TRACE: RECALL AS EXECUTION FOR ACCOUNTABLE MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents that work across sessions must remember what matters, apply updates, and honor deletion and access requests; retrieved text cannot say whether a fact is still valid or permitted. We introduce TRACE, which implements Recall-as-Execution: each write records an operation (add, replace, delete), and each recall executes them to decide which facts hold at the requested time for the requesting principal. Every structured answer carries a certificate: the instructions it rests on and why each withheld fact was excluded. Temporal queries, forgetting and counterfactual replay are one call with a different time or ledger. We prove execution deterministic and grouping-independent, and check it against the deployed executor. On fixed ledgers all 659 recalls are admissible with no misuse, against 0.238–0.826 for a dense reader, and all 435 evaluated certificates verify, against 0.000 for baselines reporting no exclusions. On AUTOMEMORYBENCH, TRACE satisfies task success, safety and temporal validity together on 4.0% of cases, against at most 0.1% for any full-lifecycle vendor. A matched control locates the margin: folding the same operations into present state matches our admissibility, so the gain is in executing updates, not our format; but present state alone caps time-indexed queries at 0.333, where executing the ledger answers 1.000. Ablating supersession or governance collapses safety to 0.253 and 0.000 while task accuracy holds near 0.92: the result is in the execution semantics, not the backbone.

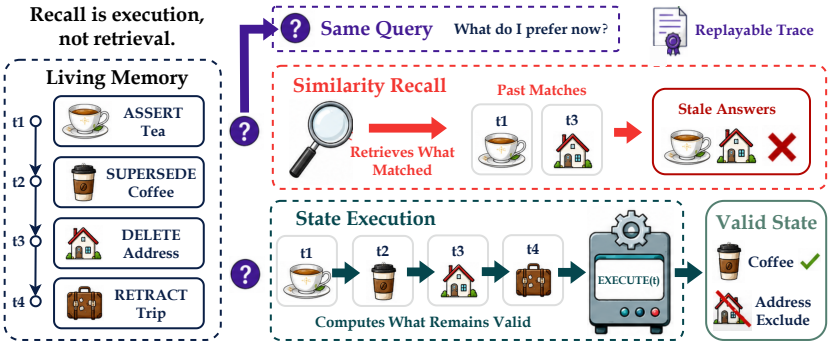

Figure 1: **TRACE at a glance.** Retrieval ranks past text. TRACE compiles writes into a typed ledger and returns $\mathrm{execute}(L, t, \mathrm{principal})$, the state in force at t.

1 INTRODUCTION

Agents that work across sessions rely on memory to carry information forward, and that information changes: a user moves, a policy expires, a shared detail becomes private. What the agent may use at any moment is therefore not what it once recorded, but what survives everything recorded since. Recall is an admissibility question before it is a relevance question, and the two come apart: an old address matches an address query as well as its replacement, and a deleted credential sits in a passage that otherwise answers it. Settling such a case takes three things a similarity score does not carry: the history of updates, the rules governing their use, and an explanation that can be checked against both. Improving recall does not by itself make a memory system reliable.

Existing work supplies pieces. ❶ Memory organization and retrieval (Chhikara et al., 2025; Xu et al., 2025; Fang et al., 2025) select and consolidate information, but relevance does not establish that a

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.

fact is still permitted. ❷ Temporal and conflict-aware stores (Rasmussen et al., 2025; Volkov et al., 2026) track change or keep competing claims. ❸ Transactional memory (Li et al., 2026) validates belief updates and gates when they drive actions. Each improves what is stored; none makes the read accountable, because what may be used is still decided inside a retrieval score, where it can be neither stated nor checked. Recall-as-Execution takes that decision out of the score: record every change as an operation, execute the operations for the requested time and principal, and return the answer together with the evidence of that execution. Membership then has a derivation, and time and principal are arguments to it, so historical and counterfactual questions are the same procedure.

TRACE implements this design (Figure 1). An LLM compiler turns incoming events into instructions that add, replace, withdraw, delete or accumulate facts, each keeping its timestamp and source evidence in an append-only ledger. Nothing is overwritten: a deletion is itself an instruction, so forgetting changes what may be used without destroying the record of why. At recall time an execution engine applies the recorded changes and the access restrictions to produce the state available to that query, within which the read interface selects current values, orders events, computes aggregates or ranks relevant facts. The same run emits a certificate carrying the supporting instructions and the reason for every relevant exclusion, so a verifier can replay the structured answer.

Our experiments measure whether recalled facts are admissible and whether their evidence can be verified. On fixed ledgers TRACE answers all 659 admissibility queries with a fact that is both still valid and permitted, where a dense retrieval reader over the same ledgers scores between 0.238 and 0.826, and it returns a verified certificate on all 435 evaluated recalls, against 0.000 completeness for every retrieval baseline, none of which explains what it withheld. Against exhaustive search on 1000 constructed programs the slicer returns the minimum evidence slice every time, with a complete exclusion report, in the live regime and in a stress regime whose targets have been deleted or superseded (§5). On the memory-requiring slice of AUTOMEMORYBENCH (Wu et al., 2027), TRACE satisfies task success, safety and temporal validity together on 4.0% of cases, against at most 0.1% for every vendor baseline with full coverage of the memory lifecycle.

❶ We introduce TRACE, which implements Recall-as-Execution by compiling memory updates into typed instructions and executing them into the state in force at the queried time, so what may be used is computed rather than inferred from a similarity score.
❷ We specify a single recall interface that returns supporting evidence together with an explanation of what was excluded, state its soundness, minimality, and completeness contract, and require a compliant verifier to refuse any answer whose certificate fails a check.
❸ We evaluate memory accountability on its own axis, separate from recall accuracy, and trace it to the execution semantics: ablating supersession or governance drives safety to 0.253 and 0.000 while task accuracy is unchanged, and every retrieval baseline we run scores 0.000 on certificate completeness, having no exclusion interface to score.

2 TRACE: MEMORY AS AN EXECUTABLE LEDGER

TRACE represents memory updates as typed instructions and answers queries by executing them to reconstruct an admissible state (Figure 2). A write boundary compiles incoming events into an immutable ledger; at recall, the membership engine resolves updates, temporal validity and authorization before a projection reads the state, and that same execution supplies the certificate evidence. Nothing in this pipeline ranks text. Membership is decided by the recorded operations, and relevance only orders what execution has already admitted, so a fact that was superseded, deleted or put out of scope cannot be surfaced by being similar to the query.

2.1 WRITE BOUNDARY

The write boundary converts an event stream (emails, chats, calendar edits, tool traces) into typed memory updates. An LLM compiler assigns each update an operation (ASSERT, SUPERSEDE, ACCUMULATE, DELETE, RETRACT) and two views: a *state view* carrying the subject, attribute and operation that resolve membership, and a *language view* retaining the source utterance and provenance that express and audit the answer. A write gate checks typing, duplication and authorization before appending, so every transition that execution sees is already well typed and unique.

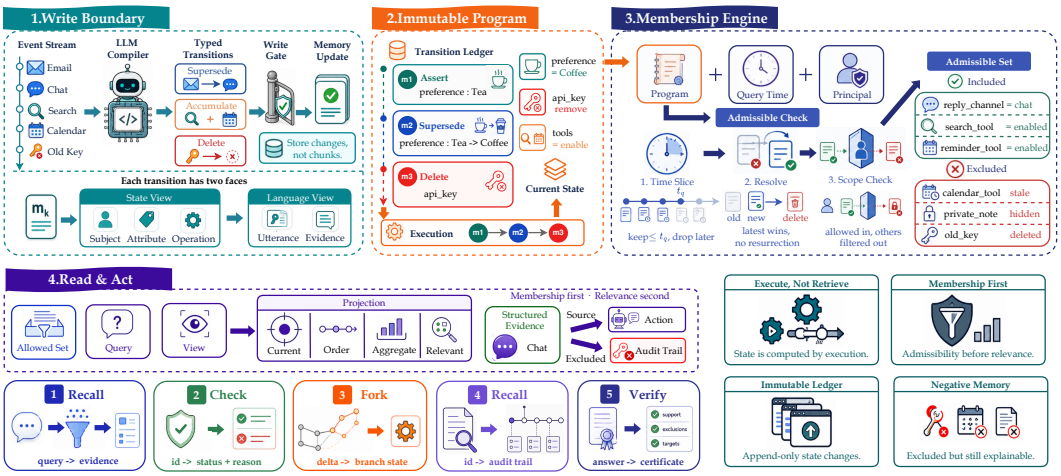

Figure 2: **The TRACE framework.** Writes become ledger instructions; execution resolves admissible state for answers and certificates.

2.2 TRANSITION LEDGER

The ledger stores admitted updates in an append-only sequence. An event time records when an update applies in the world and a transaction time when it entered memory, so recall can separate the history of an event from the history of what the system knew about it.

Definition 1 (Instruction ledger)**.** A ledger $L = [i_1, \ldots, i_n]$ is an append-only sequence of typed instructions. Each instruction carries a target entity/attribute, a payload proposition, an *event time* t_e when the change takes effect in the world, an optional bound t_v^+ closing the valid interval $[t_e, t_v^+]$, a *transaction time* t_x ordering ingestion, and an optional authorization scope. The instruction types are ASSERT (introduce a proposition), SUPERSEDE (replace the current value of an attribute), RETRACT (withdraw a prior assertion), DELETE (logical exclusion, not physical erasure), and ACCUMULATE (append into a deduplicated slot).

No update erases an entry; a DELETE records a timestamped change in admissibility, so forgetting is auditable. Recall folds them with $\oplus$, whose algebra makes execution deterministic:

Proposition 1 (Monoid on states)**.** $(\text{State}, \oplus, \varnothing)$ is a monoid: $\oplus$ is associative and $\varnothing$ is a two-sided identity. Instructions act on states via a lift $\hat{\imath}(\mathcal{S}) = \mathcal{S} \oplus \tau(i)$, where τ maps each instruction to its state-transformer. Under this lift, executing a prefix $i_1, \ldots, i_n$ is the monoid fold $\tau(i_1) \oplus \cdots \oplus \tau(i_n)$.

Proposition 2 (Execution equals fold; grouping)**.** EXECUTE is the monoid fold of the instructions selected at t, taken in transaction order, so the resolved state depends on that selected set only and not on how it is grouped. Fixing (t, q, z) therefore fixes a unique answer $\pi_q(\mathcal{S}(t))$. Appending one instruction can be applied incrementally by a single $\oplus$-merge; the wall-clock cost of that merge is bounded by the per-attribute state representation, independently of $|L|$ (Appendix D.1).

Associativity lets an existing fold be extended when new instructions arrive. §3 states the verification coverage for Propositions 1–2 and for the specifications below.

2.3 MEMBERSHIP ENGINE

The membership engine takes a ledger L, query time t, and requesting principal and returns the state available to that query. Each instruction carries two stamps: an event time t_e, when the change holds in the represented world, and a transaction time t_x, when it entered memory. The engine uses them for different jobs. Execution has three steps: (1) **Time Slice** keeps the instructions with $t_e \leq t$, so t is the point in world time the query asks about; (2) **Resolve** folds that subsequence in t_x order, so a later write wins over an earlier one on the same key; and (3) **Scope Check** applies expiry and authorization to the resolved propositions. Selecting before the fold rather than filtering after it is what lets a query reach a value that a later instruction has since replaced or deleted (Appendix D.3). The engine records an exclusion reason for every withheld proposition (`deleted`, `retracted`, `stale`, `unauthorized`), so a later audit can replay the decision.

EXECUTE$(L, t, q, z) \rightarrow (a,\ C = \langle \sigma, \rho, a \rangle)$ **in:** ledger L, query time t, query q, auth scope z

(1) Time slice.
1 $L_{\le t} \leftarrow [\, i_k \in L : t_e(i_k) \le t\,]$, in t_x order
(2) Resolve. $\oplus$ applies SUPERSEDE/ RETRACT/DELETE and keeps each entry with its admissibility metadata
2 $R \leftarrow \varnothing$; **for** $i_k \in L_{\le t}$ in order:
  $R \leftarrow R \oplus i_k$
(3) Scope check on the resolved map R
3 $\rho \leftarrow [\,]$; $\mathcal{S} \leftarrow \varnothing$
4 **for** each $p \in R$ with key $(\text{subj}, \text{attr})$:
5 **if** p inadmissible at t under z
6 (deleted, retracted, outside valid time, or out of scope):
7 append $\langle \text{key}(p), \text{reason}(p), \text{id}(p) \rangle$ to ρ # no content
8 **else** $\mathcal{S} \leftarrow \mathcal{S} \cup \{p\}$
9 $a \leftarrow \pi_q(\mathcal{S})$
(4) Certificate slice.
10 $\sigma \leftarrow L_{\le t}$
11 **repeat for** $i \in \sigma$: **if** steps 2–9 on $\sigma \setminus i$ yield a
12 **then** $\sigma \leftarrow \sigma \setminus i$
until no drop succeeds
13 **return** $a,\ C = \langle \sigma, \rho, a \rangle$

Lines 10–12 are the reference definition of a minimal slice, by reduction. The deployed `certify` reaches the same slice in one pass over L, and `verify` rechecks minimality by leave-one-out replay (Appendix D.6).

Inside the fold, SUPERSEDE replaces a prior value, DELETE tombstones a key so no later instruction revives it, and ACCUMULATE adds to a deduplicated slot.

The refusal ledger ρ carries a key, a reason and an instruction id, never the withheld value.

Algorithm 1: **Recall in TRACE.** Time slice, fold, scope check, reduce.

Definition 2 (Machine state and execution)**.** Execution folds the ledger with a transition operator $\oplus$ into a resolved map $R(t)$ from (entity, attribute) to a value with its admissibility metadata:

$$R(t) \;=\; \bigoplus_{\{\, i_k \,:\, t_e(i_k) \le t \,\}} i_k \;=\; \Big(\big((\varnothing \oplus i_1) \oplus i_2 \big) \oplus \cdots \Big), \tag{1}$$

over the instructions whose event time is at or before t, in transaction-time order. The machine state $\mathcal{S}(t)$ is what survives the scope check on $R(t)$: the entries neither tombstoned nor retracted, whose valid-time bound has not passed at t, and that the requesting principal may see. A query q is answered by a projection $\pi_q(\mathcal{S}(t))$, so $\mathcal{S}(t)$ is all the query may read and π_q only picks a view of it.

A proposition is in force at t when its event time has arrived and its valid-time bound has not passed, so the two ends of the valid interval are both read against t. Counterfactual recall executes a modified copy L' with selected instructions removed or injected, preserving the original ledger.

Specification 1 (Admissibility)**.** Let a proposition p be targeted by a DELETE, be withdrawn by a RETRACT, be past its valid-time bound, or lie outside the query's authorization scope. The read interface exposes only propositions in $\pi_q(\mathcal{S}(t))$; a compliant implementation must therefore never emit p for the corresponding t and scope, regardless of the projection π_q. A value that a later SUPERSEDE has replaced is a separate case: it is not withheld but overwritten, and the instruction that replaced it is named in the certificate slice.

The read interface exposes only the admissible output of execution, so no projection can recover an excluded proposition by scoring it higher. §3 states how this contract is verified and §4.1 evaluates it against a similarity reader over the same ledger.

2.4 QUERY PROJECTIONS AND CERTIFICATES

A query projection π_q reads the admissible state through four views: **Current** (latest value), **Order** (temporal order), **Aggregate** (counts, sums, set-valued answers) and **Relevant** (top-k over admissible propositions). All four share one membership decision, so the readout never changes what is available. Execution also emits a certificate tying the answer to its ledger evidence:

Definition 3 (Recall certificate)**.** For a recall at (t, q), the certificate is $C(t, q) = \langle \sigma,\ \rho,\ a \rangle$ where σ is a minimal slice of the ledger, $a = \pi_q(\mathcal{S}(t))$ is the answer, and ρ is an exclusion report naming every proposition that execution withheld at (t, q): deleted, retracted, past its valid-time bound, or out of scope. Instructions and propositions are identified by memory_id, so each entry of σ and ρ names one version rather than a key, and ρ records the key $(\text{subject}, \text{attribute})$, the reason, and the memory_id of the instruction responsible. Every key named in ρ is by construction absent from $\mathcal{S}(t)$, so the check that a names nothing in ρ is decided at the key level without loss. The report never reproduces governed content, so a deleted secret is accounted for without being disclosed.

Table 1: Four design choices in the five-knob framework.

System	Encoding	Update	Timing	Readout
Dense RAG	embedding	append	*never*	top-k similarity
mem0 / MemoryOS	extracted fact	add/update/delete	write-time	rank / summarize
Zep / Graphiti	typed edges	upsert / invalidate	write-time	graph traversal
StateFuse	typed records	preserve conflicts	query-time	state resolution
MemTX	typed records	transactional commit	write-time	provenance query
TRACE (ours)	**typed instruction**	**state execution**	**query-time**	**project / certify**

Specification 2 (Certificate soundness, minimality, completeness)**.** A certificate $\langle \sigma, \rho, a \rangle$ is *sound* when replaying σ alone yields the same answer a as the full ledger; *minimal* when removing any instruction from σ changes a; and *complete* when a contains no proposition named in ρ and ρ covers every inadmissible proposition relevant to q.

A verifier re-executes σ, inspects ρ, and rejects any certificate failing one of the three checks; the pipeline releases a only on acceptance and otherwise refuses with the report (Appendix D.2).

Recall procedure. Algorithm 1 summarizes recall: execute the selected ledger, record exclusions, project the state to obtain a, and reduce the supporting trace to a minimal σ. Temporal and counterfactual queries reuse the same call with a different time or ledger copy.

Design space. Five choices characterize a memory system: write encoder E, state composition $\oplus$, fold schedule M, readout π, and substrate (Table 1); in TRACE they are the LLM compiler, typed transition operator, query-time execution, state projections and an append-only ledger. The fold schedule decides when updates resolve: a retrieval reader sets $M =$ never and never resolves them, while TRACE keeps the history needed to rebuild any requested state with its evidence.

3 GUARANTEES

Recall runs on an algebra we prove and an interface we specify and machine-check. Propositions 1–2 give the algebra: over the operator semantics of Appendix D.3, $\oplus$ is associative with identity $\varnothing$ and EXECUTE is exactly the fold of the lifted instructions, so any grouping of the selected prefix yields the same state and batched and incremental execution agree. Specifications 1 and 2 bind any compliant implementation, not only ours: an answer drawn from $\mathcal{S}(t)$ alone, and a certificate that is sound, minimal and complete. No reader with $M =$ never can supply the last of the three (Proposition 3), so it separates the paradigms definitionally. That makes the interface testable, and we test it. A symbolic sweep *decides* the equivalence between execution and its specification over every ledger within the bounds of Appendix D.8, with no counterexample, and the deployed `certify` matches exhaustive slicing on all 1000 tested programs with a complete exclusion report on each (Table 12, Appendix D.2).

4 EXPERIMENTS

4.1 FIXED-LEDGER ACCOUNTABILITY

Fixed-ledger tests isolate execution; end-to-end ones add compilation and readout. The former use update splits of AUTOMEMORYBENCH, MemBench, MINTEval, StructMemEval (Wu et al., 2027; Tan et al., 2025; Lee et al., 2026; Shutova et al., 2026) (659 admissibility, 435 certificate queries); the latter LongMemEval and LoCoMo. Every number here is same-ledger, same-judge (§A).

Table 2 runs TRACE and a retrieval proxy over the same ledgers. TRACE answers all 659 queries admissibly and passes every implemented certificate check on all 435 live-target recalls, where the proxy scores 0.000; on admissibility the proxy falls between 0.957 and 0.000 and a dense reader between 0.238 and 0.826 (Table 18). The content is identical

Table 2: **Fixed-ledger accountability.**

Split	n	AdmAcc		Verif.	
		TRACE	Proxy	**TRACE**	Proxy
AMB	224	**1.000**	0.906	–	–
MemBench	23	**1.000**	0.957	**1.000**	0.000
MINTEval	370	**1.000**	0.170	**1.000**	0.000
StructMem	42	**1.000**	0.000	**1.000**	0.000

Table 3: **System results on AUTOMEMORYBENCH, LongMemEval, and StructMemEval.** All AUTOMEMORYBENCH cells are the `audit_subset` split under one scorer (§B.1).

System	AUTOMEMORYBENCH			LongMemEval	StructMemEval
	StrictCore%	Task%	AMQ%		
TRACE and retrieval reference					
TRACE (ours)	**4.0**	**4.0**	**66.6**	**0.782**	**0.592**
Dense-RAG (bge-m3)	-	-	-	0.446	-
Managed platforms					
Supermemory (GPT-5, top-200)	0.0	0.0	25.0	0.934	0.000
Maximem (Synap SaaS)	0.0	0.1	27.2	0.920	-
Oracle Agent Memory	-	-	-	0.938	-
Mi-Memory / MemStack	-	-	-	0.875	-
Memory framework systems					
mem0	0.0	0.2	25.4	0.944	0.204
EverMind	0.1	1.0	29.0	0.830	0.000
MemOS	0.0	1.2	31.0	0.778	-
Letta / MemGPT	0.1	0.3	30.8	-	0.204
A-Mem	0.0	0.3	30.4	-	0.204
LangMem	0.1	0.5	22.1	-	0.000
LightRAG	0.0	0.0	37.5	-	-
Retrieval / lightweight baselines					
OpenAI-Memory	-	-	-	0.502	0.306
Zep / Graphiti	0.0	0.4	35.6	0.158	0.265
MemInsight	0.0	1.0	20.8	0.582	0.245
MemoryOS	0.0	0.0	23.4	0.086	0.449
SimpleMem	0.0	0.6	37.4	0.070	0.204
txtai	0.0	0.0	34.7	0.056	0.204
MemEngine	0.0	0.0	30.5	-	0.204
Memoripy	0.0	0.6	35.0	-	0.041
Non-memory agent controls					
Aider	5.5	11.4	35.3	-	-
OpenCode	2.9	6.3	33.3	-	-
Claude Code	8.6	17.8	36.8	-	-
Codex	6.4	10.3	37.7	-	-

across readers, so the difference is in the reading, and a matched control says which part of the reading: a store folding the identical operations into present state at write time reproduces our admissibility exactly, yet holding present state alone it answers time-indexed queries at the latest-value ceiling of 0.333 where EXECUTE(L, t) answers at 1.000. Executing recorded updates is what beats retrieval, and keeping the ledger is what answers as-of queries, with no snapshot schedule or versioning index (Appendix E.1).

Why the gap is structural. A ranking reader returns the evidence it scored highly and never the propositions it withheld, so it has nothing to put in ρ and nothing for a completeness check to read: it scores zero on the joint certificate criterion, and no tuning changes it (Appendix E.1). Execution produces both halves from the same fold: the admissible state answers the query and the discarded entries account for what was excluded.

Table 4: **StrictCore separates systems only where a lifecycle runs** (§B.1).

System	Rec.	Strict	AMQ
Reports memory operations			
TRACE (ours)	**58.9**	**4.0**	**66.6**
mem0	27.8	0.0	25.4
MemInsight	20.3	0.0	20.8
LangMem	18.9	0.1	22.1
Reports none (anchors)			
ReMe	0.0	12.2	38.8
LightRAG	0.0	0.0	37.5
SimpleMem	0.0	0.0	37.4
Zep-cloud	0.0	0.0	37.5

4.2 SYSTEM COMPARISONS

Table 3 compares memory systems and agent controls on StrictCore, the joint criterion scored binary per case, and AMQ, the benchmark's seven-axis lifecycle composite (Table 6). On AUTOMEMORYBENCH TRACE

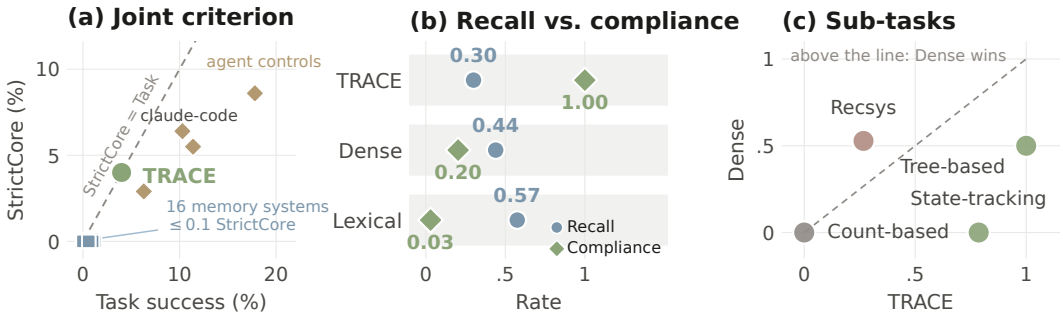

Figure 3: **System comparison.** (a) Joint criterion on AUTOMEMORYBENCH. (b) Recall against compliance, each reader at its own budget (Table 9 matches them). (c) Subtasks vs. dense-RAG.

reaches 4.0% StrictCore (task success, safety and temporal validity at once) against at most 0.1% for every vendor memory system with full coverage of the memory lifecycle, a margin of forty times. Table 4 regroups the runs by whether a system reports any memory operation at all, which is what makes the joint criterion readable; the systems that report none are anchors rather than rankings, their memory layer being unobservable in the trace (Appendix B.1). Among the systems that do report, the separable axes open a wider gap than the joint criterion alone: native recall 58.9 against a best of 27.8, safety 100.0 with zero forbidden activations, and AMQ 66.6 against 38.8 for the best vendor in either group. The agent controls keep no memory layer and re-read the whole transcript each turn, so they bound full-context access; against them what separates TRACE is conversion, not coverage (Appendix B.1): its StrictCore equals its Task success ($4.0/4.0$), every task it solves is also safe and temporally valid, where the controls convert 0.46–0.62 of theirs (Figure 3a).

Figure 3 puts the three comparisons together: TRACE leads every memory system on the joint criterion and on update-heavy subtasks, dense-RAG on Recsys-style queries. Panel (b) reports each reader at its own budget, where TRACE holds compliance at 1.000 against 0.030 and 0.203 on a third as many items. Equalizing it removes the caveat, not the result: at $k = 3$ and $k = 10$ TRACE leads on both, reaching 1.000 recall at 1.000 compliance at $k = 10$ against lexical retrieval's 0.575 at 0.030 (Appendix C.3).

Table 5 reports the MemBench R-First comparison, and on StructMemEval ($n = 49$) TRACE scores 0.592 against 0.449 for MemoryOS and 0.306 for OpenAI-Memory under the same answerer and judge. Two blocks match end to end: the AUTOMEMORYBENCH columns of Table 3, which TRACE leads among reporting systems, and the MemBench R-First block, where it sits within 0.017 of the best reader while being the only one that also reports what it withheld. Appendices B.3, A, E.2 and B carry the subtask results, protocols and blank cells of Table 3.

Table 5: MemBench R-First accuracy ($n = 1500$).

System	R-First
TRACE (ours)	**0.912**
SimpleMem	0.929
Zep / Graphiti	0.929
OpenAI-Memory	0.925
MemoryOS	0.907
Dense-RAG (bge-m3)	0.317

4.3 TEMPORAL RECALL, GOVERNANCE, AND COMPONENT ABLATION

Figure 4 shows higher temporal accuracy for the recorded TRACE pipeline and zero observed leakage on the deletion and adversarial tests; governance holds on the compiled ledger, and the narrative-fallback path is measured and closed, redaction keeping the governed value out of the evidence on every adversarial query at no cost in recall (Appendix E.4). Figure 5 then locates that behavior: over six variants on 104 cases and 2184 queries, removing supersession or governance sharply reduces safety while barely moving task success, and removing the whole State-VM reduces all three, which separates task accuracy from the safety properties. Appendices A, C.3, D.9 and E.3 give the settings, the matched-budget analysis, the attribution and the compiler configuration.

5 ANALYSIS

Baseline comparisons. Same-ledger experiments hold the stored content fixed and vary how it is read, isolating the benefit of resolving updates. A zero certificate score means the baseline supplies no exclusion report, so completeness cannot be met; where a deployed system implements update or deletion semantics, we take its behavior from its own run, not a proxy (§6).

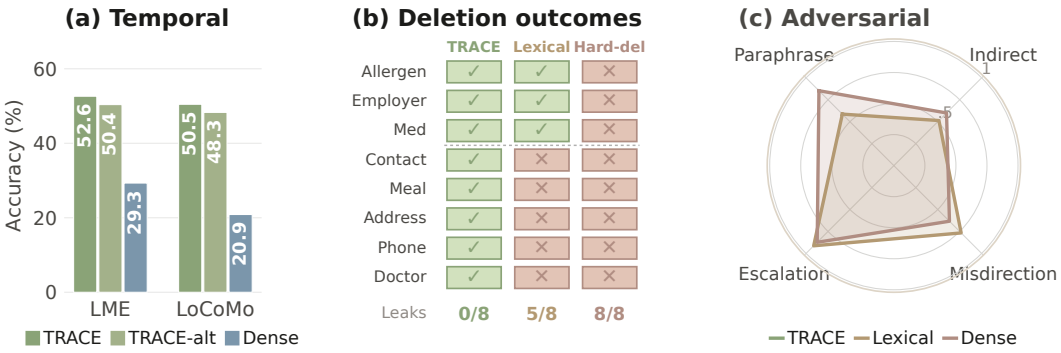

Figure 4: **Temporal recall and governance.** (a) Temporal accuracy. (b) Leakage after deletion over 500 synthetic trials. (c) Leakage under 600 attacks. Lower is better in (b) and (c).

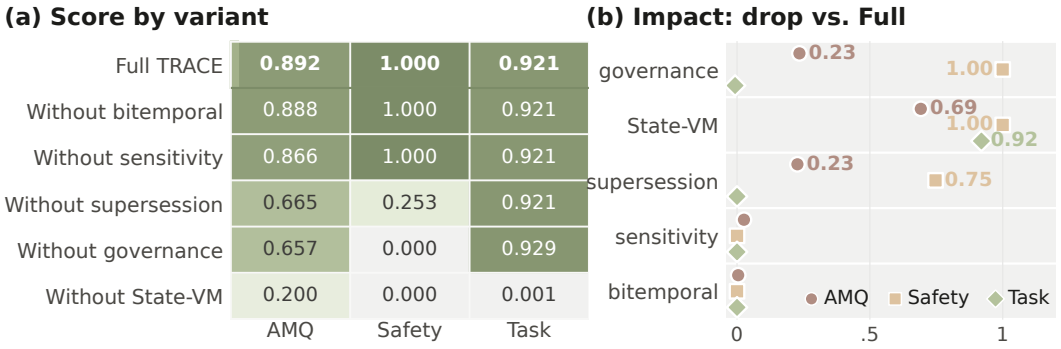

Figure 5: **State-VM component ablation.** Cell color encodes the score (sequential green); removing supersession or governance drops Safety toward 0 while Task success stays high.

Certificate validity. The deployed slicer is compared with exhaustive subsequence search on 500 programs in two regimes: live-only targets, and an any-target stress regime whose query may name a tombstoned or superseded target. Soundness, deletion-minimality, slice-index agreement and slice size hold at $500/500$ in both, so the heuristic equals the exhaustive minimum on all 1000 programs (Table 12), and on none does the verifier find an inadmissible proposition the report missed.

What separates the two regimes is the queries, not the certificates. The stress regime samples answer targets from every proposition in the ledger, so $393/500$ of its queries ask about one execution has excluded; the certificate reports each as excluded and the fail-close policy of Appendix D.2 refuses the answer, the specified behavior for a question about a deleted memory, so TRACE answers 21.4% of the stress queries and refuses the rest, against 100% live-only (Appendix D.4). Reaching agreement took one correction to the slicer, which had selected the last instruction acting on a target without checking for an earlier DELETE (Appendix D.6).

What certification costs. Constructing and verifying a certificate costs $\approx 4\times$ a plain EXECUTE from $|L| = 100$ to the 5000 we measured, reaching 22 ms there. Both stages are one pass over the ledger plus work in $|\sigma|$, which is why the factor holds across that range; it is stable, not proved constant (Appendix D.5). The certified slice does not grow with the ledger (Figure 6a), so replaying the support behind an answer stays a fixed-size read.

Task-level differences. The subtask breakdown localizes the gains to the regimes execution targets: knowledge updates and temporal reasoning improve, while tasks without update semantics move by at most -0.033 on subsets small enough that a few items move the rate (Appendix B.3). Oracle headroom remains on both, so the gains are not a saturation artifact.

Execution cost. The deterministic execution kernel makes no LLM calls, and Figure 6(b) measures it alone: certification adds the factor above, while compilation, verbalization and relevance ranking are backbone LLM costs, shared by every system compared, that dominate end-to-end latency. Incremental maintenance can reduce repeated replay for a fixed query, though historical queries, changed scopes and counterfactual branches may need reconstruction. Eager and lazy schedules match (Appendix D.9), so the accuracy comes from explicit update resolution, not the schedule.

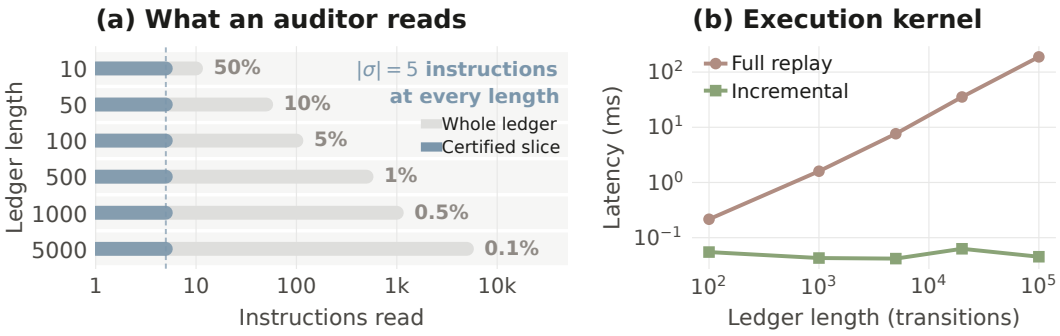

Figure 6: **What a certificate costs.** (a) The slice holds at 5 instructions as the ledger grows $500\times$: half a 10-instruction history, a thousandth of a 5000-instruction one. (b) Kernel replay latency.

6 Related Work

Agent memory. Systems differ in how memory is written and ranked: experience retrieval (Park et al., 2023), hierarchies (Packer et al., 2023; Kang et al., 2025), fact consolidation (Chhikara et al., 2025; Fang et al., 2025), linked or annotated records (Xu et al., 2025; Salama et al., 2025), forgetting (Zhong et al., 2023), learned retrieval (Wang et al., 2023), heterogeneous stores (Li et al., 2025), and graph or temporal structure (Guo et al., 2024; Rasmussen et al., 2025). LazyMem (Yu et al., 2026) defers construction to query time but still hands the reader text to rank. TRACE changes what recall returns: typed updates executed into the admissible state before ranking.

State semantics and provenance. Provenance explains output dependencies (Buneman et al., 2001; Green et al., 2007), while StateFuse (Volkov et al., 2026) and MemTX (Li et al., 2026) resolve conflicts and validate belief updates over immutable histories. Each serves a store or a transaction manager; a versioned store or snapshot index can also answer a question about an earlier state. What TRACE adds is the packaging: one recall call carries the admissibility semantics, the time and scope selection, and the evidence, so the answer arrives with the instructions that support it and the reason for every relevant exclusion, and a verifier can reject it on any of the three. Unlike textual citation evaluation (Gao et al., 2023), the certificate covers the structured answer.

Evaluation. Benchmarks test conversational recall (Wu et al., 2024; Maharana et al., 2024), interference (Lee et al., 2026), memory across interactions and tasks (Hu et al., 2025; He et al., 2026), and access, forgetting and invalidation (Ren et al., 2026; Uddin et al., 2026). We separate recall quality from admissibility and certificate validity, and fixed-ledger tests from end-to-end compilation.

7 Conclusion

Recall-as-Execution records each update as a typed instruction and answers a query by executing that ledger into the state time and authorization allow, separating what may be used from what is relevant; TRACE realizes it end to end, and three things follow. The recall interface becomes *specifiable*: an admissibility and certificate contract binds any implementation, the algebra beneath it is proved, and the executor is machine-checked against it. Justification becomes *free*: the fold that produces the answer also produces its supporting instructions and the reason for every exclusion. Accountability becomes *measurable* on its own axis: on fixed ledgers TRACE holds 100% admissibility over 659 queries, with no leak in 600 adversarial or 500 deletion trials and a certificate on every evaluated recall, against 0.000 for readers with none; ablating supersession or governance removes the safety behavior and leaves task accuracy untouched, the pattern a structural explanation predicts and a backbone-quality one does not. What the paradigm buys beyond any single number is that the hard cases stop being special: a question about last March, a question after a deletion request, and a question asked by a different principal are one call with a different time, ledger or scope, answered by the same executor and covered by the same certificate. Ranking leaves the membership decision implicit in a score and cannot be asked to defend it; execution has to state that decision, and can certify it. That is the shift we argue for: a memory that is run rather than searched, and audited.

REPRODUCIBILITY STATEMENT

Every number in this paper comes from a recorded run; no result is estimated or projected. Appendix A gives the evaluation protocol for each benchmark: the split, the query count, the compiler and verbalizer models, and the judge. Appendix B states, cell by cell, which entries in Table 3 come from our own runs and which the vendors report under a different protocol, so the table is not read as a matched ranking. Appendix C describes the construction of every synthetic ledger used in the controlled experiments, including how each control is deliberately deprived of the capability under test. Appendix D.8 gives the bounds of the symbolic check ($N \leq 20$, $M \leq 10$) and the random-triple settings and seed behind Table 13. Every experiment the paper describes has been run and reports a number; Appendix C.3 additionally reports the matched-budget re-run, and Appendix D.5 the certificate cost. Appendix D.4 states the certificate coverage and refusal counts and the two quantities it does not measure; Appendix D.5 reports certificate construction and verification cost against execution; and Appendix B.1 states what the AUTOMEMORYBENCH gap can and cannot be ascribed to, cell by cell.

ETHICS STATEMENT

This work studies deletion, expiry, and authorization in agent memory, so the experiments involve records that stand in for personal data: addresses, phone numbers, medications, and similar attributes. All such records are synthetic; we collected no data from human subjects, and no real personal data appears in any ledger, benchmark, or released artifact. We use the conversational benchmarks (LongMemEval, LoCoMo, MemBench, MINTEval, StructMemEval) as published.

The paper reports an adversarial leakage evaluation. We describe attack categories at the level needed to interpret the results and do not release attack strings targeting any deployed system. The dual-use risk is limited: the attacks probe whether a memory system returns content it was asked to forget, which is a property a deployer should be testing for.

We want to be explicit about one limitation with ethical weight. Exclusion from the recall interface is not physical erasure: the ledger is append-only, so a deleted value remains in storage and remains readable by anything that bypasses EXECUTE. A deployment that must satisfy a legal erasure request therefore needs a separate compaction mechanism, and this paper does not provide one (Appendix G). A second gap of the same kind, the narrative fallback re-surfacing a value the governance gate had withheld, we found by testing for it, measured at 1.000 leakage, and closed; Appendix E.4 reports both the measurement and the fix.

USE OF LARGE LANGUAGE MODELS

Large language models were used only as a writing aid, for drafting and editing prose. They contributed no claim, method, or result, and no number in this paper was produced or altered by a language model. The authors are responsible for all content.

REFERENCES

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001*, pp. 316–330, 2001. doi: 10.1007/3-540-44503-X_20. URL https://www.pure.ed.ac.uk/ws/files/16509989/Why_and_Where_A_Characterization_of_Data_Provenance.pdf.

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

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. Lightmem: Lightweight and efficient memory-augmented generation. *arXiv preprint arXiv:2510.18866*, 2025.

Tianyu Gao, Howard Yen, Jiatong Yu, and Danqi Chen. Enabling large language models to generate text with citations. *arXiv preprint arXiv:2305.14627*, 2023.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 31–40, 2007. doi: 10.1145/1265530.1265535. URL https://www.cs.ucdavis.edu/~green/papers/pods07.pdf.

Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779*, 2024.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex Pentland. MemoryArena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. *arXiv preprint arXiv:2507.05257*, 2025.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory os of ai agent. *arXiv preprint arXiv:2506.06326*, 2025.

Hyunji Lee, Justin Chih-Yao Chen, Joykirat Singh, Zaid Khan, Elias Stengel-Eskin, and Mohit Bansal. Minteval: Evaluating memory under multi-target interference in long-horizon agent systems. *arXiv preprint arXiv:2605.18565*, 2026.

Xiaoyang Li, Yiqi Wang, Haohui Lu, Zhi Chen, Mo Li, Pingan Song, Mingkai Zheng, and Taotao Cai. Memtx: Transactional belief commit for stateful agent memory. *arXiv preprint arXiv:2607.23929*, 2026.

Zhiyu Li, Shichao Song, Chenyang Xi, Hanyu Wang, Chen Tang, Simin Niu, Ding Chen, Jiawei Yang, Chunyu Li, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Huayi Lai, Hao Wu, Bo Tang, Zhenren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. MemOS: A memory OS for AI system. *arXiv preprint arXiv:2507.03724*, 2025.

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.

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.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative Agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

Zhe Ren, Yibo Yang, Yimeng Chen, Zijun Zhao, Benshuo Fu, Zhihao Shu, Bingjie Zhang, Yangyang Xu, Dandan Guo, and Shuicheng Yan. Gatemem: Benchmarking memory governance in multi-principal shared-memory agents. *arXiv preprint arXiv:2606.18829*, 2026.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. Meminsight: Autonomous memory augmentation for llm agents. *arXiv preprint arXiv:2503.21760*, 2025.

Alina Shutova, Alexandra Olenina, Ivan Vinogradov, and Anton Sinitsin. Evaluating memory structure in LLM agents. *arXiv preprint arXiv:2602.11243*, 2026.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. Membench: Towards more comprehensive evaluation on the memory of llm-based agents. *arXiv preprint arXiv:2506.21605*, 2025.

Md Nayem Uddin, Kumar Shubham, Eduardo Blanco, Chitta Baral, and Gengyu Wang. From recall to forgetting: Benchmarking long-term memory for personalized agents. *arXiv preprint arXiv:2604.20006*, 2026.

Sergey Volkov, Yang Li, and Ye Luo. Statefuse: Deterministic conflict-preserving memory for multi-agent systems. *arXiv preprint arXiv:2607.05844*, 2026.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting language models with long-term memory. *arXiv preprint arXiv:2306.07174*, 2023.

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

Jiajun Wu, Jian Yang, Chen Li, Linzheng Chai, Ge Gao, Ensheng Shi, and Yuchi Ma. AutoMemoryBench: State-contract evaluation for auditable agent memory, 2027. URL `https://github.com/wuyuVerse/AutoMemoryBench`.

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*, 2025.

Jing Yu, Yibo Zhao, Jiaming Zhang, and Xiang Li. LazyMem: Retrieve broadly, construct selectively for efficient long-term agent memory. *arXiv preprint arXiv:2607.22690*, 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.

The appendix is organized so that the flow mirrors the main paper. §A states the evaluation protocol that underlies every table and figure. §B then explains how to read Table 3: which cells are absent and why, and how the external splits decompose by subtask. §C reports the controlled state-resolution experiments that support the paradigm-level claims but do not appear in the main text. §D covers the formal-layer coverage, component attribution, and update-scheduling studies that back §3; within it, §D.3 states which operator interactions the algebra reaches and which time stamp does which job, §D.4 gives the certificate coverage, refusal accounting, and directed rejection checks, and §D.7 reconciles the two admissibility metrics. §E reports the reader-level, compiler-budget, and proposition-loss diagnostics that scope the deployed pipeline. §F covers execution cost and the qualitative design-space mapping, and §G records the scope of each measurement.

A Evaluation Protocol Details

This section fixes the evaluation vocabulary used throughout the rest of the appendix: what admissibility, compliance, and accountability mean; which splits appear in which table; and how vendor-reported figures enter the comparisons.

A.1 Metrics and evaluation slices

Engine correctness uses deterministic scoring on a fixed ledger; end-to-end evaluation lets an LLM compiler build it from raw text. The two protocols measure different error sources. Admissibility excludes deleted, expired, superseded, or unauthorized propositions from the resolved state, and AdmAcc requires both an admissible answer and the correct current value. Compliance is the binary readout metric: the returned set holds no forbidden items. Accountability combines admissibility with verifiability, measured by the implemented soundness, minimality, and completeness tests.

The fixed-ledger splits are AutoMemoryBench contract queries (Wu et al., 2027), MemBench knowledge updates (Tan et al., 2025), MINTEval state tracking (Lee et al., 2026), and Struct-MemEval residence-state queries (Shutova et al., 2026). Each is the public release of that benchmark; we slice it rather than regenerate it. The per-split counts in Table 2 are not whole-benchmark sample sizes. Certificate checks cover the three external splits; the AutoMemoryBench split contributes only to AdmAcc here. The main benchmark table reports AutoMemoryBench scores as percentages and external-task scores as fractions.

Definitions and scoring units. Table 6 fixes each metric, its unit of account, and how it aggregates. All rates are micro-averaged over the scoring unit named there, with no per-case reweighting.

Table 6: **Metric definitions.** Unit is what one score is computed over; all rates are micro-averages over that unit.

Metric	Unit	Definition
AdmAcc	query	Answer is drawn from $\mathcal{S}(t)$ *and* carries the correct current value. Both conditions required.
Misuse	forbidden kind × query	Any forbidden value appears anywhere in what the reader returns (§D.7).
Compliance	query	Returned set contains no forbidden item. Binary.
Recall	query	Distinct currently-valid propositions returned, over the number in force.
Native recall	case	Propositions the system's own memory layer surfaces, over those the contract requires.
Safety	case	No forbidden activation anywhere in the trajectory.
Task	case	The contract's task assertions all pass.
StrictCore	case	Task ∧ Safety ∧ temporal validity, on the same case.
AMQ	case	Weighted mean of seven lifecycle axes: write 0.20, task utility 0.20, retrieval 0.15, update 0.15, compression 0.15, safety 0.10, efficiency 0.05. An axis a run does not expose is dropped and the remaining weights are renormalized, so AMQ is scored over the axes a system reports.
Write F1	proposition	Compiled instructions against the contract's expected instructions.
Cert. checks	recall	Soundness, minimality and completeness (Specification 2), each a separate rate.

A worked case: a AUTOMEMORYBENCH case with 4 task assertions, 2 forbidden kinds and 1 temporal assertion scores Task $= 1$ only if all 4 assertions pass, Safety $= 1$ only if neither forbidden kind activates, and StrictCore $= 1$ only if Task, Safety and the temporal assertion all hold. One failed task assertion sets Task $= 0$ and therefore StrictCore $= 0$, while Safety can still be 1. A run that reports no memory operation at all is scored on the same task assertions, but its governance axes are recorded as unobservable rather than as passes (§B.1), which is why those rows are read as anchors. AdmAcc at 1.000 says every answer was drawn from the admissible state with the right value; it does not say that nothing unsafe appeared anywhere in a longer generation, which is what Misuse measures separately.

A.2 COMPARISON RECORDS AND ADAPTERS

Same-ledger controls isolate the reader. Deployed-system runs compare complete pipelines, while vendor-reported scores provide context under different settings. Models, judges, slices, and retrieval budgets must match before a difference can be attributed to a system change.

The TRACE LongMemEval score 0.782 comes from the 500-question run with Flash compilation, GPT-5.5 answer generation and a DeepSeek-V4-Pro judge. Its matched control is dense-RAG at 0.660 on the same 500 questions: the conversation is chunked, the top 20 chunks by bge-m3 cosine are fed verbatim, and the same GPT-5.5 answerer and DeepSeek-V4-Pro judge score the result, so the two runs differ only in what reaches the answerer. That is the matched pair we report on this benchmark; the remaining LongMemEval cells in Table 3 come from other judges or from vendor reports and are listed as such below. The dense-RAG 0.446 entry in Table 3 is the same retrieval budget under a GPT-4o judge with a Flash answerer, which is why it differs from the 0.660 above; mem0 0.944 is vendor-reported. These entries do not form a matched ranking, and we draw no ordering from them. The StructMemEval column is the $n = 49$ shard, run through the same fleet runner, answerer, and judge as every competitor cell, so it is matched; Appendix B.3 reports the expanded four-subtask decomposition, which uses a different pool and is not comparable cell-for-cell with it. The TRACE score of 0.592 there is the 3-run majority of 0.551, 0.571 and 0.612, while the baselines are single runs, so TRACE carries a majority-vote budget the others do not; its lowest single run, 0.551, still leads both. The two end-to-end matched blocks are the AUTOMEMORYBENCH columns of Table 3, where every system runs the same `audit_subset` split through one scorer with TRACE in its no-gold configuration, and the MemBench R-First block of Table 5.

What identifies a result. Every number in this paper is produced by a configuration with eight fields, and we state them wherever a comparison turns on them: the data slice and its size, the compiler backbone, the answerer, the judge, the read budget, the random seed, the memory SDK version, and the scorer revision. Two numbers are comparable only when all eight match, which is the rule we use to decide whether a pair of cells may be ranked. The fixed-ledger results carry no compiler, answerer or judge, because they score EXECUTE directly on a stored ledger under a deterministic scorer; those runs are identified by slice, seed and scorer revision alone. Results scored before the tokenizer correction of §D.7 are marked where they appear, and the cells regenerated after it read from the corrected artifact named there.

Certificate checks and benchmark memory-ID credit are distinct. A statement-level certificate does not automatically supply the `activated_memory_ids` required by a benchmark adapter. Missing ID emission is an integration limitation, not a property implied by execution.

A.3 TEMPORAL AND GOVERNANCE TEST SETTINGS

Figure 4a uses temporal subsets of LongMemEval ($n = 133$) and LoCoMo ($n = 321$), with a GPT-4o judge. TRACE uses Flash compilation and GPT-5.5 verbalization. The dense answerer is Flash on LongMemEval and GPT-5.5 on LoCoMo, so the comparison does not isolate memory architecture from answer generation. A separate all-GPT-5.5 LoCoMo run scores 0.483.

The deletion test in Figure 4b comprises 500 synthetic trials. Its mem0-style hard-delete simulation removes the memory row but retains an audit body containing the deleted value. This is not a deployed mem0 SDK measurement. The 500 trials are 100 deletion cases times 5 paraphrases of the same query, one deleted attribute per case, under a fixed seed. The cases draw that attribute from eight categories (allergen, employer, medication, emergency contact, favorite meal, home address,

phone number, primary doctor), 60–65 trials each, and the panel summarizes the run by those categories: a cell marks whether the reader leaked on that attribute at all, and the $x/8$ figure counts categories with at least one leak. It agrees with the trial-level rates, $5/8$ against a measured lexical leak rate of 0.63 and $8/8$ against the hard-delete reader's 1.000, while TRACE leaks in no category and on no trial. Besides checking leakage, the test requires a no-record sentinel: TRACE passes all trials and both controls fail that check.

The separate 600-attack test in Figure 4c combines four attack types, three strengths, and 50 base cases. Both retrieval controls use top-8. It probes query phrasing against a fixed compiled ledger, not whether an LLM assigns correct sensitivity labels. The different-budget recall/compliance study is reported separately in Appendix C.3, without repeating its table here.

B MAIN-TABLE NOTES

How to read Table 3: §B.2 lists the four causes of blank cells and §B.3 decomposes the external-benchmark averages, so readers can see where the aggregate gains and losses come from.

B.1 WHAT THE AUTOMEMORYBENCH GAP CAN AND CANNOT BE ATTRIBUTED TO

Table 3 shows TRACE at 4.0% StrictCore against $\leq 0.1\%$ for every full-coverage vendor memory system that exercises the memory lifecycle. The same systems score 0.0–1.2% on Task success alone. Because StrictCore is a conjunction that includes Task, a system failing Task fails StrictCore for reasons that need not involve memory at all, and the table does not say which cause is operating.

We can say what the numbers were produced by. Every vendor cell comes from that system's own SDK driven through a common contract harness, with the same suite, the same task prompts, and the same scorer; no vendor cell is a proxy or a reported figure. The TRACE cell is measured on the same `audit_subset` split, all eight domains and $15{,}120$ queries, scored by the same `Scorer` and the same StrictCore definition, with the solver in its honest configuration: it reads none of the gold annotation the scorer uses, only the per-memory contract fields the sandbox exposes to every system. What we cannot say is how each system's failures decompose. The decomposition would need, per system, a count of runs that failed to produce parseable tool calls, failed to write to memory at all, wrote but retrieved nothing, and retrieved an inadmissible value. Only the last is a memory-semantics failure in the sense the paper argues about. We have not produced that decomposition and do not report one.

A system that scores higher on StrictCore. One full-coverage system we ran, ReMe, reaches 12.2% StrictCore, above our 4.0%, and we report that rather than omit it. It is not a counterexample to the argument, for a reason visible in what its run does and does not record: ReMe reports a memory operation on 0% of queries, so every safety and temporal-validity conjunct it satisfies is satisfied without any observable memory operation behind it. We are careful about what that zero licenses. It means no activation trace was reported, which we cannot separate from an adapter that retrieves but does not expose a trace; it is not a measurement that ReMe never touched the lifecycle. Either reading leaves its StrictCore unusable as evidence of lifecycle governance, which is the only thing we need here: it is a task-accuracy result wearing a governance conjunction. On the diagnostics that do not depend on trace reporting it sits at 38.8 AMQ against our 66.6, and its native recall is likewise uncreditable at 0.0 against our 58.9. The comparison that carries our claim is against systems that do exercise the lifecycle and are scored on it, where the margin is forty-fold. We flag ReMe, LightRAG, SimpleMem, and Zep-cloud as anchors on this ground: their task-level numbers stand, but their governance behavior is not measured, which is why ReMe's 12.2 StrictCore is read this way rather than dropped. Table 4 is also where the ReMe and hosted Zep-cloud runs appear; the latter is a separate deployment from the self-hosted Zep/Graphiti row of Table 3. The conversion rates in Figure 3a are each computed on that system's own solved set, and those sets differ, so the panel compares how reliably a system converts what it already solves rather than its safety at matched difficulty.

Adapter-level faults are quiet here: a slow embedding backend, an SDK whose write path drops a field, or an unparseable output format each produce a clean zero indistinguishable from a capability result. We therefore scope the claim to what survives that uncertainty: among the memory systems

we could run end to end, none reporting a memory operation on any query satisfied the joint task–safety–temporal criterion on more than 0.1% of the suite. The attributable comparison is the fixed-ledger one in §4.1, which removes the harness and compiler and holds the stored content identical across readers. That is where the memory-semantics argument rests.

B.2 MISSING-CELL REASONS

Missing cells in Table 3 fall into four categories: (1) **No SDK to onboard.** Oracle, Mi-Memory / MemStack, Maximem SaaS. (2) **SDK too slow to finish in the pod budget.** letta, A-Mem, Lang-Mem, LightRAG, MemEngine, Memoripy. (3) **Vendor score on a different caliber.** Reported in Appendix B.3. (4) **Non-memory agent controls.** Claude Code, Codex, Aider, and OpenCode are not run on the external memory benchmarks.

B.3 PER-SUBTASK BREAKDOWN ON EXTERNAL BENCHES

The full StructMemEval and MemBench evaluations contain heterogeneous subtasks. Table 7 gives their breakdown; the MemBench R-First row is reported separately in Table 5. On StructMemEval, TRACE takes state-tracking at 0.786 against dense-RAG's 0.000 and tree-based at 1.000 against 0.500, but drops to 0.267 on the recsys pool (0.528 for dense), which carries 79% of the benchmark's weight and pulls the micro-average to 0.345 vs. dense's 0.459. This decomposition runs the expanded four-subtask pool ($n = 229$), a different and much recsys-heavier slice than the $n = 49$ set in Table 3, which is why the two micro-averages are not comparable to each other. Both decompositions in this subsection come from the run whose aggregates §E.3 records as the earlier pair, 0.719 and 0.345; the per-subtask split was not recomputed for the budget-4096 rerun reported there. The -0.114 micro-average is the cost of the design on a pool that is dominated by similarity-style recommendation queries, where there is no update to resolve; the state-tracking and tree-based splits, where there is, move the other way by $+0.786$ and $+0.500$. On MemBench, TRACE scores 0.912 on R-First, fourth of six in Table 5 behind SimpleMem and Zep at 0.929 and OpenAI-Memory at 0.925. On the two state-heavy subtasks that require resolving an update it scores R-Third 0.891 and F-Third 0.806; no baseline publishes those cells under this protocol, so these are absolute numbers and not a ranking. It loses on F-First recall (0.476), dragging the overall micro to 0.719 against 0.766. Read together, the two decompositions are evidence that update-heavy and retrieval-heavy tasks can behave differently within the same evaluation. They do not imply that the gains are uniform across every subtask of every benchmark. On LongMemEval the two subtasks without update semantics move by -0.014 (single-session-user, $n = 70$) and -0.033 (preferences, $n = 30$); each gap is a single item on those subsets, so we read them as unchanged.

Table 7: **Per-subtask task accuracy.** Results vary across lifecycle, retrieval, and counting tasks; missing baseline cells are unreported.

Bench / subtask	n	TRACE	Dense-RAG	Interpretation
StructMemEval (n=229 full)				
state-tracking	14	**0.786**	0.000	lifecycle heavy: TRACE dominates
tree-based	20	**1.000**	0.500	lifecycle heavy: TRACE dominates
recsys	180	0.267	**0.528**	pure retrieval (RAG regime, 79% weight)
count-based	15	0.000	0.000	both fail (see counting predicate set for isolated study)
overall micro	229	0.345	**0.459**	*dominated by recsys weight*
MemBench (n=9955 full)				
R-Third (recall / 3rd-person)	1500	**0.891**	–	state-heavy
F-Third (bitemporal / 3rd)	3455	**0.806**	–	TRACE main territory (double-time)
F-First (bitemporal / 1st)	3500	0.476	–	recall bottleneck in compile
overall micro	9955	0.719	**0.766**	*F-First drags overall*

C Additional Lifecycle Experiments

The tests in this section isolate specific state-resolution behaviors on constructed ledgers. They complement the aggregate deletion and adversarial results in Figure 4 with three narrower probes: historical, counterfactual and counting queries (§C.1), sensitive-value handling under a long event history (§C.2), and the recall–compliance tradeoff under different return budgets (§C.3). These are not evaluations of deployed vendor systems or of natural-language compilation; the deletion and 600-attack tests in Figure 4 are the only ledger-compiled measurements.

How to read these controls. Each control is constructed to isolate one capability: the latest-value control is time-agnostic, the counterfactual control executes the unmodified ledger, and the scalar control keeps one value per attribute. Each therefore measures whether that capability is necessary for the task and whether TRACE supplies it, which is what §C.1–§C.2 report. A retrieval system engineered with the same capability (a time-indexed retriever, a retriever re-run over an edited corpus, a multi-valued store) would also clear these probes; the comparisons against deployed baselines are Table 3 and Figure 4.

C.1 Historical state, counterfactuals, and counting

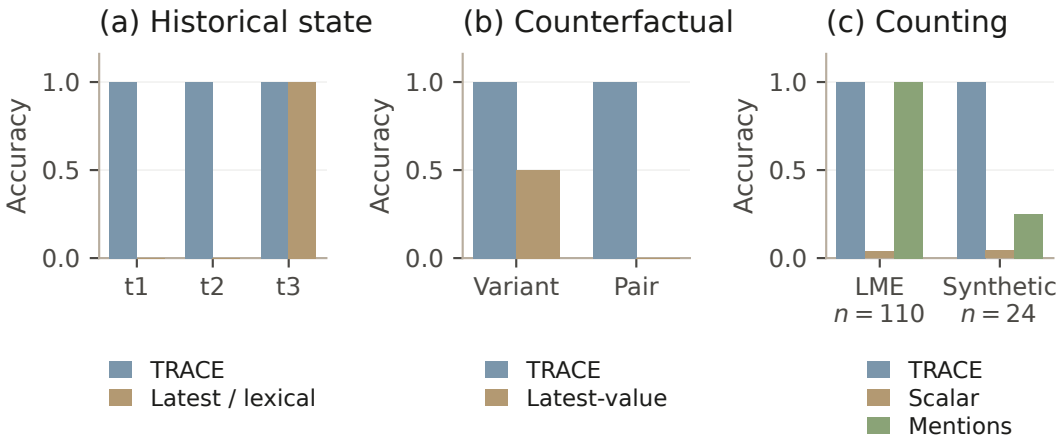

Figure 7: **Controlled state-resolution tests.** (a) Historical queries. (b) Counterfactual branches. (c) Counting. All use constructed ledgers.

Historical queries. The test uses 100 synthetic cases at three query times; the latest-value and mem0-style lexical controls coincide. A fixed latest-value answer succeeds only at the last of the three query times in this construction. The time-aware reader resolves each historical state (Figure 7a). This is a limitation of the time-agnostic controls, not of every retrieval implementation.

Counterfactual branches. The test contains 100 synthetic groups; pair success requires every variant in a group to be correct. The control executes the unmodified ledger for every variant, whereas TRACE replays the counterfactual ledger. Figure 7b separates correctness on individual variants from correctness on an entire group. The tested families remove one or more instructions, remove and inject instructions, or apply multistep changes.

Counting under governance. The test covers 110 LongMemEval-derived questions and 24 synthetic cases. The LongMemEval-derived ledgers are synthesized to represent what a correct compiler would produce; they do not measure real compilation accuracy. The scalar control keeps at most one value per attribute. The mention-scan control has oracle predicate hints but does not resolve supersession, retraction, deletion, or expiry. Figure 7c separates the derived questions from the synthetic operator tests.

C.2 Sensitive values and long event histories

The sensitive-value test uses 300 queries over 100 cases, with direct, lexical-similar, and semantic-similar requests. The keyword filter is a mem0-style simulation, not a deployed SDK result. The

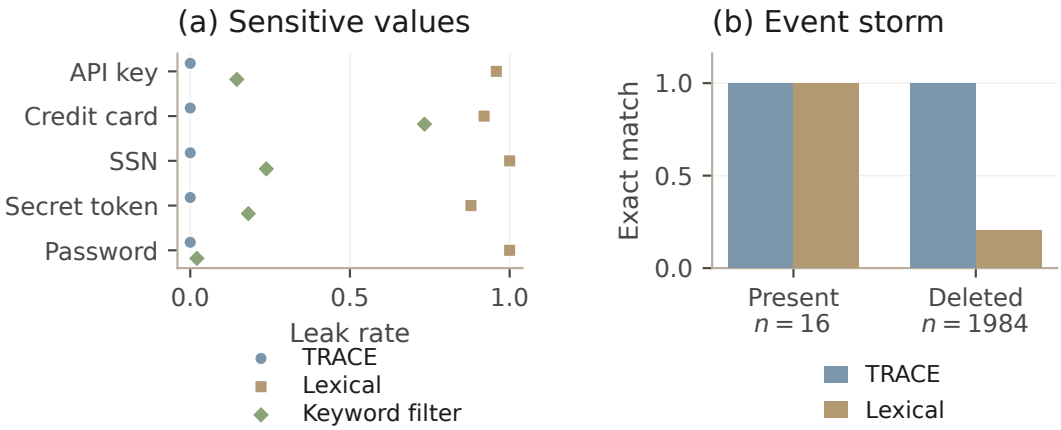

Figure 8: **Additional governance controls.** (a) Sensitive-value leakage. (b) Exact match after 1000 events per case.

lexical reader and keyword filter expose values missed by their surface-form checks, while the compiled-state gate blocks all observed leaks. This result assumes correct sensitivity labels. The event-storm test mixes assert, supersede, retract, and delete operations in a 5:3:1:1 ratio. Its 2000 queries target present or tombstoned values; no dense reader was run on this test. The test is strongly imbalanced toward tombstoned targets, so Figure 8b shows both regimes separately rather than relying on aggregate accuracy alone.

C.3 Recall and compliance under different return budgets

Each case contains ten currently-valid propositions, five forbidden ones, and five stale values a SUPERSEDE has replaced. Table 8 reports recall, leak-free compliance, and leaked items per query on 300 queries. TRACE returns a mean of three items there; both retrieval controls use top-10, so that table alone does not hold the return budget fixed. The asymmetry cuts both ways: a larger budget gives the retrieval readers more chances to surface a forbidden proposition and just as many more to surface a safe one, so we equalize it below and report both.

The claim the table supports is that, on the same compiled ledger, the retrieval readers place forbidden propositions in their returned sets while the executing reader does not, and that no TRACE answer was leak-free by accident of an empty return.

How to read k. Each case holds 20 retrievable rows: 10 currently-valid propositions, 5 stale originals left behind by the mid-life SUPERSEDE, and 5 deleted sensitive ones. Recall counts distinct hits on the 10 current ones, leaks count hits on the 5 deleted ones, and a returned row can be neither. Every reader returns exactly k distinct rows, so $k = \text{recall} \times 10 + \text{leaks} + \text{stale hits}$, and the third term is what the first two do not show. At $k = 10$ it is 0.00 for TRACE, 1.98 for lexical and 4.07 for dense: the retrieval readers spend a fifth to two-fifths of their budget returning values that a SUPERSEDE had already replaced. At $k = 3$ the same term is 0.00, 0.42 and 1.14. Execution removes stale and deleted rows from $\mathcal{S}(t)$ before ranking, so its whole budget lands on propositions in force.

Matched budgets. Because the readers return different numbers of items, we re-ran the comparison with the budget equalized in both directions on the same ledgers and queries (Table 9). Neither direction removes the gap; both widen it. Cutting the retrieval readers to three items costs them most of their recall (lexical $0.575 \to 0.196$, dense $0.439 \to 0.143$, both now below TRACE) because the safe propositions they were recovering at top-10 sat outside the top three, while their compliance stays far from clean at 0.500 and 0.653. Raising TRACE to ten items takes its recall to 1.000 with compliance still at 1.000 and no leak, since execution has already removed the forbidden propositions from $S(t)$ and a larger budget can only surface admissible ones. The budget asymmetry in Table 8 was therefore concealing the result rather than producing it: at equal budget TRACE leads on recall and compliance simultaneously, which is the comparison the earlier table could not make.

Table 8: **Recall and compliance.** 300 queries; TRACE returns three items on average, retrieval uses top-10.

Reader	Recall ↑	Compliance ↑	Leaks/query ↓
TRACE	0.300	1.000	0.00
Lexical-RAG	0.575	0.030	2.27
Dense-RAG	0.439	0.203	1.54

Table 9: **Recall and compliance at matched return budgets.** Same 100 ledgers and 300 queries as Table 8. Here k is the number of items every reader returns.

k	**Reader**	**Recall ↑**	**Compliance ↑**	**Leaks/query ↓**
3	TRACE	**0.300**	**1.000**	**0.00**
3	Lexical-RAG	0.196	0.500	0.62
3	Dense-RAG	0.143	0.653	0.42
10	TRACE	**1.000**	**1.000**	**0.00**
10	Lexical-RAG	0.575	0.030	2.27
10	Dense-RAG	0.439	0.203	1.54

D FORMAL-LAYER COVERAGE AND COMPONENT ATTRIBUTION

This section supports the claims stated in §3. §D.8 reports the bounded coverage of the transition algebra and the symbolic membership check; §D.9 reports the seven-axis component ablation and update-schedule study that complement the three-metric ablation in Figure 5.

D.1 STATE-REPRESENTATION ASSUMPTIONS FOR THE FOLD-COST CLAIM

Proposition 2 states that appending a single instruction can be absorbed by one $\oplus$-merge. The wall-clock cost of that merge depends on the state representation, not on associativity by itself. Our deployed state is a hash map attr $\to$ payload in which each payload is either a scalar Last-Write-Wins register or the deduplicated value set of an ACCUMULATE slot (§D.3). Under this representation the per-attribute merge is $O(1)$ in expectation; the cost is independent of $|L|$ but does grow with the number of distinct attributes touched by the query, and with the payload size for ACCUMULATE attributes. Deployments storing richer payloads, such as long lists of provenance pointers, must adjust the claim accordingly. Bulk replay from cold state stays linear in the number of instructions selected at t; the incremental claim covers only appends.

D.2 FAIL-CLOSE OUTPUT POLICY

Specification 2 obliges a compliant implementation to reject answers whose certificate fails soundness, minimality, or completeness. Our deployed pipeline enforces this policy as follows. After the heuristic slicer produces $\langle \sigma, \rho, a \rangle$, the verifier replays σ and compares its resolved state against the answer a. If the two disagree, or if the exclusion report ρ misses an inadmissible proposition relevant to q, the pipeline suppresses a and returns a structured refusal that records the failing check and the exclusion report.

The refusal path therefore fires on every certificate that fails any of the three checks. On the 1000 constructed programs no certificate fails soundness, minimality, or exclusion-report coverage, so the path is exercised by a different condition: a query whose answer target has itself been excluded by execution. There are 393 such queries in the 500-program stress regime and none in the live-only regime. For each, the certificate lists the target in its exclusion report and the pipeline returns that report in place of an answer. Appendix D.4 gives the accounting.

D.3 WHAT THE ALGEBRA COVERS, OPERATOR BY OPERATOR

Proposition 1 is stated over $(\text{State}, \oplus, \varnothing)$ and it is worth being explicit about which part of the system that object models, because the five instruction types are not all inside it.

The resolved state is a finite map from an attribute key $k = (\text{subject}, \text{attribute})$ to a payload. The payload has two components, and the second one matters: a *deletion flag* $d \in \{0, 1\}$ and a last-writer-wins register $r = (t_{\text{ing}}, \text{value})$, where the value may be the distinguished symbol † recording a RETRACT. ACCUMULATE attributes additionally carry a deduplicated set of values, held in insertion order; there is no capacity bound and no element is ever evicted. The lifts are

$$\tau(\text{ASSERT/SUPERSEDE}) = (0, (t, v)), \quad \tau(\text{RETRACT}) = (0, (t, \dagger)), \quad \tau(\text{DELETE}) = (1, \bot),$$

and $\oplus$ is pointwise on keys,

$$(d_1, r_1) \oplus (d_2, r_2) = \big(d_1 \vee d_2,\ \arg\max_{t_{\text{ing}}}(r_1, r_2)\big),$$

with set union on ACCUMULATE slots. Union is associative and commutative on the set of values, so the resolved slot does not depend on grouping; the order in which the values are rendered follows transaction order and is the part of ACCUMULATE that sits outside the single-valued statement. This map is the $R(t)$ of Definition 2: the scope check admits k into $\mathcal{S}(t)$ only when $d = 0$ and r holds a real value, so a key whose flag is set is never in force whatever its register says, and every projection π_q reads $\mathcal{S}(t)$ after that decision.

The deletion flag is *absorbing*, and that is what makes the fold agree with execution. It is not an ordinary tombstone value competing inside the register: were it one, the program $L = [\text{DELETE}(k, t_1),\ \text{ASSERT}(k, v, t_2)]$ with $t_2 > t_1$ would fold to v, because the ASSERT carries the larger ingestion stamp, whereas EXECUTE returns the empty state, since a deleted proposition cannot be revived (§D.6). Under the definition above the $\vee$ carries the flag past the later ASSERT and the projection drops the key, which is what EXECUTE does. RETRACT is deliberately not absorbing: it writes † into the register, so a later ASSERT with a larger stamp legitimately revives the proposition.

Disjunction is associative, $\arg\max$ over distinct ingestion stamps is associative, and a pointwise combination of associative operations on disjoint keys is associative, which is the whole of Proposition 1; the empty map is the identity. Both components are also insensitive to the order in which they are combined, so the fold does not depend on grouping, which is Proposition 2. We checked this correspondence rather than assuming it: on the single-valued fragment the corrected fold reproduces EXECUTE on every one of the $37{,}448$ programs given by all operator sequences of length at most five over one key, on $40{,}000$ random multi-key programs, and under $30{,}000$ re-orderings, with no mismatch, and it decides the program L above correctly (§D.8).

Four things sit outside the algebraic claim. Authorization scope and valid-time expiry are evaluated by π_q at read time against the query's principal and timestamp, so they are outside $\oplus$. Commutativity is not a monoid requirement and we claim no law: it holds whenever ingestion stamps are distinct, which they are by construction in the deployed ledger, and Table 13 reports 0.9705 only because its synthetic triples reuse stamps deliberately. Idempotence holds of our representation without being required of a monoid, and nothing depends on it. Operator interactions beyond the pointwise argument, such as a DELETE at t_1 composing with an ACCUMULATE at a later t_2 on the same key, are established within bounds. That particular interaction is in fact decided, not sampled: the symbolic check encodes all five instruction types for the liveness properties and returns no counterexample over every ledger with $N \leq 20$ and $M \leq 10$ (§D.8), and random testing adds no counterexample in $10{,}000$ random triples and 576 shuffled folds (Table 13). What remains unproven is behavior above those bounds and the value-level composition of ACCUMULATE with the LWW register, which is outside the single-valued statement. The algebraic core is proven; the operator-interaction layer is verified within bounds.

Where the exclusion report comes from. The resolved state holds one register per key, so it cannot by itself say what was withheld. The report is not read off it. `excluded_report` makes a second pass over the same event-time prefix in transaction order and records, per key, the instruction that last removed the key from force: a DELETE (reason `deleted`) or a RETRACT (reason `retracted`). A later ASSERT on a retracted key clears that record, because the key is in force again; after a DELETE nothing clears it. Keys that survive the fold but fail the projection are then added with reason `stale` (valid-time bound passed) or `unauthorized` (scope or sensitivity). Each entry carries the memory_id of the responsible instruction, so the report names versions and not just keys.

Two consequences. A value replaced by a later SUPERSEDE is not in the report: it was not withheld, it was overwritten, and the instruction that overwrote it is the one in σ. And the deployed report is built for the whole $(t, \text{principal})$ rather than filtered to the query, so it is a superset of the "relevant

to q" set in Specification 2; the completeness check in `verify` compares it against a report rebuilt from the source ledger under the same rule, which is why completeness is a comparison with the ledger rather than a property of the certificate alone.

Which time selects, which time orders. The two stamps do different jobs and the distinction decides the following case. Let A be asserted with event time 0 and a valid-time bound of 10, and let B be recorded later, with event time 10. A query at $t = 5$ must return A. Execution selects before it folds: the time slice keeps the instructions with $t_e \leq t$, so B is not in the fold at all, A resolves, and the bound 10 has not passed at $t = 5$, so A is in force. A reader that folded everything first and then tested the bound would hold only B, find it not yet valid at $t = 5$, and return nothing. This is why the query time is applied at the slice and not at the projection, and why the resolved state is a function of t rather than a single current snapshot filtered afterwards. The transaction stamp never selects; it only orders the fold, so that a correction recorded later wins over the entry it corrects.

D.4 Certificate service availability

A certificate interface has two failure modes that must be reported together: returning an answer whose certificate does not hold, and refusing to answer at all. Table 10 gives both. No certificate fails a check in either regime, so every refusal is triggered by the same condition: the query names a proposition that execution excluded.

What "live-only" selects. The two regimes differ only in how the certificate test samples answer targets: live-only draws them from the propositions in force at t, the stress regime from every proposition that appears in the ledger, including deleted and superseded ones. Neither setting edits the ledgers, and neither is applied to the natural-language benchmarks. The end-to-end runs pose whatever question the benchmark poses, including its deletion and invalidation questions, and answer from the resolved state in the normal way; the 78.6% refusal rate is a property of the stress sampler, not of a deployment workload.

Directed rejection checks. Every certificate the pipeline generates passes, so the pass rate alone does not establish that a bad certificate would be caught. We therefore took 500 passing certificates from the same program family and applied one targeted perturbation at a time, requiring the property it attacks to fail. Dropping a necessary instruction from σ is caught by soundness on $500/500$; padding σ with an instruction that decides no answer is caught by minimality on $500/500$; dropping an entry from ρ is caught by completeness on $478/478$ of the programs whose report is non-empty; and naming an excluded proposition in the answer is caught by the same check on $478/478$. Each of the three properties has a perturbation only it detects, and the verifier rejects on it.

Table 10: **Certificate outcomes and service availability.** Counts over 500 synthetic programs per regime, each checked against exhaustive subsequence search. The stress regime samples answer targets from every proposition in the ledger, so most of its queries ask about a proposition execution has excluded; the pipeline returns the exclusion report instead of an answer in those cases.

Outcome	Live-only	Any-target
Sound slice	500	500
Deletion-minimal slice	500	500
Slice = exhaustive minimum	500	500
Exclusion report covers all exclusions	500	500
Certificate fails a check	0	0
Query names an excluded proposition	0	393
Answered	100%	21.4%
Refused with exclusion report	0%	78.6%

Scope of the table. Two further quantities are outside this measurement. (1) *Task success under refusal.* The 500 programs are synthetic instruction sequences used to compare the slicer against an exhaustive reference; they are not benchmark questions, so no task score attaches to them. The end-to-end results in this paper run the live-only configuration, which carries no refusals. (2) *Whether refusal is the right product behavior.* Returning the exclusion report rather than an answer is what Specification 2 requires when the requested proposition has been deleted, but whether a deployment should instead answer "this was deleted on t" in natural language is a design question this measurement does not settle.

D.5 CERTIFICATE CONSTRUCTION AND VERIFICATION COST

The kernel timings in Figure 6b are for execution only. Table 11 adds the two certificate stages on the same ledgers, reported as a multiple of EXECUTE so the numbers do not depend on the machine. Ledgers are random programs over the operator set of §2; each cell is the median of five repetitions; the query asks for five live propositions.

Table 11: **Certificate overhead.** Median of five repetitions per cell. $\times$ columns are relative to EXECUTE on the same ledger. $|\sigma|$ is the certified slice size.

$\|L\|$	EXECUTE (ms)	`certify`	`verify`	**total**	$\|\sigma\|$	$\|\sigma\|/\|L\|$
10	0.025	1.8×	6.3×	8.1×	5	0.500
50	0.091	1.6×	3.4×	5.0×	5	0.100
100	0.226	1.4×	2.7×	4.1×	5	0.050
500	0.536	1.5×	2.4×	3.9×	5	0.010
1000	1.137	1.7×	2.3×	4.0×	5	0.005
5000	5.434	1.6×	2.5×	4.1×	5	0.001

The **total** column is `certify` plus `verify` and does not include the EXECUTE it is measured against, so a certified recall costs $1 +$ total executions in all. Over the measured range the factor settles near $4\times$ by $|L| = 100$ and stays there to $|L| = 5000$, adding 22 ms on a five-thousand-instruction ledger. We report that as a stable measured factor rather than a complexity result: the construction above is one pass over the ledger and the verifier is two replays plus $|\sigma|$ replays of the slice, so the work is linear in $|L|$ for fixed $|\sigma|$, but we have not proved a bound and the table only covers $|L| \leq 5000$ for this query shape.

The certified slice does not grow with the ledger: it stays at the five instructions the five answers depend on, falling from half the ledger at $|L| = 10$ to a thousandth at $|L| = 5000$. This is the cost of replaying an answer's support, which is what $|\sigma|$ bounds. It is not the cost of checking that the exclusion report is complete: that comparison is against the source ledger and scans it (Appendix G). The query here asks for five live propositions, so $|\sigma| = 5$ is a property of the answer, not a general bound; an aggregate over many facts carries a slice proportional to the facts it aggregates.

The small-$|L|$ rows are dominated by fixed per-call costs, which is why the relative factor is largest where the absolute cost is smallest (0.2 ms at $|L| = 10$). These are in-process single-thread timings of the deterministic kernel on one machine, median of five repetitions; absolute milliseconds are machine-dependent and the ratios are reported because they are more stable across machines, not because they are invariant. Compilation and verbalization are LLM calls and are excluded.

D.6 TOMBSTONE HANDLING IN THE SLICER

An earlier slicer selected, for each answer target, the last instruction acting on it, skipping the target only when that last instruction was itself a DELETE or RETRACT. Execution treats DELETE as a permanent tombstone: once a proposition is deleted, every later instruction on it is a no-op. A chain such as [DELETE, ASSERT] therefore left the proposition absent under full execution, while the slice, which contained the ASSERT but not the DELETE, revived it on replay, so the slice did not reproduce the executed state. This produced 80 unsound slices in the any-target regime and none in the live-only regime, where no answer target is deleted by construction.

The deployed construction. Algorithm 1 defines what a minimal slice is by reduction; `certify` does not run that loop. For each answer key it takes the instructions on that key with $t_e \leq t$, in transaction order, and keeps one instruction:

for each answer key k: chain $\leftarrow$ [$i \in L$: key$(i) = k$, $t_e(i) \leq t$] in t_x order
  if DELETE $\in$ chain **or** chain is empty **or** last(chain) is RETRACT: skip k
  else add last(chain) to σ

This is one pass over the ledger and yields $|\sigma| \leq$ the number of answer propositions, which is why the slice does not grow with $|L|$ (Appendix D.5). `verify` then checks the three properties independently of how σ was built: soundness by replaying σ and comparing the target values against a replay of the full ledger; minimality by removing each instruction of σ in turn and requiring a

target value to change; completeness by rebuilding the exclusion report from the source ledger and requiring the certificate's report to cover it and the answer to name none of it. Minimality is therefore verified in the sense of Specification 2 whatever the construction, and the exhaustive comparison of §5 independently confirms the constructed slice is also of minimum length on the tested programs.

The slicer now skips any target whose instruction chain contains a DELETE at or before t_{query}, matching the tombstone rule in EXECUTE. RETRACT is unchanged: it ends the current binding without tombstoning, so a later ASSERT legitimately revives the proposition and the last-instruction rule remains correct there. The change removes all 80 counterexamples, leaves the live-only regime at $500/500$, and brings the any-target regime to $500/500$ on soundness, deletion-minimality, and agreement with the exhaustive minimum. Deletion-minimality and globally minimum slice length are distinct properties in general; on these programs the heuristic satisfies both.

D.7 ADMISSIBILITY ACCURACY AND MISUSE ARE DIFFERENT MEASUREMENTS

AdmAcc and Misuse are different measurements and should not be collapsed. AdmAcc is per query and asks whether the value the reader returns is the admissible one. Misuse is scored per forbidden kind over the same queries and asks whether *any* forbidden value also appears in what the reader surfaced. A reader can in principle return the correct current value while also surfacing a stale one, so a perfect AdmAcc does not by itself establish zero leakage.

An earlier version of this table reported one misuse event for TRACE ($1/224$ on AUTOMEMORYBENCH) and we treated it as a governance failure. It was not: it was a defect in the scorer. The routine that decides which tokens distinguish a forbidden value from the admissible one tokenized with a pattern that retains colons, while the returned text was normalized with a pattern that strips them. On the one affected query the admissible answer contained `plan:` and the forbidden value contained `plan`, so a word present in *both* strings was counted as unique to the forbidden one. The returned state on that query is byte-identical to the admissible answer and the forbidden value is not a substring of it. Putting both sides on the same tokenizer removes the asymmetry. The fix is symmetric across systems, and it moves the baselines too: lexical retrieval's AUTOMEMORYBENCH AdmAcc rises from 0.893 to 0.906.

Table 2 carries the corrected value; the run that produces it is the admissibility matrix regenerated under the current scorer (`run_admissibility.py --all`, artifact `admissibility_matrix_corrected_scorer.json`), and every proxy cell in that table reads from it. Two AUTOMEMORYBENCH control numbers appear in this paper and they are two readers over the same 224 probes, not two versions of one result: 0.906 is the lexical proxy of Table 2 and 0.812 is the bge-m3 cosine top-1 dense reader of Table 18, which Table 19 carries across unchanged. They differ in retrieval method and in nothing else.

With the corrected scorer TRACE records no misuse event on any of the 659 fixed-ledger queries, against 423 for the lexical reader. What the metric bounds is still specific: exact token containment at the read boundary, over the evidence the reader surfaces. A paraphrased disclosure, or one a downstream verbalizer introduces, is outside it.

D.8 FORMAL-LAYER COVERAGE

Table 12 reports the live-only versus any-target certificate test.

The tests below check different properties and do not establish unrestricted implementation equivalence.

Algebraic checks. The random-triple suite tests the encoded transition algebra; it should not be read as a proof of every operator path in the execution engine.

Fold against the executor. Proposition 2 claims that folding the lifted instructions reproduces EXECUTE, so we tested that correspondence directly on the single-valued fragment (ASSERT/SUPERSEDE/RETRACT/DELETE) with the absorbing deletion flag of §D.3. The fold agrees with EXECUTE on all $37{,}448$ programs formed by every operator sequence of length at most five over one key with two candidate values, on $40{,}000$ random multi-key programs, and the folded result is unchanged under $30{,}000$ re-orderings: no mismatch in any of the three. The check includes the sequence $[\text{DELETE}, \text{ASSERT}]$, which separates the absorbing flag from an ordinary in-register

Table 12: **Certificate checks against exhaustive search.** Programs passing each check, out of 500 per regime. *Live-only* queries name a live target; *any-target* may name a tombstoned or superseded one. The *Answered* row is a property of the queries, not the certificates: 393 any-target queries ask about a proposition execution has excluded, and the fail-close policy of Appendix D.2 refuses them with their exclusion report.

Check	Live-only	Any-target
Soundness	500/500	500/500
Deletion-minimality	500/500	500/500
Slice = exhaustive min	500/500	500/500
Slice size match	500/500	500/500
Slice index-set match	500/500	500/500
Exclusion report complete	500/500	500/500
Answered (query well-posed)	500/500	107/500

tombstone and which the earlier statement of the algebra resolved the wrong way. ACCUMULATE's value-level composition with the register stays outside this correspondence, as below.

Table 13: **Algebraic test coverage.** The suite runs 24 cases over 544 transitions and 10,000 random triples at seed 7. Commutativity is not required.

Property	Pass rate	# trials	Verdict	Scope
Associativity $a \oplus (b \oplus c) = (a \oplus b) \oplus c$	1.0000	10000	✓	monoid law
Idempotence $a \oplus a = a$	1.0000	10000	✓	extra property (not required)
Identity $\emptyset \oplus a = a \oplus \emptyset = a$	1.0000	10000	✓	monoid law
Commutativity (design-permissive)	0.9705	10000	$\approxeq$	only when (s, a, t_{ing}) ties
Fold order-invariance (24 shuffles/case)	1.0000	576/576	✓	case-level
Engine uniqueness `execute` $\equiv$ fold $+ \pi$	1.0000	24/24	✓	engine-level

$N = 24$ cases, 544 transitions, 10000 random triples, shuffle depth 24. Seed 7.

Bounded SMT checks. The symbolic model exhaustively searches every ledger up to $N = 20$ instructions over $M = 10$ attribute keys for violations of the three properties in Table 14: tombstone absorption, delete compliance, and LWW value equality. The search covers all operator orderings, all key assignments, and all event-time assignments over a four-tick clock, with t_{query} fixed at the final tick and values drawn from a three-symbol alphabet. That is 162 (N, M) cells per property, 486 solver calls, and every one returns UNSAT in 44 s total. Tombstone absorption and delete compliance encode all five instruction types including ACCUMULATE, so the DELETE-then-ACCUMULATE interaction is decided symbolically rather than sampled. LWW value equality is a statement about a single-valued register and excludes ACCUMULATE by definition, not by omission. UNSAT means there is no counterexample within the bound; it is not a proof at unbounded size, nor of certificate correctness.

Table 14: **Bounded SMT search.** Exhaustive over all ledgers within each bound; a cell is one (N, M) solver call. Every cell is UNSAT.

Property	ACCUMULATE	N	M	Counterexamples
Tombstone absorption	in scope	3–20	2–10	0 / 162 cells
Delete compliance	in scope	3–20	2–10	0 / 162 cells
LWW value equality	out of scope	3–20	2–10	0 / 162 cells

ACCUMULATE liveness is inside the tombstone and delete-compliance checks; the LWW *value* equality is single-valued by definition and excludes it. UNSAT $\Rightarrow$ zero counterexamples. $t_{\text{query}} = 3$, $T = 4$.

Noise probe. A separate toy probe inserts 0, 10, 50, or 200 distractor operations and returns a one-instruction slice in each setting, showing that slice size tracks dependency rather than ledger size. It is a narrow construction, not the cost measurement, which §D.5 reports directly.

D.9 ATTRIBUTION AND UPDATE SCHEDULING

The component ablation is reported once, in Figure 5. Its intervals come from 1000 bootstrap resamples drawn over cases rather than queries, so each interval accounts for the 21 queries a case contributes together. Table 15 adds a different seven-axis attribution probe; it is not a second rendering of the main three-metric ablation. The governance and write-gate removals reduce their target-axis scores by 0.997 and 1.000, respectively.

Table 15: **Seven-axis attribution.** Changes relative to the full pipeline; the bottom row gives absolute reference scores.

Variant	AMQ	task	write_f1	update	safety	memdep	statsens	Predicted hit
no_govern	-0.260	+0.000	+0.000	-0.576	-0.997	+0.000	+0.000	safety
no_writegate	-0.252	-0.083	-1.000	+0.000	+0.000	+0.000	+0.000	write_f1
no_supersession	-0.160	+0.000	+0.000	-0.535	-0.535	+0.000	+0.000	update
no_sensitivity	-0.120	+0.000	+0.000	+0.000	-0.587	+0.000	+0.000	safety
no_counterfactual	+0.000	+0.000	+0.000	+0.000	+0.000	-0.875	-1.000	statsens
no_bitemporal	-0.015	+0.000	+0.000	+0.000	+0.000	+0.000	+0.000	update
Full (abs.)	0.936	0.979	1.000	1.000	1.000	0.875	1.000	–

Table 16 changes the update schedule while holding the other components fixed. Eager and lazy execution have matching scores in this probe, so the evidence favors explicit state resolution rather than an accuracy advantage unique to query-time scheduling. The source's safety-slice score is not the joint benchmark StrictCore metric used in Table 3.

Table 16: **Update schedule.** 24 cases, 12 probes each (288 queries). Safety-slice is a diagnostic score, not joint StrictCore.

Knob M	Safety-slice	Task	AMQ	Governance safety	Singlehop AMQ
never	0.004	0.305	0.497	0.000	0.582
eager	1.000	0.367	0.760	1.000	0.778
lazy	1.000	0.367	0.760	1.000	0.778

Backbone swap. Table 17 retains the per-subtask and overall results under a fixed framework and scorer. Variation across four backbones shows configuration sensitivity; it does not by itself identify which module causes the remaining errors.

Table 17: **Backbone sensitivity.** StructMemEval uses a Qwen3.7-Max judge; MemBench uses MCQ exact match. All rows are completed recorded runs.

Backbone	**Overall**	**State**	**Tree**	**Recsys**	**Count**
StructMemEval					
GPT-5.5	0.240	0.857	0.900	0.139	0.000
Qwen3.7-max	0.245	0.643	1.000	0.150	0.000
DeepSeek-v4-pro	0.236	0.643	1.000	0.139	0.000
DeepSeek-v4-flash	0.231	0.786	0.950	0.117	0.133
Backbone	**Overall**	**F-Third**	**R-Third**	**F-First**	**R-First**
MemBench					
GPT-5.5	0.716	0.807	0.881	0.475	0.905
Qwen3.7-max	0.694	0.800	0.871	0.423	0.901
DeepSeek-v4-pro	0.704	0.788	0.854	0.477	0.890
DeepSeek-v4-flash	0.712	0.790	0.857	0.497	0.888

E READER, COMPILER, AND CONFIGURATION DIAGNOSTICS

This section reports the deployed-pipeline diagnostics that scope the main-table numbers: how the dense-RAG reader and mem0 SDK compare on admissibility and certificate coverage (§E.1), the conversational-benchmark scores under the finalized pipeline (§E.2), and how the Flash compiler behaves as the output budget and proposition-loss rate are swept (§E.3). Together they document sources of variability that Figure 4 and Table 3 hold fixed.

E.1 READER AND CERTIFICATE COMPARISONS

Table 18 reports the dense-RAG reader (bge-m3, cosine top-1) and the limited mem0 SDK smoke run. Unavailable SDK cells remain unreported; dense retrieval is a separate control and cannot substitute for those missing measurements.

Table 18: **Reader-level admissibility.** Dense-RAG evaluation with limited mem0 SDK smoke results. TRACE AdmAcc is in Table 2, and N/A marks a measurement we do not have. The TRACE and dense-RAG columns use the corrected tokenizer of §D.7. That fix moved dense-RAG by a single query and left every other cell unchanged, so the comparison does not turn on it. Daggered mem0 cells predate the fix and could not be re-scored, as the run kept only aggregate counters. The mem0 AUTOMEMORYBENCH cell is five probes rather than the 224 of its row, so it is an observation and not an estimate of mem0's rate on this split.

		TRACE	dense-RAG (bge-m3)		mem0 (smoke)
Bench	n	Misuse↓	AdmAcc↑	Misuse↓	Misuse↓
AMB	224	0.000	0.812	0.304	$0.400^{\dagger}$ (n=5)
MemBench	23	0.000	0.826	0.609	$0.609^{\dagger}$
MintEval	370	0.000	0.281	0.719	*N/A*
StructMem	42	0.000	0.238	0.762	*N/A*

Table 20 decomposes the certificate-interface comparison on its evaluated inputs. A ranking reader's citations replay, so soundness and minimality are scorable and some readers reach 1.00 on both; completeness is the half none reach. A zero completeness score for a retrieval reader means it does not supply the exclusion explanation required by this protocol, not that retrieval cannot be extended to do so. These selected recalls do not cover the excluded targets in Table 12.

Proposition 3 (Completeness requires execution)**.** No reader with fold schedule $M = \text{never}$ can produce a complete exclusion report ρ for a query whose relevant-inadmissible set is non-empty. Soundness and minimality remain scorable for such a reader.

A complete ρ must name every relevant proposition that execution withheld together with the instruction responsible, which presupposes a partition of the stored items into admissible and withheld. Under $M = \text{never}$ no recorded update is ever resolved, so that partition is never formed and none can be reported: whenever the relevant-inadmissible set is non-empty, completeness fails by construction. That is the column on which every baseline in Table 20 scores 0.00, while soundness and minimality are untouched by the argument because they constrain σ only against the reader's own answer. The statement concerns the interface, not hardness: a retrieval system can be extended to satisfy completeness, but the extension has to resolve the recorded updates, and that step is execution.

Matched eager materialized-state reader. The strongest fair control is not retrieval but a state-aware store that consumes the *identical* compiled operations (ASSERT/SUPERSEDE/DELETE with sensitivity), materializes current state eagerly at write time under tombstone-aware last-writer-wins and the same access gate, and emits the same exclusion report. Table 19 runs this control head to head with TRACE on the identical inputs. On present-time admissibility over the 224 AUTOMEMORYBENCH probes it reproduces TRACE (both 1.000 AdmAcc, misuse 0.000) while dense-RAG over the same content reaches 0.812. What wins over retrieval is therefore executing recorded state, not our output schema. The reader diverges from TRACE the moment a query carries a time argument: holding only current state, it answers every time-indexed query with the latest value and collapses

to the $1/3$ ceiling (0.333) on the cross-time set, where EXECUTE(L, t) re-folds the ledger at each queried t and answers at 1.000. What this isolates is the cost of keeping only the present state. Answering as-of queries requires retaining history in some form, and a replayable ledger is one option among several: periodic snapshots or valid-time indexed versions would also serve, at their own storage and maintenance cost. We measured the present-state reader, not those designs, and make no claim that they reduce to EXECUTE. The comparison that would settle the remaining question is one in which both systems can answer as-of queries, scored on certificate validity, exclusion reporting, and update versus query cost; we have not run it.

Table 19: **Matched eager materialized-state reader vs. TRACE.** Both readers take the same compiled operations, the same exclusion schema and identical probe inputs. Present-time admissibility is measured on the 224-query AUTOMEMORYBENCH split, cross-time accuracy on 100 cases at three query times each. The state-aware control ties TRACE on present state, then collapses to the latest-value ceiling once queries are time-indexed. The dense-RAG row is carried from Table 18 on the same probes; we ran no dense cross-time control, and the lexical control that we did run sits at the same 0.333 ceiling (§C.1).

Reader	Present AdmAcc	Present Misuse	Cross-time Acc
TRACE EXECUTE(L, t)	1.000	0.000	1.000
Eager materialized reader	1.000	0.000	0.333
Dense-RAG (bge-m3 top-1)	0.812	0.304	—

Table 20: **Certificate-interface decomposition.** Additional baseline checks; TRACE's joint result is in Table 2. Soundness, minimality, and completeness are not additive quantities.

Bench	System	n	Sound	Minimal	Complete	Verifiable
MemBench	dense_rag_recency	23	1.00	0.00	0.00	0.00
	dense_rag_lexical	23	0.96	0.00	0.00	0.00
	mem0_style	23	1.00	1.00	0.00	0.00
	mem0_relations	23	1.00	1.00	0.00	0.00
MintEval	dense_rag_recency	370	1.00	0.00	0.00	0.00
	dense_rag_lexical	370	0.60	0.00	0.00	0.00
	mem0_style	370	1.00	1.00	0.00	0.00
	mem0_relations	370	1.00	1.00	0.00	0.00
StructMem	dense_rag_recency	42	1.00	0.00	0.00	0.00
	dense_rag_lexical	42	0.76	0.00	0.00	0.00
	mem0_style	42	1.00	1.00	0.00	0.00
	mem0_relations	42	1.00	1.00	0.00	0.00

E.2 CONVERSATIONAL BENCHMARKS UNDER THE FINALIZED PIPELINE

The finalized pipeline of Appendix A compiles with Flash, reads the full resolved state, and verbalizes with GPT-5.5. It scores 0.782 on the 500-question LongMemEval set and 0.552 on the full LoCoMo set ($n = 1540$), both under a DeepSeek-V4-Pro judge, with the matched dense-RAG control of Appendix A at 0.660 on the same LongMemEval questions under the same answerer and judge. Compilation and readout are therefore sources of variability on these two conversational benchmarks, not evidence that conversational memory falls outside the scope of execution-based recall. The single-factor evidence is the compiler-budget sweep of Figure 9a and the proposition-loss sweep of §E.3.

E.3 COMPILER SETTINGS AND PROPOSITION LOSS

With `deepseek-v4-flash` and an output budget of 4096, the task reruns score 0.720 on MemBench and 0.306 on StructMemEval (earlier runs: 0.719 and 0.345). They lack matched oracle/no-compile controls and governance scores.

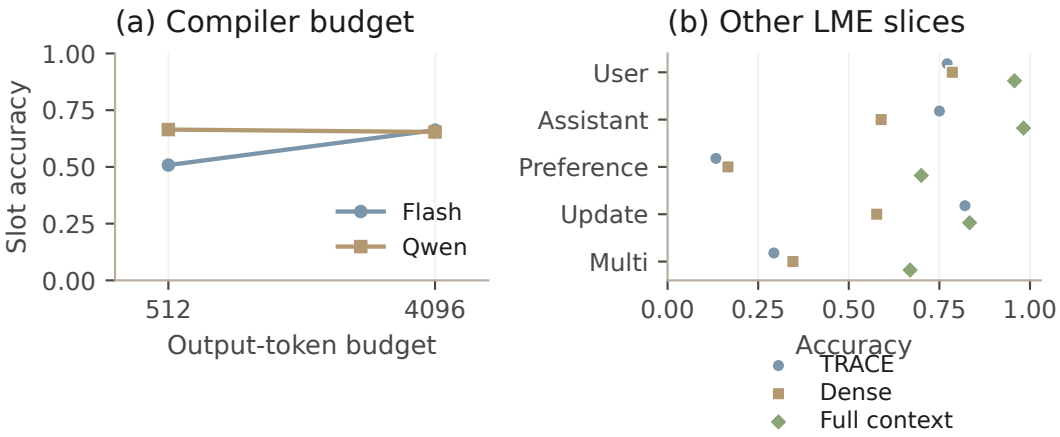

Figure 9: **Compiler and task boundaries.** (a) Output-budget sensitivity. (b) Non-temporal LongMemEval accuracy; full context is an unmatched reference.

Figure 9a shows a budget effect on the Flash compiler, while the Qwen control is comparatively stable. Both use narrative fallback at budgets 512 and 4096. Flash has 500 scored cases per run; Qwen has 495 and 500 after errors. No result below 512 is inferred. Panel (b) uses a GPT-4o judge; its full-context reference uses GPT-4.1, not a matched architectural oracle. Temporal scores appear only in Figure 4a. Panel (b) preserves both positive and negative task differences. Remaining full-context headroom does not distinguish scope limitations from compiler or readout errors.

Table 21 drops 0–40% of structural propositions. Narrative-body fallback preserves much of the task accuracy in this test, but those scores alone do not establish that fallback respects the structured state's governance filters; §E.4 measures that directly. The body-off condition exposes the dependence on extracting propositions correctly.

Table 21: **Compile degradation curve.** Task accuracy vs. proposition drop rate. *body-on*: narrative body as read-time fallback. *body-off*: compiled state only.

Setting	0%	10%	20%	40%
MemBench (on)	0.710	0.704	0.712	0.714
MemBench (off)	0.570	0.563	0.542	0.507
StructMem (on)	0.223	0.179	0.183	0.183
StructMem (off)	0.183	0.096	0.105	0.048

E.4 Governance under narrative fallback

The narrative-body path lets the verbalizer read raw source text when a structured proposition is missing. That text enters $S(t)$ as a narrative proposition carrying no sensitivity label, and `visible()` treats an absent label as non-sensitive, so the governance gate passes it. We measured what that costs.

Setup. 100 synthetic cases, each a ledger with three public slots, one sensitive slot (API key, card number, SSN, token, or password) labeled `secret`, and the raw episode text carrying the same secret verbatim. Queries are issued with `allow_sensitive=False`. Three adversarial queries per case probe for the secret (direct, lexical, semantic); two benign queries have structured answers; one benign query has an answer that appears *only* in the narrative body, which is what the fallback path exists to recover. We cross two compilation conditions (the sensitive slot extracted correctly, or dropped to simulate a compiler miss) with two readouts, the structured projection and the fallback projection. Leakage is scored at the read boundary, on the evidence handed to the verbalizer, so the measurement bounds the governed content a downstream verbalizer is given to work from rather than any particular verbalizer's output.

Result. Table 22 reports it. The structured readout never leaks, confirming §C.2, but recovers none of the narrative-only facts. The fallback readout recovers all of them and leaks the governed secret

on *every* adversarial query. The leak does not require a compiler miss: the ablated and correct compilation cells are identical at 1.000, because the raw text carries the secret whether or not the structured slot was extracted. This is worse than the conditional gap we previously described.

Fix. Redacting from the narrative body any literal the structured layer labeled sensitive removes the leak entirely (1.000 → 0.000) while leaving benign recall and narrative-only recall at 1.000. The alternative of marking the whole narrative proposition sensitive also removes the leak, but drops narrative-only recall to 0.000, which is equivalent to disabling fallback. Redaction is therefore strictly better on this test, and it is what the compiler now does before attaching the narrative proposition. What this buys is precise: the governance gate no longer depends on the structured *slot* surviving compilation, because the ablated and correct cells are both 0.000 after redaction. It still depends on the literal having been labeled sensitive somewhere, so a value the compiler neither extracted nor labeled remains outside the redactor's reach, and is bounded by compilation completeness. Redaction is applied when the narrative is attached; if a value is later DELETEd or a scope is tightened, the structured layer handles that on the next recall, but the already-attached narrative body is not retroactively scrubbed. Re-attaching narratives on lifecycle events is a deployment step outside the scope of this experiment.

Table 22: **Governance under narrative fallback.** Across 100 cases the reader answers 300 adversarial queries and 200 benign ones, plus 100 that only the narrative body can answer. The principal is denied sensitive access. Leakage is the governed literal reaching the verbalizer's evidence.

Readout	Mitigation	Leak ↓	Benign ↑	Narr.-only ↑
Structured only	—	0.000	1.000	0.000
Fallback	none	**1.000**	1.000	1.000
Fallback	redact	**0.000**	1.000	**1.000**
Fallback	gate	0.000	1.000	0.000

One caveat on reading the numbers: the secrets are synthetic literals and leakage is exact string containment, so the test detects verbatim re-surfacing and not paraphrase.

F Execution Cost and Design Space

This section situates the execution kernel against the memory-system design space. The latency figures back the cost claim in Figure 6b, and the design-space tables extend Table 1 to five further systems. The latency measurements themselves appear only in Figure 6b. The recorded timings use five repetitions. Incremental execution starts from a pre-materialized checkpoint and replays ten new transitions; checkpoint construction, compilation, verbalization, certificate construction, and verification are excluded. A separate nine-trial context-window toy test reports error 0.667 for a six-item native window and zero for ledger execution. This small controlled probe does not establish superiority at the agent level.

Tables 23 and 24 give the qualitative design-space mapping and capability coding. They describe interfaces and design choices, not an additional quantitative benchmark.

Each axis in Table 24 is coded against a stated criterion, and the coding is read off each system's public documentation and release paper, not measured by us. *Train-free* means the memory layer needs no gradient update to accept a new fact. *Auditable* means a public interface returns, for a given answer, which stored items produced it. *Editable* means a documented operation revises or removes a stored item in place, as opposed to appending a correction. *Branchable* means two alternative histories can be held at once and queried separately. *State-native* means the read path returns a resolved value rather than a ranked list the caller must reconcile. A ∼ marks a capability the documentation describes only for part of the store or only through an internal API. Because the coding tracks documented interfaces, a system can be marked absent on an axis that it supports but does not expose.

G Limitations and Measurement Scope

What certificate verification does and does not settle. Re-executing σ and inspecting ρ are local checks. Confirming that ρ covers every relevant inadmissible proposition requires comparison

Table 23: **Five-knob mapping** $(E, \oplus, M, \pi, \text{substrate})$ for five additional systems not detailed in Table 1.

System	Encoding	Update	Timing	Readout	Storage
MemOS	MemCubes	append / schedule	write	rank / route	multi-cube
A-Mem	linked notes	note evolution	write	linked ranking	note graph
SimpleMem	labeled facts	append	write	keyword / entity	inverted index
LightMem	summaries	batch append	deferred	ranking	vector store
MemInsight	attributes	append	write	attribute ranking	row store

Table 24: **Independent axes.** ✓=yes, ∼=partial, –=no.

System	train-free	auditable	editable	branchable	state-native
mem0	✓	–	∼	–	–
Zep	✓	∼	–	–	∼
MemOS	–	–	∼	–	–
MemoryOS	✓	–	–	–	–
A-Mem	✓	∼	∼	–	–
SimpleMem	✓	–	–	–	–
LightMem	✓	–	–	–	–
MemInsight	✓	∼	–	–	∼
TRACE	✓	✓	✓	✓	✓

against the source ledger, so completeness is verified against the ledger rather than from the certificate alone. §3 states these properties; the deployed slice-minimization heuristic that produces σ is evaluated against them. Confirming completeness is therefore the step that scans the ledger, and a query whose answer aggregates many facts carries a correspondingly larger σ.

Execution semantics operate on a recorded ledger, so the open boundary is compilation from natural-language history into typed transitions: an update the compiler omits or misattributes is outside what execution can resolve. The narrative-body fallback recovers most of that information in the degradation study, and its governance path is measured and closed (Appendix E.4): redacting sensitive literals from the narrative body before it is attached keeps the governed value out of the evidence handed to the answerer on every adversarial query, with no cost in recall.

The slicer's slice matches the exhaustive minimum on all 1000 constructed programs, and `verify` confirms minimality by leave-one-out on all 435 evaluated recalls; agreement at that scale is measured rather than proved, and symbolic verification covers selected predicates within stated bounds (§D.8). The admissibility and certificate statements of §2 are specifications a compliant implementation must meet. Forgetting here means exclusion from the recall interface, with physical erasure supplied by the store underneath it. Comparisons across vendors carry the protocol of their source runs; Appendices A and B record the settings behind each column, and the scope of each individual measurement is recorded below.

The rest of this section records the scope of each measurement reported in the main text.

What a DELETE binds. The tombstone is set on the key $(\text{subject}, \text{attribute})$, not on one version of it, and it is permanent in the forward direction: after a DELETE on a person's address, a later ASSERT on that key is a no-op and the address does not come back. This is the stronger of the two readings, and it is deliberate, because it is what makes "deleted stays deleted" a property of execution rather than of a filter. A deployment that wants to withdraw one recorded value while keeping the attribute writable uses RETRACT, which ends the current binding without tombstoning.

The second choice is what a query about an earlier time may see. EXECUTE selects instructions by event time, so a DELETE recorded with a later event time is not in the fold for an earlier t, and an as-of query before the deletion reproduces the state as it stood then, including the value deleted afterwards. That is the behavior historical replay requires, and it is separate from the present-time governance guarantee, which is what the deletion and adversarial experiments measure. A deployment that must

also suppress deleted content in historical views applies the tombstone at the projection instead of the slice; we do not measure that configuration.

Governance fallback. Leakage in Appendix E.4 is scored by exact containment of synthetic secrets at the read boundary, which bounds what any verbalizer is given to work from but does not score a paraphrase introduced later in generation. Zero leaks across 600 generated attacks is evidence for that attack distribution.

Formal layer. Propositions 1 and 2 claim associativity, identity, and the fold equivalence; neither claims $\oplus$ is idempotent (Table 13 records idempotence as an observed property of our state representation, not a monoid requirement). The symbolic check decides the permanent-tombstone rule within its bounds; certificate construction, authorization, valid-time expiry, and the value-level composition of ACCUMULATE with the last-write register lie outside it and are carried by the certificate contract and its tests (§D.8, Appendix D.3). The slicer's agreement with exhaustive search is empirical agreement on a finite sample rather than a proof of minimality in general.

Slicer minimality. The 1000 constructed programs are drawn from one generator with three subjects and three attributes, so exhaustive agreement on that sample does not by itself establish minimality for every ledger.

Availability regime. In the stress regime the pipeline refuses 78.6% of queries because they ask about deleted propositions. That rate follows from how the regime samples targets and is not a deployment workload (Appendix D.4).

Agent controls. The non-memory agent controls in Table 3 keep no memory layer and re-read the transcript each turn. Three of the four post a higher StrictCore and Task than TRACE on AUTOMEMORYBENCH; what those rows bound is full-context access, not memory semantics, and they convert only 0.46–0.62 of their task successes into joint passes.

Temporal comparison. The temporal-accuracy runs in Figure 4a differ in answerer configuration across systems, so that panel compares the pipelines as they are deployed rather than one answerer under a controlled ablation.

Deletion reader. The deletion evaluation uses a simulated hard-delete reader that removes the memory row while retaining a readable audit body. It establishes that this reader leaks; deployed-SDK deletion behavior would require its own run.

Whole-system boundaries. A missing deletion is one failure mode; a wrong target, operation, timestamp, sensitivity label, or authorization scope can also change the resolved state, the certificate implementation adds its own failure modes, and a verbalizer may assert claims a structured answer does not cover. The formal interface specifies what execution does with a given ledger, and those surrounding boundaries are evaluated separately.

Table provenance. Several comparison tables mix historical configurations or vendor reports with local runs; the main table's LongMemEval column in particular is not a matched ranking. The older conversational diagnostics lack the settings needed for a matched comparison, and some experimental cells are unreported.