. Nothing is deleted.

The harness rewords a task only at the surface: facts, numbers, entities, constraints, and the expected answer are held fixed. A rewrite shorter than half the source is rejected; above 4000 characters only the trailing question is reworded.

The injected block is what READ returns, taken unaltered from a LoCoMo chain in the released traces. The first entry carries a certificate because the environment recorded its outcome; the other two are served without one, marked as recorded rather than verified:

```
## Curated prior attempts (this task's chain)

[✓ Prior attempt v1 -- environment-verified outcome]
Task:  What movie did Joanna watch on 1 May, 2022?
Verified answer:  The Lord of the Rings

[Prior attempt -- store version v1]
Task:  When did Joanna hike with her buddies?
As recorded:  not found

[Prior attempt -- store version v1]
Task:  When did Joanna finish her first screenplay?
As recorded:  last Friday
```

The distinction the block draws is the one Eq. (5) makes: only the first entry is presented as something that worked, and only because a party other than the actor recorded that it did. What the actor merely believes it got right reaches the model as "As recorded": a claim about what is stored, not about what is true.

I.10 Evaluation Harness

Every table that reports an iteration chain uses the same harness. This section defines the two accuracy aggregates and the threshold that counts an iteration as correct. The harness runs the arms over iteration chains. Each task runs K times, and for Patched and CuratorMem the patch is written after the task is scored, so iteration t reads what iterations $0..t-1$ wrote. Tables report the final iteration; the chain-level form below is used only in the per-chain diagnostics,

$$\begin{aligned} \mathrm{Acc}_{\mathrm{final}} &= \Pr_X[\,\text{last iteration correct}\,], \\ \mathrm{Acc}_{\mathrm{chain}} &= \Pr_X[\,\text{all iterations correct}\,], \end{aligned} \tag{19}$$

where an iteration counts as correct when its score is at least $\tau_{\mathrm{acc}}{=}0.5$. Here τ_{acc} is an evaluation threshold, not the curation map ρ of Eq. (11).

I.11 Curation Module and the Three Arms

Table 2 has three of our arms plus two external layers. This section says what those three arms share and what the environment variable actually switches. Raw, Patched and CuratorMem share one store implementation and one retriever code path, while A-Mem and Mem0 keep their native retrieval. Raw never writes; Patched records the raw patch and serves it verbatim; CuratorMem adds the curation stages on the write path and the endorsement gate on the read path. Between Patched and CuratorMem the only differences are what is written, what may be served, and the overlap floor (Table 19); the arm is selected by an environment variable and nothing else changes.

I.12 Per-Agent Integration

Each benchmark already has an agent loop. This section records that CuratorMem does not replace that loop: it adds one call before the task and one after. Each benchmark keeps its own agent loop, unchanged: CuratorMem places a context block at the start of a task and curates after it, one call at each end of the loop.

J EXPERIMENTAL SETTINGS

This appendix specifies the protocol behind Table 2: benchmark selection, shared task variants, execution limits, and the constants in Table 19. It evaluates two question-answering benchmarks (GAIA and LoCoMo) and two interactive tool-use benchmarks (GAIA2 and τ^2-bench); the corresponding algorithms are in Appendix I.6.

Runs use temperature 0. Within a condition, all arms share the agent, backbone, budget, and cached paraphrase set; only the memory path varies. Successive GAIA variants have median content-word Jaccard 0.57 (n=100), so writing and retrieval differ in approximately half of their surface form. Under this protocol, GAIA produces 300 entries and 39 of its 100 chains change outcome across iterations. Execution is limited to 48 tasks globally, 2 Docker-backed tasks, and 6 tasks per benchmark. Each arm writes a trace recording code revision, critic identity, and score provenance; these traces form the manifest in Appendix D.3.

Table 19 gives the curation constants and the per-benchmark protocol, and Table 18 what is held equal across arms. Two entries of the latter need a word here. A-Mem and Mem0 keep their native retrieval, because imposing ours would test our read policy on their representation rather than compare the layers. The overlap floor differs between the two patch arms by design, 0.05 for Patched against 0.08 for CuratorMem, since the two measure overlap against different text. Appendix I.3 prices that asymmetry rather than declaring it.

Table 19: **Settings, shared across all arms.** The upper block is the method's retrieval, read-path and curation constants; the lower block is the per-benchmark protocol, where K is the number of attempts per task and iteration 0 is the empty-store parity check.

Hyperparameter	Value	What it controls
Retrieval		
top-k	3	patches served per task (Patched and CuratorMem)
overlap floor θ	0.05 / 0.08	relevance gate for Patched / CuratorMem
reuse-weight clip	$[0.3,\ 1.5]$	bounds on the similarity pool's reuse weight
Read path		
injection budget L	900 chars	bounds the curated block; the lineage footer may follow it, and the gate rejects past L+130
cross-chain fallback	τ^2 only	chain lookup empty ⇒ similarity pool over all of M
Curation		
critic threshold τ_c	5 of 10	below this, a patch is enriched rather than dropped
success threshold τ_s	0.5	an outcome at or above it may be endorsed
Per-benchmark protocol — benchmark, agent, sample, K, metric		
GAIA	100 tasks, K=3	multi-step search; exact match with a judge tie-break
GAIA2	100 tasks, K=3	ARE tool-loop; official judge
LoCoMo	100 chained / 497 native, K=3	A-Mem agent, top-15 notes; token-F1, J under the native protocol
τ^2-bench	40 airline × 2 trials, K=2	tool agent + user simulator; task reward (database + communication)

Benchmark loader. One loader serves every benchmark, so the agents and the curation module see a single task interface: each per-benchmark routine returns `{task_id, description, expected, context, metadata}`: the gold answer or oracle events in `expected`, background in `context`, the chain key, difficulty and category in `metadata`.

Backbones. Reasoning effort changes a served model's behaviour enough that a run must state it. HY3 (high) is a proprietary reasoning model on an internal OpenAI-compatible gateway at `reasoning_effort=high`, set explicitly because the platform's default shifted during the project; GPT-5.5 (low) is a commercial reasoning model on a third-party gateway at the low effort tier; DeepSeek-v4 (`deepseek-v4-pro`) runs through the vendor's API. All three are served rather than self-hosted, so we give identifiers, effort tier and decoding settings. For reproduction without gateway access, Appendix D.2 reruns the protocol on three open sub-3B models served from vLLM.

J.1 RUN-TO-RUN VARIANCE

Intervals around the residuals in Table 10 must be interpreted relative to variation in an unchanged configuration. This section estimates that reference variation. For GAIA/HY3 at n=100, its scale is approximately $\pm$5.8 points.

Table 20 obtains five estimates. Four come from iteration 0: because the store is empty and no arm injects anything, arms in a sweep are the same program on the same tasks, and their disagreement estimates noise. The fifth is a deliberate seven-fold repeat of the memory-free arm on the same 40 GAIA tasks.

Table 20: **Run-to-run spread**, in points on each benchmark's own scale.

What was repeated	Where	Quantity	Points
Free: arms that were the same system at iteration 0			
Raw and Patched	GAIA2/HY3	sd of per-task differences	10.6
Raw and Patched	τ^2/HY3	sd of per-task differences	46.1
six arms of one sweep	GAIA2/HY3, $n{\approx}100$	range of arm means	4.58
five memory-free arms	GAIA/HY3, $n{=}100$	median interval width	12.2
Deliberate repeat			
memory-free arm, $\times 7$	GAIA, $n{=}40$	range of arm means	11.99

The five concurrent memory-free arms define the reference floor. Their pairwise 95% intervals have median width 12.2 points. Accordingly, at $n{=}100$, an effect smaller than approximately ± 5.8 points is not distinguishable from this variation.

We do not read a direction from a single GAIA/HY3 cell. Direction is instead tested on the pooled effect in Table 10, where each comparison uses at least 100 tasks. The interval widths in Table 10 are not caused by the drift correction: arms that we know are identical produce intervals of the same width.

The same floor shows up in the rest of the table. Of the 40 drift-corrected comparisons in Table 10, 33 are more than twice the six-arm range and 25 are more than four times it. Two additional $A/B/C$ runs at $n{=}148$ also fall inside the floor. And the movement is concentrated rather than spread over all tasks: of the 37 tasks scored in all seven repeats, 27 never crossed the threshold.

The τ^2 row is much larger than the others, at 46.1 points. This comes from the metric, not from instability. The τ^2 reward is binary, so every task scores 0 or 1 and a few tasks decide the mean. We therefore count the tasks on which two arms disagree instead of taking the difference of their means. We also reran the whole τ^2 condition from scratch. It reproduces the direction we report: the curated store finishes 5.00 drift-corrected points ahead of the memory-free arm and wins the disputed tasks seven to four, with iteration-0 parity of 2.50.

One consequence for the ablation. Table 8 compares six arms that ran at the same time. We therefore read each row against that sweep's own CuratorMem$-$Patched margin of $+0.18$, not against the $+3.7$ on GAIA2 in Table 10.