BINDMEM: EXECUTABLE MEMORY FOR LONG-HORIZON REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents must interpret and combine information across past interactions, yet access to supporting evidence does not ensure correct reasoning. Even a successfully computed value may fail to satisfy the question's required scope or precision. We introduce BINDMEM, a neuro-symbolic memory interface that preserves source-local interpretations (WRITE), executes query-dependent programs (READ), and checks answer requirements before disclosing derived values (DISCLOSE). WRITE materializes source-local interpretations as provenance-linked typed records whenever their inputs are available from the source context. READ compiles each question into a typed program and an answer contract, then executes it over the retrieved records while tracking supporting evidence and unresolved conditions. DISCLOSE supplies a computed value only when it meets the requested precision and scope; otherwise it supplies the selected evidence alone. The final reader retains the original retrieved text throughout. Across six benchmark variants with two instruction-tuned readers, BINDMEM attains the highest overall score in all 12 comparisons against seven memory baselines, and it also improves a frontier reader. Fixed-retrieval controls isolate the contributions of stored interpretations, execution-selected evidence, and selective disclosure, and with identical stored records BINDMEM outperforms the evaluated Python pipelines for both readers.

1 INTRODUCTION

Long-horizon conversational agents must answer questions using information accumulated across past interactions. Prior work supports this capability through persistent memory management, structured representations, and retrieval. MemGPT and MemoryBank maintain information beyond the immediate context (Packer et al., 2023; Zhong et al., 2024); HippoRAG uses graph-based associations for retrieval (Gutiérrez et al., 2024); A-MEM, Mem0, and MemGAS develop mechanisms for organizing and accessing long-term memories (Xu et al., 2025; Chhikara et al., 2025; Xu et al., 2026).

Retrieving relevant evidence, however, does not complete the reasoning required to answer a question. Sources may contain expressions whose meanings depend on their original context, and questions may require selection, comparison, or aggregation across multiple records. Our oracle-evidence diagnostic illustrates this retrieval–execution gap: even with annotated supporting turns, two instruction-tuned readers achieve only 22.2–25.0% accuracy on temporal questions and 52.3–59.1% on multi-hop questions, compared with 72.3–76.8% on single-hop questions. These results motivate explicit support for interpreting and combining evidence after retrieval.

Instead of leaving this computation entirely to the reader, programmatic reasoning externalizes parts of it into executable operations. PAL and Program of Thoughts delegate computation to executable code (Gao et al., 2023; Chen et al., 2023), while broader neuro-symbolic approaches combine language-model capabilities with structured execution (Cheng et al., 2023; Patel et al., 2024). Long-horizon memory introduces two additional questions: which source-local interpretations should be preserved before a future query is known, and when does the available evidence justify disclosing a computed result?

The first question concerns *when to interpret a source*. An event described as "yesterday" can be assigned a date when its source timestamp is available. Preserving this interpretation makes it usable even if a later representation omits that timestamp. The future question determines which other event

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.

to compare it with and which relation to compute. Inspired by binding-time analysis (Nielson & Nielson, 1988), we organize this distinction by input dependencies: source-local interpretations can be materialized at WRITE, while query-dependent selection and composition remain at READ.

The second question concerns *when to disclose a value* to the reader. A computation can be numerically well-defined without fully answering the question. For example, if a question asks for the total transportation cost of a trip, summing the retrieved fares is sufficient only when the evidence specifies a complete list of transportation fares. Thus, successful execution must be accompanied by checks against the question's requested scope and precision before disclosing results to the reader.

Together, these considerations lead us to formulate memory reasoning as staged interpretation and execution under explicit answer requirements. Input dependencies determine what can be materialized before a query arrives, while the query determines how those interpretations are composed and what evidence is required to support the resulting answer. We introduce BINDMEM to implement this formulation through WRITE, READ, and DISCLOSE. WRITE attaches typed interpretations and provenance to the source text. READ uses a constrained frontend with language-model assistance to construct MEMORYIR programs that compose standard relational operations (Codd, 1970) with arithmetic and assertions. Execution propagates supporting evidence and unresolved conditions through *selection, linking, and aggregation*. DISCLOSE provides evidence from execution as context for the reader's reasoning, adding a derived value when it satisfies the question's requirements.

Contributions. We identify a retrieval–execution gap through oracle-evidence diagnostics and formulate memory reasoning as staged interpretation and execution under explicit answer requirements. We propose BINDMEM, which connects source-local interpretations to query-dependent typed programs, propagates provenance and unresolved conditions through execution, and checks answer requirements before disclosing derived values. Across six benchmark variants and two readers, BINDMEM achieves the highest score in all 12 benchmark–reader comparisons. Fixed-retrieval controls isolate the contributions of stored interpretations, query-dependent composition, and value disclosure.

2 MOTIVATION AND PROBLEM SETTINGS

2.1 MOTIVATION: THE RETRIEVAL–EXECUTION GAP

Long-horizon memory systems preserve, organize, and retrieve past information to construct context for queries (Gutiérrez et al., 2024; Xu et al., 2025; 2026). We study the key step that follows retrieval: interpreting evidence in its source-local meaning and combining it as the query requires.

Temporal and multi-hop questions tend to be harder than single-hop questions. The difficulty can be attributed to two sources: assembling the required evidence and reasoning over it once available. We first examine evidence assembly, which requires support from multiple sessions. In a natural retrieval diagnostic on LoCoMo (Maharana et al., 2024), a supporting session counts as available when it appears in the top-10 retrieved sessions. At least one supporting session is available for 89.4% of multi-hop questions, but all supporting sessions are jointly available for only 30.1%. However, *does this retrieval bottleneck fully explain the performance gap?*

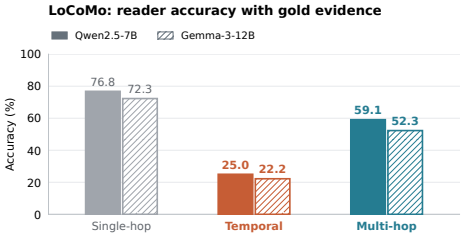


Figure 1: **LoCoMo score with oracle evidence.** Both readers receive the annotated supporting turns from LoCoMo. Temporal and multi-hop scores remain below the single-hop score, showing that access to supporting evidence and their session timestamps alone does not ensure correct answers. No component of BINDMEM is used.

The answer is no. We experiment with supplying readers with all supporting sessions. As shown in Fig. 1, even with this oracle evidence, temporal and multi-hop accuracy remain below single-hop accuracy for both readers. We refer to the difficulty that remains despite access to annotated support as the *retrieval–execution gap*:

annotated support is available $\not\Rightarrow$ the reader answers correctly.

These results motivate separating source-local interpretation from query-dependent computation, as we describe next.

2.2 Problem Setting and Staged Interpretation

We consider an agent that interacts with a user through conversations over time. At time t, its append-only archive and answering process are

$$\mathcal{A}_t = \{a_1, \ldots, a_t\}, \qquad \hat{y} = f_\theta(q, M(q, \mathcal{A}_t)), \tag{1}$$

where each session a_i contains its original text and timestamp t_i^{src}, M constructs a context for query q, and f_θ is the reader.

When a session is recorded, future questions are unknown, but some interpretations are already supported by its local context. Consider a source turn

$$c = \text{I started the new job yesterday.} \quad t^{\text{src}} = 10 \text{ May } 2023.$$

With the source timestamp t^{src} in its local context L_c, WRITE resolves "yesterday" to 9 May 2023, before any question arrives. Preserving this date retains the interpretation even if a later context omits the timestamp. A future question such as "How long after moving did I start the job?" determines which other event to select and which relation to compute. Memory must therefore distinguish interpretations that can be preserved in advance from query-dependent computations.

Binding-time analysis in partial evaluation (Nielson & Nielson, 1988) suggests organizing this distinction by input dependencies. A supported computation can run at WRITE when all its inputs are source-local; computations requiring the question or retrieved records remain at READ.

> interpret locally when context is available;
> compose across memories when the query is known.

This split determines when to compute, but leaves open when the resulting value is sufficiently supported to be supplied to the reader. Successful execution may still leave the question's required precision or evidence scope unresolved. An answer contract makes these requirements explicit through type, precision, and scope. Section 3 brings these two decisions together through persistent source interpretations, query-dependent execution, and selective disclosure.

3 Executable Memory for Long-Horizon Reasoning

3.1 Overview: A Neuro-Symbolic Memory Interface

BindMem implements the staged interface through WRITE, READ, and DISCLOSE. Let E_q denote the retrieved turns and $\mathcal{Z}_q = \bigcup_{c \in E_q} Z(c)$ their stored records. The three stages are

$$\begin{aligned} \text{WRITE}: \quad & Z(c) = I(c, L_c), \\ \text{READ}: \quad & (P_q, \chi_q) = \text{Compile}(q), \\ & r_q = \text{Exec}(P_q; E_q, \mathcal{Z}_q), \\ \text{DISCLOSE}: \quad & B_q = \mathcal{D}_{\chi_q}(r_q). \end{aligned}$$

WRITE uses the deterministic interpreter I to construct typed records $Z(c)$ from each source and its local context, retaining links to the supporting evidence. READ compiles the query into a typed program P_q and an answer contract χ_q specifying the required type, precision, and evidence scope. It then executes the program over E_q and $\mathcal{Z}_q$ to produce state r_q. DISCLOSE checks this state against χ_q and constructs an additional payload B_q containing a supported value with evidence, selected evidence alone, or nothing.

Let C_q^W denote the reader context containing the retrieved text E_q and textual annotations of its WRITE records $\mathcal{Z}_q$. The final reader receives

Table 1: Notation for BindMem.

Symbol	Meaning
c, L_c	Source turn and its local context.
E_q	Turns retrieved for query q.
$Z(c)$	Typed records stored for turn c.
$\mathcal{Z}_q$	Stored records attached to E_q.
P_q	Typed query program.
χ_q	Required answer type, precision, and evidence scope.
r_q	Execution state, including values, evidence, and unresolved conditions.
B_q	Additional payload selected by DISCLOSE.

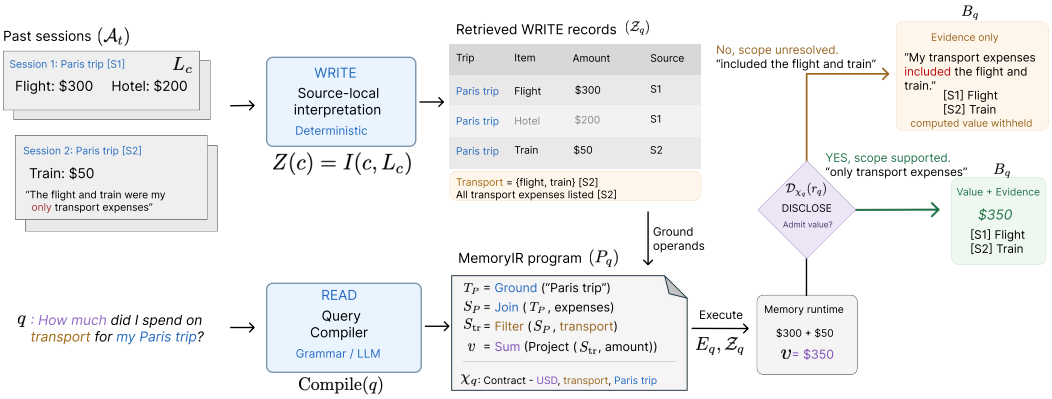


Figure 2: **Overview of BINDMEM.** WRITE preserves source-local interpretations, READ executes query-dependent programs, and DISCLOSE checks answer requirements before adding derived values. The upper branch replaces the exhaustive statement with "My transport expenses included the flight and train", leaving coverage unresolved. The reader retains the original retrieved text and WRITE annotations throughout.

$C_q^W \oplus B_q$, retaining access to the original retrieved text throughout. The answering process in Eq. (1) becomes

$$\widehat{y} = f_\theta(q, C_q^W \oplus B_q), \tag{2}$$

where $\oplus$ denotes concatenation of the rendered context blocks. The reader retains the original retrieved text and WRITE annotations regardless of the disclosure decision. Fig. 2 illustrates these stages through the Paris trip example introduced below. Appendix B describes the full operator set and execution semantics.

3.2 WRITE: INTERPRETING A SOURCE WHILE ITS CONTEXT IS AVAILABLE

Convert text to records. WRITE keeps the original source turn and augments it with a small set of provenance-linked typed records whose required inputs are available from its local context. A source interpreter maps locally supported predicate–argument structure and metadata into event, time, scalar, and scope records. Event records retain speaker metadata and normalized predicate/content terms; time records resolve supported relative expressions from source timestamps; scalar records associate values and units with their items; and scope records preserve source-supported complete sets with their category, members, and supporting span. Our implementation uses a fixed dependency parser with deterministic source constructors; unsupported or ambiguous fields remain unresolved. WRITE uses no ingestion-time LLM calls (Appendix A.1).

Example 1. Temporal interpretation. Suppose a speaker says, on 8 May 2023, "I moved into my new apartment yesterday." On 11 August 2023, the same speaker says, "I adopted a cat 90 days ago." WRITE resolves the dates to 7 May and 13 May, respectively, retaining which event each date describes and which timestamp anchored the interpretation. A future question can use these dates without resolving the relative phrases again.

Example 2. Preparing evidence for semantic linking. Consider two sessions containing "I signed up for a pottery class" and "The pottery class is held at Riverside Studio." WRITE retains the event terms and their sources. READ can later connect the registration evidence to the location evidence.

Example 3. Associations, membership, and coverage. Consider two sessions about a Paris trip as illustrated in Fig. 2:

> **Session 1:** For my Paris trip, the flight cost 300 USD, and the hotel cost 200 USD.
>
> **Session 2:** For my Paris trip, the train cost 50 USD. The flight and train were my only transport expenses.

WRITE interprets each session using its local context. Session 1 supplies the flight and hotel costs and their trip associations. Session 2 supplies the train cost and unambiguously identifies the flight and train costs as the total transport expenses. These interpretations retain their source evidence.

Typed records and source evidence. We write a stored record as

$$z = \langle \kappa, v, \alpha, \Pi, \xi \rangle,$$

where κ is the value type, v is the interpreted value, α contains semantic arguments and source-supported associations, Π identifies supporting spans and metadata, and ξ records precision. In the Paris example, Session 1 supplies the flight's 300 USD value and trip association for READ's grounding and computation. Session 2 supplies transport membership for READ's filter and completeness evidence for DISCLOSE. Each record retains its own supporting spans; READ must still establish whether the flight mentions refer to the same expense.

Dependency-guided materialization. A dependency pass over a fixed library of typed templates extracts source-local computations as recipes. WRITE evaluates these recipes at ingestion and stores their results for reuse at READ. The same constructors produce identical results at READ if all required inputs are available. Appendix A specifies the dependency analysis and deferred evaluation.

3.3 READ: Turning Questions into Executable Operations

Convert questions to programs. READ must express the query-dependent computation to perform over retrieved memories. We therefore introduce MemoryIR, a typed intermediate representation for computation over memory records. A MemoryIR program is a typed DAG that composes standard relational operations (Codd, 1970) with arithmetic and assertions. A constrained frontend compiles each question into a MemoryIR program and an answer contract, using deterministic paths for recognized requests and supported compositions and invoking a language-model proposer only after those paths decline. Appendix B.1 formalizes the program interface and execution semantics.

Grounding binds the entities and events mentioned in the query to compatible records in the retrieved evidence. Lexical matching is used first, with an encoder fallback for ambiguous cases; unresolved alternatives are kept rather than guessed. The resulting program is then validated and executed only over E_q and $\mathcal{Z}_q$.

Example 1 continued: temporal comparison. Returning to Example 1, suppose the later query asks, "How many days after I moved did I adopt a cat?" READ matches the two event descriptions to their stored records, checks the ordering of their resolved dates, and computes

$$7\ \text{May} < 13\ \text{May}, \qquad 13\ \text{May} - 7\ \text{May} = 6\ \text{days}.$$

The event descriptions determine the operands, while the stored date fields supply their values. The result retains both source anchors.

Example 2 continued: Connect two pieces of semantic evidence. The linking operation brings together source statements needed to answer a question. For "Where is the class I signed up for held?", the linking operation connects the registration and location clauses through the question-mentioned bridge "class". It supplies the two source statements with their session provenance. The reader interprets this linked evidence to answer "Riverside Studio". The linking operation supplies evidence using the shared term as a matching cue.

Execution state and propagation. Each MemoryIR operation returns an execution state containing its result, supporting evidence, and unresolved checks:

$$r_i = \langle \tau_i, \widehat{V}_i, \Pi_i, \varsigma_i, \mathcal{O}_i \rangle.$$

Here, τ_i is the output type, $\widehat{V}_i$ the possible values (points, intervals, finite alternatives, or unknown), Π_i the supporting sources, ς_i the scope supported by those sources, and $\mathcal{O}_i$ the unresolved requirements. DISCLOSE checks the final state r_q against the answer contract χ_q.

Consider the Paris trip example (Example 3) in Fig. 2, where the query asks, "How much did I spend on transport for my Paris trip?" READ first grounds the trip, follows its expense references across the retrieved records, filters for transport expenses, and sums their amounts. The program for READ

Step	Role in the Paris example	Result / supporting source
Ground	Bind "Paris trip" to its retrieved entity records.	$T_P = \{e_P\}$; trip mentions in S1 and S2.
Join	Follow trip references from T_P to the expense records.	$S_P = \{s_1, s_2, s_3\}$; trip associations in S1 and S2.
Filter	Apply the transport membership predicate m_{tr} to S_P.	$S_{\text{tr}} = \{s_1, s_2\}$; named members in S2.
Project	Read the amount field of each record in S_{tr}.	$A = \langle 300, 50 \rangle$ USD; amounts from S1 and S2.
Reduce (Sum)	Sum the unit-compatible amounts in A.	$v = 350$ USD, retaining the supporting sources.

Table 2: **Execution of the Paris trip query.** S1/S2 denote the two sessions. Grounding binds operands; DISCLOSE separately checks whether the computed sum supports the requested total.

can be described as a sequence of operations: let $\mathcal{Z}_q^E$ and $\mathcal{Z}_q^S$ denote the retrieved event/entity and scalar records.

$$\begin{aligned} T_P &= \text{Ground}(\text{“Paris trip”}, \mathcal{Z}_q^E), \\ S_P &= \text{Join}(T_P, \mathcal{Z}_q^S; \text{eventRef}), \\ S_{\text{tr}} &= \text{Filter}(S_P; m_{\text{tr}}), \\ v &= \text{Reduce}\big(\text{Project}(S_{\text{tr}}; \text{amount}); \text{Sum}\big). \end{aligned} \tag{3}$$

Table 2 shows how this grounded MEMORYIR program is executed. Here, e_P denotes the resolved trip identity and s_1, s_2, s_3 the flight, train, and hotel records; the trace assumes resolved trip and item references. The predicate m_{tr} uses the transport membership statement stored from Session 2. Its named items are resolved against the retrieved expense records before projection reads \$300 from Session 1 and \$50 from Session 2. With resolved references, the program (3) yields 350 USD as the travel expense based on the amount and membership evidence from both sessions.

During MEMORYIR execution, each operation propagates provenance, coverage, and unresolved requirements, where an unresolved item association remains relevant to the reduction, while arithmetic over interval-valued dates retains the resulting interval. This also shows that READ can use one result as the input to the next operation. Full primitive registry and transfer contracts appear in Appendices A.3 and B.3.

3.4 DISCLOSE: SELECTING WHAT TO SUPPLY TO THE READER

Check query requirements. DISCLOSE checks whether the result from valid executions with source evidence satisfies the answer contract's type, precision, and scope requirements. It outputs the derived result along with its supporting evidence only when all these requirements hold and no required checks remain unresolved; otherwise, it outputs the evidence alone.

A computed sum may not be the total amount. For the Paris trip, the flight and train amounts sum to \$350. Once their references are resolved, Session 2's "only" statement supports disclosing this value as the transport total. Replacing that statement with "My transport expenses **included** the flight and train" leaves the completeness unresolved: the value still answers a question about the two named expenses, but not necessarily the total.

Value admission. For a valid final state r_q, DISCLOSE derives Y_q from $\widehat{V}_q$ at the precision required by χ_q and checks coverage ς_q against its scope requirements (ScopeOK_q), admitting a value when

$$\text{Ready}_q = [|\mathcal{Y}_q| = 1] \wedge \text{ScopeOK}_q \wedge [\mathcal{O}_q = \varnothing]. \tag{4}$$

Answer uniqueness is evaluated at the requested precision: several possible dates can determine one month. Scope requires resolving all explicitly requested items or source evidence supporting completeness of the relevant set; scanning all retrieved records alone does not establish completeness.

Reader context. DISCLOSE supplies the value with sources and scope when Ready_q holds, and otherwise usable evidence alone. Malformed programs, contradictory required assertions, or no usable evidence yield no payload. Checks concern interpreted evidence only.

System	Benchmark score (%) ↑											
	LoCoMo		LME-S		LME-M		LifeMem		EgoMem		RHELM	
	Q	G	Q	G	Q	G	Q	G	Q	G	Q	G
HippoRAG	37.4	44.5	42.8	46.5	38.7	37.2	41.5	41.7	54.9	46.8	8.5	14.5
MemGPT/Letta	11.9	13.2	28.7	34.3	26.8	32.0	59.6	48.9	59.3	51.4	9.8	16.7
MemoryBank	19.7	23.2	22.2	26.6	22.8	24.8	48.0	46.3	44.7	44.8	0.9	1.0
TReMu	30.8	37.5	37.7	48.5	23.0	38.8	48.5	51.3	43.3	43.9	11.6	16.8
Mem0	34.1	40.2	35.1	40.5	27.5	34.7	47.6	48.1	47.3	46.5	8.6	10.8
MemGAS	38.5	44.3	45.6	57.6	32.6	36.2	60.5	58.9	58.3	54.9	16.3	17.5
A-MEM	35.1	42.2	43.6	52.2	41.1	43.0	54.1	48.8	56.7	52.5	10.7	16.7
BINDMEM	**46.5**	**50.4**	**54.8**	**66.1**	**45.1**	**55.1**	**63.2**	**65.1**	**63.3**	**62.2**	**19.6**	**28.1**

Table 3: **Cross-benchmark performance with one shared semantic interface.** LoCoMo reports the official overall F1/refusal score. LifeMem and EgoMem report multiple-choice exact-match accuracy; LongMemEval and RHELM report judge-based answer accuracy. Q/G denotes the Qwen2.5-7B/Gemma-3-12B instruction-tuned readers. LongMemEval scores are three-run means. Bold marks the best value in each reported comparison.

4 EXPERIMENTS

4.1 SETUP

Benchmarks. We evaluate BINDMEM on six benchmark variants: LoCoMo (Maharana et al., 2024), LongMemEval-S/M (Wu et al., 2025), LifeMem/EgoMem (Zheng et al., 2026), and RHELM (Zhang et al., 2026). Source interpretation, query compilation, grounding, execution, and disclosure rules are shared across benchmarks. Dataset-specific retrieval configurations and evaluation details are provided in Appendix D.

Baselines and readers. We compare BINDMEM against HippoRAG (Gutiérrez et al., 2024), TReMu (Ge et al., 2025), MemGPT/Letta (Packer et al., 2023), MemoryBank (Zhong et al., 2024), Mem0 (Chhikara et al., 2025), MemGAS (Xu et al., 2026), and A-MEM (Xu et al., 2025). The readers are Qwen2.5-7B-Instruct (Yang et al., 2024) and Gemma-3-12B-Instruct (Team et al., 2025), abbreviated Q/G, without method-specific fine-tuning. For these two readers, systems within each comparison share the final-reader prompt, greedy decoding, and scoring procedure. Matched analyses additionally fix retrieved identities and ordering.

Metrics. LoCoMo uses factual F1 and adversarial refusal accuracy for end-to-end evaluation. LongMemEval uses the official answer-checking templates with GPT-4.1. LifeMem and EgoMem use EM accuracy for multiple-choice questions. RHELM uses GPT-4.1 (OpenAI, 2025) judging.

4.2 MAIN RESULTS AND ANALYSES

Cross-benchmark performance. Table 3 compares BINDMEM with seven memory baselines across six benchmark variants. Using the same source interpreter, query compiler, and execution and disclosure rules, BINDMEM achieves the highest overall score in all 12 reported benchmark–reader comparisons. Gains over the strongest baseline reach 8.0/5.9 points on LoCoMo, 9.2/8.5 points on LongMemEval-S, and 4.0/12.1 points on LongMemEval-M for Qwen/Gemma, with improvements of up to 10.6 points on LifeMem, EgoMem, and RHELM. This consistency supports the applicability of the shared interface across the evaluated settings, while the variation in gains motivates examining which components contribute. The following analyses isolate stored interpretations, read-time execution, and value disclosure under fixed retrieval.

Contributions of memory augmentation stages. Table 4 holds retrieved turn identities and ordering fixed while progressively adding stored interpretations (Write), execution-selected evidence (Evidence-only), and admitted values (Full) to the raw-text Base. Each stage improves benchmark scores across both benchmarks and readers: Write adds 2.4–6.4 points, Evidence-only adds 1.0–8.4, and Full adds another 1.6–3.8. The Full–Evidence-only comparison fixes programs, grounding, execution, and evidence rendering, isolating the benefit of supplying admitted values beyond their

Benchmark	Reader	Base	Write	Evidence only	Full
LoCoMo	Qwen	35.2	41.6	42.9	**46.5**
	Gemma	45.4	47.8	48.8	**50.4**
LME-S	Qwen	40.8	45.1	51.0	**54.8**
	Gemma	50.1	55.1	63.5	**66.1**

Table 4: **Matched stage contributions.** Benchmark score (%) under fixed retrieved evidence: LoCoMo reports the official overall F1/refusal score and LME-S judge-based answer accuracy. All four conditions in each row use the same question set, including questions receiving no execution payload.

Reader	Interface	All questions Acc. (%) $n = 500$	Trace audit Correct / attempted $n = 50$
Qwen	Fixed-surface	51.2	–
	Compositional	54.8	38/50 (76.0%)
Gemma	Fixed-surface	61.8	–
	Compositional	66.1	38/50 (76.0%)

Table 5: **Interface comparison and trace correctness on LME-S.** Accuracy averages three runs of the reader and judge on all 500 questions, with upstream artifacts fixed within each condition. Both 38/50 entries refer to the same one-time audit of 50 questions accepted by the compositional recognizer.

Benchmark	Reader	Without checks	Full
LoCoMo	Qwen	43.2	46.5
	Gemma	47.9	50.4
LME-S	Qwen	50.6	54.8
	Gemma	63.1	66.1

Table 6: Effect of scope and obligation checks on the final-answer score (%). LoCoMo reports the overall F1/refusal score and LME-S judge-based answer accuracy. Appendix F.2 reports paired per-question score changes.

Benchmark	Slice	GPT-5 Base	GPT-5 + BINDMEM	Δ (pp)
LoCoMo	Overall	88.5	92.7	**+4.2**
LME-S	Overall	77.2	80.8	**+3.6**
	Temporal reasoning	70.2	73.7	**+3.5**
	Multi-session	55.6	60.2	**+4.6**

Table 7: **Frontier-reader results with BINDMEM (%).** GPT-5 serves as both the program proposer and final reader. Retrieval, WRITE records, execution rules, and disclosure policy are unchanged. Scoring follows the same protocol as Table 3.

supporting evidence. Overall gains over Base are 11.3/5.0 points on LoCoMo and 14.0/16.0 on LME-S for Qwen/Gemma, including questions receiving no execution payload. These results show complementary benefits from interpreting sources, selecting evidence, and supplying supported computations under fixed retrieval.

Dependent composition. We compare fixed-surface and compositional interfaces under identical retrieved evidence, WRITE records, reader, and disclosure policy. The compositional interface supports dependent execution in which intermediate results determine later operands or input collections; unsupported questions retain the original reader context. Table 5 shows overall accuracy gains of 3.6/4.3 percentage points on all 500 LME-S questions for Qwen/Gemma. Separately, a single audit of 50 programs shared by both readers finds 38 traces (76.0%) with correct operations, grounding, dependencies, and computed values. The audited questions are the 50 questions accepted by the compositional recognizer from 63 candidates selected by a regex proxy.

Effect of admission checks. We ablate scope and obligation checks, holding compiled programs, grounding, execution traces, and selected evidence fixed. Both conditions retain type validity and answer uniqueness at the requested precision, excluding malformed programs, failed executions, and contradictory required assertions. Table 6 shows that these checks improve benchmark scores by 3.3/2.5 points on LoCoMo and 4.2/3.0 on LME-S for Qwen/Gemma. These gains indicate that an executable, unique result benefits from checking whether its evidence supports the requested answer.

Evaluation with a frontier reader. With GPT-5 used for both program proposal and final answering, BINDMEM improves the overall score by 4.2 points on LoCoMo and 3.6 points on LME-S (Table 7). These results show that the interface remains beneficial with a stronger reader under fixed retrieval.

Comparison with alternative reasoning interfaces. Against LLM-generated Python over the same WRITE records, BINDMEM improves overall accuracy by 11.0/6.1 points for Qwen/Gemma and attains the highest temporal and non-temporal multi-evidence accuracy among the interfaces in Table 8, supporting the benefit of MEMORYIR execution and DISCLOSE beyond stored interpretations alone.

Source interpretation	Execution	All	Temporal	Non-temp. multi-evidence	Extra LLM calls/q
Qwen					
None	Reader only	40.8	41.1	36.8	0
Read-time	LLM-generated Python	47.8	45.9	39.1	2.0
WRITE	LLM-generated Python	43.8	40.6	33.8	1.0
	BINDMEM	54.8	50.1	39.8	0.73
Gemma					
None	Reader only	50.1	50.7	35.1	0
Read-time	LLM-generated Python	56.2	51.1	39.8	2.0
WRITE	LLM-generated Python	60.0	55.6	42.1	1.0
	BINDMEM	66.1	58.9	46.1	0.73

Table 8: **Alternative reasoning interfaces on LME-S.** Accuracy (%) under fixed retrieval. WRITE rows share the same stored records; BINDMEM uses MEMORYIR execution and DISCLOSE.

Appendix F.1 gives the Python configuration and a breakdown by execution status, where successful execution does not ensure a correct final answer.

Construction and query cost. BINDMEM builds its store in 8.3 s with no ingestion LLM calls, versus 102–170 s and 239–245K tokens for A-MEM and HippoRAG, while supplying a smaller reader context. Timings reflect each system's default concurrency (Appendix E.2).

5 RELATED WORK

Memory organization and retrieval. Long-horizon agent memory systems organize interaction histories into persistent stores, summaries, graphs, or evolving memories (Wang et al., 2023; Packer et al., 2023; Zhong et al., 2024; Gutiérrez et al., 2024; Chhikara et al., 2025; Xu et al., 2025). Recent approaches use multi-granularity representations, temporal graphs and hierarchies, and time-aware selection to improve consolidation and retrieval (Xu et al., 2026; Rasmussen et al., 2025; Li et al., 2026; Du et al., 2026). Our work addresses a complementary challenge: interpreting and combining evidence once retrieved. Without changing the underlying retrieval mechanism, BINDMEM preserves source-local interpretations at WRITE and supports query-dependent execution and answer-requirement checks at READ and DISCLOSE.

Programmatic and neuro-symbolic reasoning. PAL and Program of Thoughts delegate computation to executable programs (Gao et al., 2023; Chen et al., 2023); BINDER and Faithful CoT couple language interpretation with symbolic execution (Cheng et al., 2023; Lyu et al., 2023). Logic-LM, SatLM, and LINC use LLM-generated formal representations with external solvers or theorem provers (Pan et al., 2023; Ye et al., 2023; Olausson et al., 2023). For agent memory, TReMu combines time-aware memorization with Python-based temporal reasoning (Ge et al., 2025). LOTUS expresses natural-language data processing through composable semantic operators and optimizes their execution with accuracy guarantees relative to a reference algorithm (Patel et al., 2024). BINDMEM focuses on persistent source-local interpretations and query-dependent execution over memory, propagating provenance and unresolved conditions before checking whether derived values meet the question's required scope and precision.

6 CONCLUSION

We introduced BINDMEM to address the reasoning that remains after memory retrieval. It preserves source-local interpretations, executes query-dependent programs, and checks derived results against explicit answer requirements before disclosure. Across six benchmark variants, BINDMEM improves overall performance over the evaluated memory baselines, while controlled studies isolate the benefits of stored interpretations, query-dependent composition, and value disclosure. These results support treating computation and evidence sufficiency as explicit components of long-horizon memory interfaces. Future work will explore probabilistic semantic models for interpretation and grounding while retaining explicit execution and disclosure checks.

Reproducibility Statement

The anonymized supplementary material provides code, the source constructors, the dependent-composition library, the proposer prompt, and closed DSL schema (Appendix D.4), runtime configurations, and evaluation scripts.

AI use statement

We used generative AI tools to assist with debugging existing code and correcting grammar in the manuscript. The authors reviewed all AI-assisted changes, checking code modifications for correctness and text edits for preservation of the intended meaning. We take responsibility for the final content of this work, including all text and code modified with the assistance of generative AI.

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 147–160. ACM, 1999. doi: 10.1145/292540.292555. URL https://doi.org/10.1145/292540.292555.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=lH1PV42cbF.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *ECAI 2025 – 28th European Conference on Artificial Intelligence*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pp. 2993–3000. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://doi.org/10.3233/FAIA251160.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Yiming Du, Baojun Wang, Yifan Xiang, Zhaowei Wang, Wenyu Huang, Boyang Xue, Bin Liang, Xingshan Zeng, Fei Mi, Haoli Bai, Lifeng Shang, Jeff Z. Pan, Yuxin Jiang, and Kam-Fai Wong. Memory-T1: Reinforcement learning for temporal reasoning in multi-session agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/6010a57158b77359f5f531d9af69181c-Abstract-Conference.html.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Yubin Ge, Salvatore Romeo, Jason Cai, Raphael Shu, Yassine Benajiba, Monica Sunkara, and Yi Zhang. Tremu: Towards neuro-symbolic temporal reasoning for llm-agents with memory in multi-session dialogues. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 18974–18988, 2025.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 59532–59569, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/6ddc001d07ca4f319af96a3024f6dbd1-Abstract-Conference.html.

Kai Li, Xuanqing Yu, Ziyi Ni, Yi Zeng, Yao Xu, Zheqing Zhang, Xin Li, Jitao Sang, Xiaogang Duan, Xuelei Wang, Chengbao Liu, and Jie Tan. TiMem: Temporal-hierarchical memory consolidation for long-horizon conversational agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 21700–21720, 2026. doi: 10.18653/v1/2026.findings-acl.1091. URL https://aclanthology.org/2026.findings-acl.1091/.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47–57. ACM, 1988. doi: 10.1145/73560.73564. URL https://doi.org/10.1145/73560.73564.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 305–329. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.ijcnlp-main.20. URL https://aclanthology.org/2023.ijcnlp-main.20/.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed lambda-calculus. *Science of Computer Programming*, 10(2):139–176, 1988. doi: 10.1016/0167-6423(88)90025-1. URL https://doi.org/10.1016/0167-6423(88)90025-1.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5153–5176. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.313. URL https://aclanthology.org/2023.emnlp-main.313/.

OpenAI. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/, April 2025.

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. URL https://arxiv.org/abs/2310.08560.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.248. URL https://aclanthology.org/2023.findings-emnlp.248/.

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators: a declarative model for rich, ai-based data processing. *arXiv preprint arXiv:2407.11418*, 2024.

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. URL https://arxiv.org/abs/2501.13956.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025. URL https://arxiv.org/abs/2503.19786.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting language models with long-term memory. In *Advances in Neural Information*

Processing Systems, volume 36, pp. 74530–74543, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/ebd82705f44793b6f9ade5a669d0f0bf-Abstract-Conference.html.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/d813d324dbf0598bbdc9c8e79740ed01-Abstract-Conference.html.

Derong Xu, Yi Wen, Pengyue Jia, Yingyi Zhang, Wenlin Zhang, Yichao Wang, Huifeng Guo, Ruiming Tang, Xiangyu Zhao, Enhong Chen, and Tong Xu. From single to multi-granularity: Toward long-term memory association and selection of conversational agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/93e444b4a1a64815631c3c74977ee200-Abstract-Conference.html.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 20004–20031, 2025. doi: 10.52202/085713-0593. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. URL https://arxiv.org/abs/2412.15115.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. SatLM: Satisfiability-aided language models using declarative prompting. In *Advances in Neural Information Processing Systems*, volume 36, pp. 45548–45580, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/8e9c7d4a48bdac81a58f983a64aaf42b-Abstract-Conference.html.

Han Zhang, Zihao Tang, Xin Yu, Xiao Liu, Yeyun Gong, Haizhen Huang, Yan Lu, Weiwei Deng, Feng Sun, Qi Zhang, and Hanfang Yang. Beyond static dialogues: Benchmarking realistic, heterogeneous, and evolving long-term memory. *arXiv preprint arXiv:2605.31086*, 2026. URL https://arxiv.org/abs/2605.31086.

Jianjie Zheng, Zhichen Liu, Zhanyu Shen, Jingxiang Qu, Guanhua Chen, Yile Wang, Yang Xu, Yang Liu, and Sijie Cheng. Evaluating memory capability in continuous lifelog scenario. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 7063–7089. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.351. URL https://aclanthology.org/2026.findings-acl.351/.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://doi.org/10.1609/aaai.v38i17.29946.

APPENDIX INDEX

Appendix A. Source Interpretation and Dependency-Guided Materialization p. 13
Stored records, temporal interpretation, and WRITE/READ staging.

Appendix B. Grounded Compositional Runtime p. 15
MemoryIR programs, answer contracts, grounding, and execution-state propagation.

Appendix C. Disclosure and Worked Execution p. 19
Precision and scope checks, worked traces, fallback, and matched controls.

Appendix D. Evaluation Protocol and Implementation Details p. 21
Datasets, scoring, source parsing, program interfaces, and proposer configuration.

Appendix E. Computational Cost p. 24
LLM calls, ingestion and query time, storage, and amortized call budgets.

Appendix F. Additional Results p. 27
Python baseline failure analysis and paired score changes under disclosure checks.

Appendix G. Compiler and Execution Reliability Details p. 28
Question-reformulation diagnostics and human operator-intent audits.

Appendix H. Transfer Across Retrieval Backends p. 30
Matched temporal comparisons under dense, HippoRAG, and A-MEM retrieval.

A SOURCE INTERPRETATION AND DEPENDENCY-GUIDED MATERIALIZATION

The appendix specifies the records and operator contracts used by the memory interface. WRITE uses a fixed syntactic parser followed by deterministic typed extraction rules to produce source-local interpretations; MemoryIR composes computations over those interpretations; and DISCLOSE checks the final execution state against the question's answer requirements. We refer to the typed extraction functions below as source constructors. Input dependencies determine which interpretations can be materialized before a question arrives.

A.1 RECORDS AND LOCAL CONTEXT

The archive stores each original turn together with a sparse collection of records $z = \langle \kappa, v, \alpha, \Pi, \xi \rangle$. The kind κ determines the value type; v contains the interpreted value of that type; α contains semantic arguments and source-supported associations; Π identifies supporting spans and metadata; and ξ records precision. Missing fields are unresolved rather than filled with a guessed value. A source timestamp is an interpretation anchor and need not be the date of the event described by the turn.

Record	Content	Interpretation constraint
Event	Available speaker metadata, normalized predicate/content terms, event arguments, optional time and identity	Identity follows explicit mentions or a uniquely supported local association.
Time	Date interval, precision, event reference, source anchor	A point date has equal endpoints; absent anchors remain unresolved.
Scalar	Number, unit, attribute, item identity, optional event reference	The number of records is distinct from the quantity stated in a record.
Scope descriptor	Domain, predicate, time range, named members, and the supporting completeness span	Preserves a source-supported complete enumeration. References to records in other sessions are resolved at READ before the descriptor supports value admission.

Table 9: **Stored interpretations.** Every interpreted field retains its own evidence. Scope describes what the source reports, not all real-world events.

For a turn c, L_c is the declared context available when its records are constructed: its timestamp, available speaker metadata, and any neighboring turns used for local interpretation. It does not

include future questions or unrestricted archive-wide search. When an association uses two turns, both are included in its provenance. A cached record may retain those provenance references even if a later retrieval omits one supporting turn; this does not authorize a new read from outside the retrieved evidence. Source interpretation uses a fixed dependency parser (spaCy `en_core_web_sm`) followed by deterministic typed extraction rules. The parser supplies predicate–argument structure, nominal heads, modifiers, coordination, and possessive relations; the extraction rules combine these relations with lexical cues and local metadata to construct event, time, scalar, reference, and scope records. Temporal normalization additionally uses the source timestamp. No ingestion-time LLM is used, and unsupported or ambiguous fields remain unresolved.

A source-local membership statement can name an item whose amount is recorded in another session. WRITE preserves the item mention, its stated domain and membership, and the supporting source span. It does not require the amount to appear in the same source. Any correspondence between that mention and an expense record in another retrieved session is resolved at READ.

For example, the transport enumeration for the Paris trip in Session 2 names the flight and train. The flight amount remains supported by Session 1. The stored enumeration supplies membership and coverage evidence, while the cross-session correspondence is a separate requirement for using that evidence in an aggregate.

A.2 Temporal Interpretation

Temporal cues are resolved within their clauses using the anchors licensed by those clauses. Relative dates use the source timestamp; a reference time introduced by the question is a READ input. Intervals preserve the precision of the source. An unresolved day is not replaced by an arbitrary day merely to make a later calculation executable. For event intervals $A = [a_-, a_+]$ and $B = [b_-, b_+]$, signed subtraction has the interval enclosure

$$B - A = [b_- - a_+,\ b_+ - a_-].$$

An operation requiring that B occur strictly after A additionally checks $a_+ < b_-$. This ordering requirement belongs to the requested relation; signed subtraction alone does not require positive ordering. An overlapping interval can leave the relation unresolved while still supplying useful source evidence. Calendar conventions, parser configuration, and supported extraction patterns are fixed across WRITE and deferred READ.

A.3 Shared Registry and Stage Eligibility

MemoryIR combines standard relational operations (Codd, 1970) with arithmetic and assertions, while propagating provenance, coverage, and unresolved requirements. Its registry includes source constructors and ten computational primitives: Scan, Filter, Project, Join, Sort, Select, Distinct, Reduce, Arith, and Assert. Load is the stored-reference form introduced by staging; it is not an additional collection primitive. Each entry specifies argument and result types, typed parameters, input dependencies, deterministic evaluation, and transfer of provenance, coverage, and unresolved requirements. The shared execution contracts appear in Appendix B.3. Source constructors supply the interpreted records on which these operators act.

Let $D(e)$ be an expression's dependencies, $D_{\mathrm{direct}}(o, \eta)$ the inputs read directly by an operation and its parameters, and $\mathcal{D}_W$ the declared source-text, timestamp, and local-metadata inputs. Dependency propagation follows

$$D\big(o(e_1, \ldots, e_k; \eta)\big) = D_{\mathrm{direct}}(o, \eta) \cup \bigcup_{j=1}^{k} D(e_j). \tag{5}$$

A pure expression is WRITE-eligible when $D(e) \subseteq \mathcal{D}_W$ and it has no free query or retrieved-collection binder. This is a standard conservative dependency analysis (Lucassen & Gifford, 1988; Abadi et al., 1999). Dependencies include parameters and control inputs required by the expression itself. A query filter over an already interpretable source collection does not make that collection's pure constructor query-dependent. Conversely, a constructor that requires a query-selected anchor cannot be moved to WRITE. Constructors must expose all environment reads, including reference time.

Expression	Dependency	Staging decision
Parse and normalize source text	SourceText	WRITE-eligible
Resolve a source-relative date	SourceText, SourceTime	WRITE-eligible
Extract quantity / local reference	SourceText, declared SourceMeta	WRITE-eligible when locally supported
Extract a complete enumeration	SourceText, declared local context	WRITE-eligible; partial parsing stays unresolved
Filter using a question predicate	Query, collection inputs	READ
Join selected records and sum	Selected parent results	READ on the query path
Compare with the question time	Query, event interpretation	READ

Table 10: **Dependencies determine availability.** Operation names alone do not determine the stage. Query and retrieved-collection binders cannot be treated as source-local inputs.

A.4 Split, Cache, and Deferred Evaluation

Configuration operates on parametric templates, before ingestion and before the future query is known. It performs the following pass:

1. Validate expression types, parameter dependencies, free variables, and purity. Reject invalid templates; constructor failures during evaluation remain unresolved interpretations.
2. Find maximal source-local subexpressions whose dependencies lie in $\mathcal{D}_W$ and that have no free query or collection variable.
3. Merge identical expressions into a recipe bank $\mathcal{W}$. Replace each occurrence by the typed reference $\mathrm{Load}(k_e, c)$ to obtain the residual templates.
4. At ingestion, evaluate each applicable recipe for (c, L_c) and store its successful fields with provenance. At READ, instantiate and compose residual expressions with registered primitives.

A recipe key identifies the expression, its constant parameters, and interpreter version. A stored evaluation additionally identifies its source and local-context versions. Equal text with a different timestamp or speaker is not a cache hit. Appending unrelated archive content does not change an interpretation whose declared inputs are unchanged; this property does not apply to query aggregates or latest-value selection.

A failed or absent stored reference can be evaluated at READ using the same constructor only when every required input is available in E_q and its attached records. Otherwise the reference stays unresolved. Query programs cannot recover an omitted source anchor merely by requesting a new interpretation that was not materialized.

Derivation example. For an event constructor I_E, a template $\mathrm{Filter}(I_E(c, L_c); \phi_q)$ splits into the recipe $I_E(c, L_c)$ and residual $\mathrm{Filter}(\mathrm{Load}(k_E, c); \phi_q)$. The same loaded event interpretation can be projected to its date, compared with another event, or joined with its quantities. These query-dependent operations remain in the residual program. The template supplies supported source semantics; it does not predict a future question.

B Grounded Compositional Runtime

B.1 Program Interface and Answer Contract

Definition 1 (MemoryIR program). A MemoryIR program for query q is a typed ordered DAG

$$P_q = \{(o_i, J_i, \eta_i)\}_{i=1}^{m},$$

with a designated output node, where o_i is a registered primitive, $J_i \subseteq \{1, \ldots, i-1\}$ identifies its parent nodes, and η_i contains its typed parameters. Source collections enter through Scan or stored-reference forms, and parent results may supply operands or collections to later nodes. Each node produces an execution state

$$r_i = \langle \tau_i, \widehat{V}_i, \Pi_i, \varsigma_i, O_i \rangle,$$

containing its output type, represented value, provenance, supported coverage, and unresolved requirements. Appendix A.3 defines the primitive registry and staging rules, and Appendix B.3 gives the corresponding transfer semantics.

Grounding. Before execution, grounding associates operand descriptions with type-compatible records or supporting spans in the retrieved evidence. Compilation determines the requested computation; grounding determines which source records supply its inputs. A well-typed program does not by itself establish that these bindings are correct. Grounding and execution access only the retrieved turns E_q and their attached records Z_q, with ambiguity and unresolved conditions handled as specified in Appendix B.2.

Answer contract. The answer contract is $\chi_q = (\tau_q, g_q, \sigma_q)$, where τ_q is the requested answer type, g_q maps represented values to the requested answer precision, and σ_q specifies the required evidence scope. The type and precision state whether the question requests a date, month, duration, quantity, or collection; the scope names the requested domain, predicate, and time range when applicable. The compiler derives this contract from the question. The contract specifies what must be established; it does not supply evidence of exhaustiveness or declare a collection complete. An unsupported or unresolved required contract field prevents value admission.

Compilation paths. The deterministic grammar first attempts to map the question to a typed operation specification containing an operation and its operands. In hybrid mode, questions declined by the grammar can be handled by a compositional recognizer, which checks for a supported combination of relation and output computation and instantiates the corresponding MemoryIR composition. The question supplies its operands and predicates.

An enabled language-model proposal path uses the serving reader to propose a specification within the supported surface-operation vocabulary or nested calls in a closed DSL. The path is gated on a deterministic decline: it is invoked only for questions that neither the grammar nor the compositional recognizer accepts, so a grammar-handled question issues no proposal call. Proposals are parsed into an abstract syntax tree and checked against the registered types and operations before execution. The proposer supplies a candidate specification; the deterministic parser, builder, and validator determine whether it is admissible. Unknown operations, invalid references, and type-incompatible calls are rejected. Appendix D.4 reproduces the prompt, the DSL schema, and the generation settings.

The write interpreter, grammar, composition library, proposer schema and prompt, deterministic builder, grounding, and runtime semantics are shared across benchmarks. Benchmark-specific loading and scoring do not add semantic operators.

Composition over the shared operators. An intermediate result can parameterize a later operator or determine its input collection. Let A be the grounded anchor-event interpretations, E the candidate event collection, and $\mathcal{Z}_q^S$ the retrieved scalar records. A temporally restricted total has the operator-level description

$$\begin{aligned} a &= \mathrm{Project}(A; \mathrm{time}), \\ T &= \mathrm{Filter}(E; \phi_\rho(\mathrm{time}, a)), \\ T_p &= \mathrm{Filter}(T; p_e), \\ S &= \mathrm{Join}(T_p, \mathcal{Z}_q^S; \mathrm{event_ref}), \\ U &= \mathrm{Filter}(S; p_s), \\ v &= \mathrm{Reduce}(U; \mathrm{sum}, \mathrm{value}). \end{aligned}$$

Here ρ specifies the supported temporal relation, ϕ_ρ is its temporal predicate, and p_e and p_s are the question-specified event and scalar predicates. The join follows source-supported event–scalar associations. The value-field parameter specifies the scalar field read by the reduction; unit, identity, and missing-value requirements remain part of its result state. The anchor time determines which events reach the join, and the resulting collection determines which scalar records reach the reduction. Each operator propagates the shared state defined in Appendix B.3.

Evaluated dependent compositions. MemoryIR permits dependent programs in which intermediate results determine later operands or input collections. For the experiments, the compositional frontend instantiates two temporal-relation families (after/since and before/until) with five output computations

Composition	Operands	Computation sequence	Output
`elapsed_after`	anchor, event	$a \leftarrow$ `select_time`(anchor); $b \leftarrow$ `select_time`(event); $v \leftarrow$ `elapsed`(a, b)	Duration
`total_after`	anchor, p_e, p_s	$a \leftarrow$ `select_time`(anchor); $T \leftarrow$ `filter_after`(E, a); $T_p \leftarrow$ `filter_pred`(T, p_e); $S \leftarrow$ `join_scalars`(T_p); $U \leftarrow$ `filter_scalar_pred`(S, p_s); $v \leftarrow$ `sum`(U)	Number
`count_after`	anchor, p_e	$T_p \leftarrow \cdots$; $D \leftarrow$ `distinct`(T_p); $v \leftarrow$ `count`(D)	Number
`argmax_after`	anchor, p_e, p_s	$U \leftarrow \cdots$; $v \leftarrow$ `argmax`(U)	Entity
`argmin_after`	anchor, p_e, p_s	$U \leftarrow \cdots$; $v \leftarrow$ `argmin`(U)	Entity
`*_before`	(as above)	Replace `filter_after` with `filter_before` where present; for `elapsed`, change the direction of the time difference.	(as above)

Table 11: **Evaluated dependent-composition instantiations.** Each program is lowered to the shared MemoryIR primitive registry; the table lists the compositions instantiated in our experiments.

(elapsed time, sum, count, argmax, and argmin). These evaluated compositions are expressed entirely through the shared MemoryIR primitives and execution contracts. Table 11 summarizes the instantiated programs.

The names in Table 11 describe the role of each computation: `select_time` projects a grounded event's time; the temporal, event, and scalar filters instantiate FILTER; `join_scalars` uses JOIN; and `sum` and `count` instantiate REDUCE. `distinct` uses DISTINCT, elapsed-time computations use ARITH with the requested ordering requirements, and extrema use the ordering and selection operators. These names do not extend the primitive registry. Fixed-surface lowerings and dependent compositions use the same runtime contracts. Appendix G evaluates fixed-surface operation selection; the compositional trace audit is reported separately in Table 5.

Validation and execution limits. Validation checks operation names, argument types, permitted field access, acyclicity, and literal licensing. Initial query literals must be supported by the question; grounding establishes source bindings, and an intermediate result can become an operand without appearing verbatim in the question. Proposer-supplied source identifiers cannot authorize access to arbitrary archive content. The evaluated frontend constructs MemoryIR programs from the surface-operation lowerings in Table 14 and the dependent compositions in Table 11. Grounding and execution retain every supported alternative rather than truncating to a fixed candidate count. If the supported representation cannot preserve the required alternatives, the result remains unresolved rather than discarding alternatives to manufacture a unique answer. Valid programs proceed to execution, and the resulting values, provenance, and unresolved conditions are passed to DISCLOSE.

B.2 GROUNDING AND SOURCE ACCESS

For a phrase u and a type-compatible record z, lexical ranking uses normalized stem recall

$$s_{\text{lex}}(u, z) = \frac{|S(u) \cap S(z)|}{|S(u)|}, \qquad S(u) \neq \varnothing.$$

Here $S(\cdot)$ is a set of normalized content-word stems: $S(u)$ is built from the operand phrase, and $S(z)$ from the record's predicate stems, its scalar or item noun, and its source span. Normalization lowercases, strips stopwords, and applies the same stemmer used by WRITE, so a record and an operand phrase are compared in one vocabulary. Speaker, session, timestamp, and provenance fields are not used in lexical scoring; they identify a record rather than describe it. The encoder fallback operates on source clauses and also requires content-word overlap. It uses the same `all-MiniLM-L6-v2` encoder loaded for retrieval (Appendix D), so grounding introduces no additional encoder. Lexical stem recall is the primary scorer and the encoder is consulted only for phrases it leaves unseparated. The admission margins against the strongest competing source turn are 0.25 for the lexical score and 0.05 for the encoder score. A separated match gives a provisional binding. Otherwise, all supported alternatives remain candidates and are propagated through execution without a fixed candidate cap. If the supported representation cannot preserve the required alternatives, the result remains unresolved. These scores rank interpretations; they do not certify natural-language correctness.

Grounding resolves initial source references. During execution, a parent result can parameterize a later filter or determine a JOIN input. Both operations use only retrieved turns and their attached records. Source IDs in provenance identify evidence already used by the interpretation; they are not an instruction to fetch additional archive content.

JOIN follows explicit event/entity references or supported unambiguous local associations. Sharing a city, date, or lexical stem does not establish event identity. Two equal amounts can be two distinct expenses; conversely, one expense reached through two join paths is counted once. DISTINCT merges only identities licensed by the source interpretation. An unresolved identity remains relevant to exact counts and aggregates.

Membership evidence and amount evidence may originate in different retrieved sessions. A membership statement supports filtering only after its named items have been associated with the corresponding expense records. These associations retain the evidence from both sources; an unresolved correspondence remains an execution condition.

In the Paris trip example, the flight mention in Session 2 must be associated with the flight amount record in Session 1.

Source-text bridges and reference joins. The class example constructs a source-evidence block from the registration and location clauses using the question-mentioned bridge term. It leaves the location answer to the reader. This matching relation does not assert entity identity. In contrast, the Paris trip trace uses resolved trip references to collect expense records from two sessions. It then applies the transport membership statement from Session 2 to the corresponding expense records, including the flight amount from Session 1. Both paths preserve source provenance, but only the latter produces the typed expense collection used in Table 2.

Grounding notation in the main text. The Grounding row in Table 2 introduces the binding

$$T_P = \mathrm{Ground}(\text{Paris trip}, \mathcal{Z}_q^E),$$

where $\mathcal{Z}_q^E$ contains the retrieved event/entity records. This notation introduces no additional collection primitive. The resolved trace has $T_P = \{e_P\}$. With competing interpretations, it carries binding alternatives, not a request to pool distinct trips into one aggregate. Each supported interpretation must be propagated conservatively or remain unresolved. The predicate m_{tr} is evaluated from source-supported membership; the query supplies the requested category, not a list of assumed members.

B.3 Transfer of Values, Coverage, and Conditions

Each node returns $r_i = \langle \tau_i, \widehat{V}_i, \Pi_i, \varsigma_i, \mathcal{O}_i \rangle$. An abstract value represents points, intervals, finite alternatives, or unknown. The coverage state ς_i carries the domain and available coverage evidence; $\mathcal{O}_i$ tracks unresolved requirements. This state records the coverage the execution supports, and is distinct from the requested scope σ_q in the answer contract. Represented alternatives may agree on the requested answer without a unique input binding.

The transfer rules depend on an operator's typed inputs and parameters. An input retains the same interpretation when used by a later computation. This contract allows operators to be composed while

carrying the evidence and unresolved conditions relevant to the result. FILTER denotes predicate-based selection, whereas SELECT denotes order- or extremum-based selection. Equal projected values do not establish duplicate identity; duplicate elimination requires source-supported identity.

Operation	Value transfer	Evidence / condition transfer
SCAN / LOAD	Read a collection or stored interpretation	Retain source scope, precision, and provenance.
FILTER	Preserve true, false, and undecided membership	Do not silently exclude a possible required member.
PROJECT	Extract a typed field	Retain field evidence, identity, and missing-field conditions.
JOIN	Combine source-licensed associations	Carry association ambiguity and the coverage of both domains.
DISTINCT	Merge licensed duplicate identities	Keep undecided identity relevant to counts and reductions.
SORT	Order a collection by a declared key	Preserve ties and uncertain order.
SELECT	Select by a declared order or extremum	Retain unresolved ties or order; latest/current selection also requires query-scope coverage.
REDUCE	Aggregate possible members	Track units, missing members, and duplicate identity.
ARITH	Propagate intervals or finite alternatives	Retain the preconditions of the requested relation.
ASSERT	Return true, false, or unresolved	True discharges a condition; false invalidates; unresolved remains pending.

Table 12: **Compositional execution contract.** If a supported transfer cannot conservatively represent the alternatives, it returns unknown.

Provenance includes evidence for values, membership decisions, associations, and discharged assertions. FILTER restricts coverage to its predicate; JOIN carries coverage and association requirements for its inputs; REDUCE inherits conditions that can affect the requested aggregate. A precise arithmetic result does not by itself establish coverage of the question's domain.

Conditional conservativeness. If extracted records are valid, grounding represents the relevant alternatives, and each primitive conservatively encloses its concrete outputs, then the composition conservatively encloses the corresponding program outputs. The argument is induction over the DAG. Type validation alone establishes none of the natural-language premises, and an abstract domain may lose precision even when it remains conservative. Unrepresented alternatives, unsupported transfers, or missing required information remain unresolved.

C DISCLOSURE AND WORKED EXECUTION

C.1 REQUESTED PRECISION AND EVIDENCE SCOPE

DISCLOSE evaluates the final execution state against the answer contract $\chi_q = (\tau_q, g_q, \sigma_q)$. The same rule applies to every supported computation: the represented result must determine one answer at the requested precision, its evidence must support the requested scope, and no answer-relevant requirement may remain unresolved. Operator execution propagates this information; the final check determines the payload supplied with the reader's existing context.

Let $\gamma(\widehat{V}_q)$ be the concrete values represented by the final abstract value and g_q map them to the requested precision. The image $\mathcal{Y}_q = g_q[\gamma(\widehat{V}_q)]$ contains the answers allowed by the represented values at the requested precision. Value admission requires a singleton image. This check does not require unrestricted possible-world enumeration: finite candidate sets and supported interval checks suffice. For example, all days in a represented interval within May determine the same month, whereas an interval spanning April and May does not.

SCOPEOK compares the result's supported coverage ς_q with the requested scope σ_q. For a question explicitly naming a finite set of items, the requested set is established once every named item is

resolved. Alternatively, a source-supported exhaustive enumeration can establish coverage when its interpreted domain, predicate, and time range match the requested scope and its named members are resolved against the retrieved records. The check therefore uses the interpreted scope descriptor and resolved membership, rather than the presence of an exhaustiveness cue alone.

The exhaustive statement and the members' values need not occur in the same session. In the Paris trip example, Session 2 supplies the transport enumeration, while the selected amounts come from both sessions. An exhaustive enumeration does not compensate for a missing amount or an unresolved item correspondence.

Retrieval size, scanning all returned records, or a proposer's assertion of completeness are insufficient. Source-supported closure concerns what the sources report, not unobserved real-world events.

Conditions are tracked during execution and checked against the query at disclosure. Supported checks discharge their obligations; $\mathcal{O}_q$ in Eq. 4 denotes what remains. ScopeOK makes the coverage requirement explicit, while the obligation set also tracks missing fields, units, ordering, and identity. If an implementation stores a coverage failure in that set as well, both tests express the same unresolved coverage issue; they are not independent sources of confidence.

Condition	**Payload**	**Reader context**
Valid execution; one requested answer; supported scope; no pending requirement	Value + sources	C_q^W plus evidence and derived value
Valid program with usable source evidence, but no admitted value	Source evidence	C_q^W plus evidence, without a derived answer
Malformed program, contradictory required assertion, or no usable evidence	Empty	Exactly C_q^W

Table 13: **One disclosure rule.** The value branch takes precedence. Unresolved execution and final-reader refusal are different events.

C.2 Benchmark Traces: Value and Evidence-Only Disclosure

We present two benchmark traces illustrating value and evidence-only disclosure. The temporal example uses automatic grounding and the unmodified executor and disclosure implementation. Reference answers are reported separately from execution outputs and reader-facing payloads.

Temporal execution with value disclosure. Question `b3c15d39` asks how many days it took to receive a new remote shutter release after ordering it. The reference answer is five days. Automatic grounding identifies the order in `answer_05d808e6_1` on 5 February 2023 and the arrival in `answer_05d808e6_2` on 10 February 2023. The ordering source states: "looking for my tripod on February 5th, which is also when I ... lost my remote shutter release, so I ordered a new one online." The arrival source states: "a new remote shutter release that arrived on February 10th."

The executor applies the directed elapsed-time operation. Here `DirectedElapsed` is a descriptive name for time projection and directed arithmetic with the required ordering check; it uses the same primitive contracts as the other temporal computations:

$$\text{DirectedElapsed}(\text{earlier} = \text{ordered}, \text{later} = \text{arrived}) = 10\ \text{February} - 5\ \text{February} = 5\ \text{days}.$$

The grounded dates satisfy the required order and determine one answer at day precision. The question concerns these two events rather than an exhaustive collection of events. The recorded disclosure decision is therefore `value`, supplying five days with provenance from both sessions.

The final reader retains access to the original retrieved text and WRITE annotations.

Event counting with evidence-only disclosure. The question asks, "How many times has Calvin had to deal with insurance paperwork?" The reference answer is "Once." The recorded `COUNT_EVENTS` execution completes with status `OK`, but its reader-facing payload contains matching source turns without a derived count. One turn describes dealing with insurance and repairs after

a car accident; another responds to that account and also mentions insurance. Multiple matching turns need not correspond to distinct events. The reader therefore determines the event count from the supplied evidence and its retained context. This trace illustrates successful execution with evidence-only disclosure: execution status alone does not imply that a numerical answer is supplied to the reader.

C.3 Evidence-Only and Exact Fallback

For a valid executed trace, Evidence-only and Full share the same selected sources, their order, and source rendering. Full adds a derived-value field only when Ready holds. Evidence-only omits that field without removing source numbers or pre-existing WRITE annotations. It also does not insert a computed answer through an explanatory selection label. Unresolved associations remain explicitly unresolved in the evidence payload.

Only the empty-payload branch requires $\mathrm{hash}(C_q^W \oplus \varnothing) = \mathrm{hash}(C_q^W)$. An unresolved value can still produce evidence, so it need not satisfy that empty-payload identity. The reader remains free to answer from the underlying context.

C.4 Matched Representation and Disclosure Controls

Base uses fixed retrieved text; Write adds its materialized interpretations; Evidence-only adds source evidence selected by execution; Full also adds admitted values. All four conditions use the same question set, retrieved turn identities, and ordering, including questions receiving no execution payload. The Full–Evidence-only comparison additionally fixes the programs, grounding, execution traces, and source rendering, isolating the contribution of admitted values beyond their supporting evidence.

D Evaluation Protocol and Implementation Details

D.1 Question Sets, Slice Definitions, and Scoring

Data and slices. LoCoMo uses the official release with 10 conversations, 272 source sessions, and 1,986 questions. The temporal slice is category 2 (321 questions); the remaining 1,665 questions form the non-temporal slice. Category 5 contains adversarial questions; these are part of the non-temporal slice and are scored by refusal accuracy. LongMemEval-S uses 500 questions: multi-session (133), temporal reasoning (133), knowledge updating (78), single-session user (70), single-session assistant (56), and single-session preference (30). The temporal and non-temporal multi-evidence slices use the temporal-reasoning and multi-session labels, respectively. EgoMem contains seven days of real-world egocentric lifelog data derived from EgoLife, comprising approximately 1,700 sessions and 1,774 questions. LifeMem contains one year of LLM-simulated lifelog data with 1,717 questions. Both benchmarks organize questions into four categories (QT1–QT4), each accounting for approximately one quarter of the questions. LifeMem additionally groups questions by retrieval horizon: daily (1,430), weekly (241), and monthly (46). RHELM covers 10 personas, 11,764 conversational turns, 2,180 external sources, and 1,305 questions, with approximately 500K–1M tokens per persona. Its questions span seven categories and are characterized by 27 complexity features. Benchmark categories and slice definitions are fixed before examining model outputs.

Models and retrieval. The reader models are Qwen2.5-7B-Instruct and Gemma-3-12B-Instruct, with greedy decoding at temperature zero. Dense retrieval uses `all-MiniLM-L6-v2` on every benchmark, with 384-dimensional L2-normalized embeddings and a 256-token encoder input limit. The same retrieval encoder is used for BindMem and every compared baseline, so no comparison reflects a difference in embedding model. The archive retrieval budget is $r = 4$ sessions for LongMemEval-S/M, $r = 10$ for LoCoMo, $r = 14$ for LifeMem, and $r = 10$ for EgoMem and RHELM; these depths follow each benchmark's own evaluation setting. No additional application-level token cap is imposed on the final-reader context; the 256-token limit applies only to the retrieval encoder input. For EgoMem and LifeMem, the benchmarks' evaluation protocols enforce temporal causality: sessions occurring after the query time are excluded from the accessible memory. For RHELM, conversations are indexed at the turn level, while external documents are divided into 500-token chunks with a 50-token overlap. These benchmark-specific retrieval choices are

separate from the shared memory interpretation, execution, and disclosure components. The source interpreter, query compiler, grounding rules, and execution and disclosure semantics are shared across benchmarks. Within each matched comparison, retrieval rankings are computed once, and retrieved identities and their ordering are held fixed across experiments.

Scoring. LoCoMo end-to-end comparisons use factual-answer F1 and adversarial refusal accuracy under the benchmark's evaluation protocol. LongMemEval-S and -M use the official answer-checking templates with GPT-4.1 as the judge. An empty answer receives zero credit. For EgoMem and LifeMem, all reported results use the multiple-choice setting and are scored by exact match against the correct option. RHELM uses GPT-4.1 as an LLM judge under its evaluation protocol. Within each benchmark, all compared systems use the same scoring procedure and judge configuration.

D.2 Source Parsing and Supported Interpretations

The reported implementation uses spaCy `en_core_web_sm` as a fixed dependency parser followed by deterministic typed extraction rules; it makes no ingestion-time LLM calls. The parser supplies syntactic structure but does not determine cross-session identity or query-dependent computations. Stored fields include semantic type (event, scalar, or time), available speaker metadata, normalized predicate/content terms, source text, event time and precision, scalar value/unit/noun, local references, source session/timestamp/span, and provenance. Unsupported or ambiguous fields remain unresolved. The following paragraphs specify the supported source-local interpretations used in the experiments.

Temporal interpretation. Fixed lexical patterns identify supported relative-time expressions, including "yesterday," "N days/weeks/months/years ago," "last weekday," and "last week/weekend." They are resolved by deterministic calendar arithmetic against the session timestamp. Outputs retain point or interval precision. For example, "90 days ago" anchored to 11 August 2023 resolves to 13 May 2023. A question-introduced reference time remains a READ input, and an unresolved day is not filled merely to enable a later operation.

Quantities and local references. Supported numeric expressions include dollar-prefixed amounts, numeric noun phrases, number words from one to twelve, and "once"/"twice." Numeric and unit spans are associated with their governing nominal head using the dependency structure, including supported numeric-modifier and compound relations. The resulting scalar retains the value, normalized unit, and item noun; stopword nouns and nouns shorter than three characters are skipped. First-person possessive role phrases are bound to available speaker metadata, for example "my husband" to "Melanie's husband" when Melanie is the speaker. A third-person pronoun is bound only when exactly one compatible named candidate occurs within the declared local-context window; otherwise the reference remains unresolved.

Membership and scope. A scope descriptor is constructed only for supported completeness expressions, identified by fixed cues such as *only*, *nothing else*, *that was all*, or an explicit cardinality. The dependency parse identifies coordinated nominal phrases as candidate members, the governing predicate nominal as the collection domain, and supported nominal modifiers as its predicate. The descriptor retains these fields together with the complete supporting source span and any locally expressed time range. It is admitted only when the required members are successfully parsed and any stated cardinality agrees with the resolved member count; otherwise coverage remains unresolved.

At READ, the named members are associated with the corresponding retrieved records, retaining provenance from both the membership statement and the matched records. The descriptor supports the requested scope only when its domain, predicate, and time range match the question and the required member correspondences are resolved. Thus, recognizing an exhaustiveness cue alone does not establish ScopeOK. Missing amounts and unresolved associations remain execution obligations and prevent value admission when required by the question.

D.3 Program Interfaces and Representation

The deterministic grammar, compositional recognizer, and enabled language-model proposer supply specifications to the same typed builder. The proposer is invoked only when both deterministic paths

decline (Appendix D.4). Table 14 relates the surface vocabulary to computations over the shared primitives. The builder checks supported specifications and their dependencies before execution. In a dependent composition, an intermediate result can determine a later operand or input collection. Joins follow source-supported references, and unresolved associations remain part of the execution state. Program proposal is the only auxiliary LLM path enabled in the reported runs; its calls and input/output tokens are included in the auxiliary-LLM counters.

Surface operation	Computation
`WHEN`	PROJECT the grounded event's time field.
`HOWLONG_BETWEEN`	PROJECT event times and apply ARITH to their difference.
`HOWLONG_AFTER`	Apply time projection and ARITH with the required strict-order check.
`HOWLONG_SINCE`	Apply time projection and ARITH using the query reference time.
`ORDER_FIRST`	Use the ordering and selection operators to select the earliest event.
`CURRENT`	Apply the query-time constraints and selection operators to obtain the requested value.
`COUNT`	FILTER the collection and apply REDUCE with count.
`ENUMERATE`	Apply DISTINCT and render the selected records as source evidence.
`COUNT_EVENTS`	Apply DISTINCT, then REDUCE with count.
`SUM_SCALAR`	Filter the scalar records, project their values, and apply REDUCE with sum.
`ARGMAX_SCALAR`	Use source-supported scalar associations and ordering/selection to return the corresponding event.
`NONE`	Decline and retain the reader path.

Table 14: **Frontend operation vocabulary.** The computation column relates surface requests to the shared primitive contracts; each request may lower to multiple operators. `NONE` requests no program. Recognizing a request does not establish grounding or value admissibility.

These frontend forms do not determine binding time. Staging follows the input-dependency rule in Appendix A.3: source-local interpretations can be materialized at WRITE, while query-dependent computations remain at READ.

D.4 PROPOSER PROMPT, CLOSED DSL, AND ACTIVATION GATE

Activation gate. The proposal path is not a fallback for every hard question: it is invoked only after a deterministic decline. A question first goes to the grammar, which maps recognized cues and answer-type constraints to a surface operation from Table 14; a declined question is then offered to the compositional recognizer, which accepts it only when the question matches a supported combination of temporal relation and output computation. The proposer is called only when both decline, and it may itself return `DECLINE`, in which case the question retains the unmodified reader path. Grammar-handled and composition-handled questions therefore issue no proposal call, which is why the averages in Table 15 fall below one call per question.

Proposer model and generation settings. The proposer and final reader use the same model: Qwen2.5-7B-Instruct, Gemma-3-12B-Instruct, or GPT-5 in the corresponding evaluation. For Qwen and Gemma, proposal decoding is greedy at temperature zero, with an output limit of 128 tokens. A single proposal attempt is made per question; rejected proposals are not regenerated or repaired. In the frontier-model evaluation, GPT-5 is used for both program proposal and final answering. Retrieval, stored WRITE records, the deterministic compiler and grounding rules, execution semantics, and disclosure policy are unchanged; model-generated proposals need not be identical across reader models. These fixed-retrieval results assess the benefit of the interface with GPT-5 and are not an oracle-evidence diagnostic. The frontier-model evaluation uses `openai/gpt-5` through OpenRouter. Both program proposal and final answering use `high` reasoning effort and a maximum output budget of 2,000 tokens per call.

Prompt. The proposer prompt is reproduced verbatim below. It contains no few-shot examples: `{ops}` is filled with the surface-operation vocabulary of Table 14 and `{q}` with the question, and no further demonstrations, reference answers, or retrieved evidence are supplied. Table 14 names the surface tasks; the parser also accepts the registered DSL forms described below.

```
Express this question as ONE program in the following language,
or reply DECLINE.

Operations: {ops}
Events are written EVENT(predicate="...", ...) with keyword
arguments naming the event.

Question: {q}

Reply with ONLY the program, no explanation:
```

Closed DSL. The proposal language admits nested calls whose arguments are either a nested call or a string-valued keyword argument; no free-form code, arithmetic, literals of other types, or identifiers outside the registered vocabulary are expressible:

$$\langle program \rangle ::= \langle op \rangle \texttt{(} \langle arg \rangle (\texttt{, } \langle arg \rangle)^* \texttt{)} \mid \texttt{DECLINE},$$
$$\langle arg \rangle ::= \langle program \rangle \mid \langle key \rangle \texttt{="} \langle string \rangle \texttt{"},$$

where $\langle op \rangle$ ranges over the registered surface operations, supported DSL forms including `TIME` and `AFTER_DIFF`, and the event constructor `EVENT`; $\langle key \rangle$ ranges over the declared keyword arguments of the corresponding operation.

Surface operations and DSL forms. Surface names identify the requested computation; nested DSL calls make its operands and dependencies explicit. `EVENT` supplies an event description for grounding. `TIME(EVENT(...))` specifies the time projection associated with `WHEN`. `AFTER_DIFF(`t_a`,`t_b`)` specifies `HOWLONG_AFTER`, with the earlier event first and the later event second; its lowering retains the strict-order requirement. Both forms use the same typed builder, grounding, operator contracts, and disclosure checks. `DECLINE` requests the no-program outcome represented by `NONE`. These forms do not extend the primitive registry or the supported composition repertoire.

A proposal for "How long after moving did I start the new job?" has the form

```
AFTER_DIFF(TIME(EVENT(predicate="move", agent="self")),
           TIME(EVENT(predicate="employment_start")))
```

in which the strings supply only event descriptions to be grounded, not record identifiers or source locations.

Admission of a proposal. A returned string is parsed into an abstract syntax tree and passed to the same builder and validator used by the grammar path. Unparseable output, unknown operation or keyword names, arity or type mismatches, and arguments of a type the operation does not accept are rejected, and the question retains the reader path. An admitted proposal determines only the requested computation and its operand descriptions; grounding, execution, and value admission then proceed exactly as for a grammar-compiled program, so a well-formed proposal confers no additional access to the archive and no relaxation of the disclosure checks.

E Computational Cost

E.1 Auxiliary LLM Calls at Query Time

WRITE makes zero auxiliary LLM calls. At query time, grammar-handled questions require no proposal call; the hybrid frontend invokes the proposer through the program-proposal gate described in Appendix D.4. Table 15 reports proposal tokens and additional LLM calls averaged over all LongMemEval-S questions, including questions with zero proposal usage.

E.2 Ingestion, Query, and Storage Cost Across Memory Systems

Table 15 characterizes only the auxiliary calls issued by our own frontend. This subsection compares BindMem with A-MEM, MemGAS, HippoRAG, and Mem0, each of which constructs a persistent

Condition	Proposal tokens / question		Extra LLM calls / question	
	Qwen	Gemma	Qwen	Gemma
Base	0	0	0	0
Write	0	0	0	0
Full	119	111	0.73	0.73

Table 15: **Auxiliary LLM cost on LongMemEval-S.** Proposal tokens include input and output tokens. Averages include grammar-handled questions with no proposal call. Extra calls exclude the final reader call and evaluation judging. These measurements characterize auxiliary LLM usage; they do not measure total computation, ingestion time, or storage size.

System	Ingestion LLM tokens	Ingestion calls	Ingestion time (s)	Read tokens / question	Added storage (× raw)	LoCoMo (%)	
						Q	G
A-MEM	245K	299	170	11.5K	**1.19**×	35.1	42.2
HippoRAG	239K	299	102	10.1K	4.20×	37.4	44.5
MemGAS	484K	544	63	3.8K	12.56×	38.5	44.3
Mem0	950K	1,190	2,779	**2.8K**	6.10×	34.1	40.2
BINDMEM	**0**	**0**	**8.3**	6.3K	1.43×	**46.5**	**50.4**

Table 16: **Store-construction cost, reader-context size, and added storage on LoCoMo.** Measurements cover the same 10 dialogues (272 sessions). Construction times exclude common raw-text embedding and indexing and are not concurrency-matched. Added storage is measured beyond the raw archive. LoCoMo scores repeat Table 3; Q/G denotes the Qwen/Gemma reader. Bold marks the lowest reported cost or highest score in each column.

store from the raw archive before evaluation queries are answered. We exclude MemGPT and MemoryBank from this cost comparison because their stores are maintained incrementally during interaction.

Measurement scope. All measurements use the same 10 LoCoMo dialogues (272 source sessions) as Table 3, on one host with the same CPU and one NVIDIA A100 GPU available. All systems use the shared `all-MiniLM-L6-v2` encoder of Appendix D; their storage units and method-specific indexes may differ. Ingestion tokens count the input and output tokens of every LLM call issued during store construction. A-MEM and HippoRAG each process 299 chunks and issue one note-generation or OpenIE call per chunk, respectively; these chunk counts are distinct from the number of source sessions. MemGAS processes 2×272 chunks. Mem0 extracts facts over sliding windows, so its 1,190 ingestion calls do not correspond one-to-one with sessions. Construction times exclude the common raw-text embedding and index construction but include method-specific derived embedding costs. BINDMEM processes 5,882 turns in 8.3 s on a single CPU thread without GPU computation, including parsing, record construction, provenance, and serialization. A-MEM, MemGAS, and HippoRAG use internal batching, whereas Mem0 runs sequentially; the reported wall-clock times are therefore not concurrency-matched.

Read tokens per question count the context supplied to the final reader, excluding its generation and evaluation judging. For BINDMEM, this context includes the rendered WRITE records and DISCLOSE payload; proposal tokens are reported separately in Table 15. Added storage counts bytes beyond the raw archive, divided by the raw archive size, including derived text, embeddings, and auxiliary structures. Methods that embed derived storage units use the shared encoder, but their embedding volumes depend on the units they create. BINDMEM reuses the raw-text retrieval index and adds no derived embeddings or separate index.

Where the cost goes. BINDMEM constructs its WRITE records with fixed syntactic parsing and deterministic typed extraction, issuing no ingestion LLM calls. A-MEM, HippoRAG, and MemGAS consume 239–484K ingestion tokens, while Mem0 consumes 950K. BINDMEM supplies 6.3K read tokens per question, fewer than A-MEM (11.5K) and HippoRAG (10.1K), but more than MemGAS

System	LLM calls		Calls per question at Q questions			
	Ingestion	Read / q	Q=1,986	Q=100	Q=10	Q=1
A-MEM	299	**1**	**1.15**	3.99	30.9	300
HippoRAG	299	**1**	**1.15**	3.99	30.9	300
MemGAS	544	3	3.27	8.44	57.4	547
Mem0	1,190	**1**	1.60	12.9	120	1,191
BINDMEM	**0**	1.46	1.46	**1.46**	**1.46**	**1.46**

Table 17: **LLM calls per question as a function of the question budget.** The amortized call count is $C_{\text{ingest}}/Q + C_{\text{read}}$, with the measured mean read-time rate held fixed. Read-time calls include final answering and auxiliary LLM calls, but exclude evaluation judging. BINDMEM uses one final-reader call plus 0.458 proposal calls per question on LoCoMo, rounded to 1.46. Both calls use the serving reader model. Its proposal rate is 0.73 on LongMemEval-S (Table 15). MemGAS uses two auxiliary read-time calls for summary and keyword reranking plus one final-reader call. Bold marks the minimum in each column.

System	Read stage (s/q)	Additional overhead (s/q)	Total query (s/q)
A-MEM	0.24	≈0	0.24
HippoRAG	0.24	≈0	0.24
MemGAS	0.72	≈0	0.72
Mem0	0.24	≈0	0.24
BINDMEM	0.24	0.20	0.44

Table 18: **Reported query time on the 10 LoCoMo dialogues.** Total query time is the sum of the two reported components, expressed as amortized wall-clock seconds per question, and excludes one-time store construction (Table 16). Systems run at their default concurrency, so these measurements are not concurrency-matched. Component boundaries follow each implementation: the read-stage entry covers final answering for A-MEM, HippoRAG, Mem0, and BINDMEM, and all three read-time calls for MemGAS. BINDMEM's additional overhead includes compilation, grounding, execution, and proposal calls. Table 17 reports call volume independently of concurrency.

(3.8K) and Mem0 (2.8K). Its reader context retains the retrieved text and WRITE annotations and includes any payload selected by DISCLOSE.

Cost and answer quality. Different systems minimize different cost measures in Table 16. Mem0 supplies the fewest read tokens per question (2.8K), while A-MEM has the smallest added storage (1.19× raw). Relative to MemGAS, BINDMEM uses less added storage (1.43× versus 12.56× raw) and improves the LoCoMo score by 8.0/6.1 points for Qwen/Gemma. Compared with A-MEM, it also uses more added storage (1.43× versus 1.19×), while requiring no ingestion LLM calls and supplying a smaller reader context. These measurements describe the systems' accuracy–cost trade-offs; no system minimizes every reported cost.

E.3 CALL BUDGETS AND QUERY-TIME

Table 16 reports store-construction costs, reader-context size, and added storage. This subsection reports LLM call volume and query-time measurements on the same 10 LoCoMo dialogues (272 source sessions, 1,986 questions). For hypothetical question budgets, we hold each system's measured mean read-time call rate fixed and amortize its ingestion calls over Q questions.

Ingestion. BINDMEM constructs WRITE records without ingestion LLM calls. A-MEM and HippoRAG each issue 299 calls, MemGAS issues 544, and Mem0 issues 1,190. Their construction times are reported in Table 16.

The comparison depends on the question budget. Amortizing ingestion over all 1,986 LoCoMo questions gives A-MEM and HippoRAG fewer LLM calls per question than BINDMEM (1.15 versus 1.46). At 100 questions, their amortized call count rises to 3.99, and at 10 questions to 30.9. Under the fixed mean read-time rate used in Table 17, BINDMEM remains at 1.46 because it has no ingestion calls. Mem0 uses 1.60 calls per question at $Q = 1{,}986$, while MemGAS exceeds BINDMEM at every positive Q because its read stage alone requires three calls. Zero-call ingestion therefore provides the largest reduction in amortized LLM call volume when an archive serves few queries. Call counts do not account for differences in token volume or execution time.

Query time. The reported read-stage time is 0.24 seconds per question for A-MEM, HippoRAG, Mem0, and BINDMEM. MemGAS's 0.72-second entry covers its two auxiliary read-time calls and final-reader call. BINDMEM adds 0.20 seconds for compilation, grounding, execution, and proposal calls, yielding 0.44 seconds per question. Under the measured default-concurrency configurations, this exceeds the reported time for A-MEM, HippoRAG, and Mem0 and is below that of MemGAS. These timings characterize the evaluated configurations alongside their answer scores; they do not establish a concurrency-matched latency ranking.

F ADDITIONAL RESULTS

F.1 PYTHON BASELINE SETUP AND FAILURE ANALYSIS

Models and generation budget. The Python baselines use the serving reader, Qwen2.5-7B-Instruct or Gemma-3-12B-Instruct, for code generation with greedy decoding at temperature zero. In the primary no-repair configuration, each question receives one code-generation attempt with a maximum of 500 output tokens and no retry or repair. Table 19 additionally reports variants allowing one repair attempt after an execution error. The read-time interpretation condition additionally uses one normalization call with a maximum of 700 output tokens. Without repair, the read-time and WRITE conditions use two and one auxiliary LLM calls per question, respectively, excluding the final reader and evaluation judge. A repair variant adds one code-generation call when its initial execution fails.

Inputs and code generation. The read-time normalization call converts retrieved context into JSON records with fields `date`, `amount`, `entity`, `source`, and `uncertain`. In the WRITE condition, the baseline uses the same stored records as BINDMEM, serialized as event entries (`- event: ... | time=...`) and scalar entries containing the value, unit, noun, and supporting text. The code-generation prompt asks for a short Python program that uses the supplied `MEMORY` and `RECORDS` strings and prints its answer. The read-time condition uses LLM normalization, whereas WRITE uses the fixed source parser and deterministic typed extraction pipeline. The conditions also differ in record format, so their comparison does not isolate interpretation timing alone.

Execution and reader context. Programs run in an isolated Python process (`python -I`) with an empty environment and a 10-second timeout. The permitted libraries are `datetime`, `re`, `math`, `json`, and `collections`. Pre-execution checks reject prohibited imports and calls to `open`, `eval`, and `exec`; file, network, external archive, and gold-answer access are prohibited. On successful execution, the final reader prompt includes the record block under `[NORMALIZED RECORDS]` and standard output under `[CODE EXECUTION RESULT]`. In the no-repair configuration, syntax errors, runtime errors, and timeouts trigger immediate fallback. In the repair variants, a remaining execution failure after the allowed repair attempt triggers fallback. Failed code outputs are discarded, and the reader answers from its existing context without an execution-result block or error message. These questions remain in the evaluation denominator.

Execution success and final-answer accuracy. For the matched JSON read-time and custom-format WRITE configurations, allowing one repair reduces execution failures but does not improve overall accuracy. The largest change is for Qwen with read-time interpretation: execution failure falls from 54.4% to 7.4%, while overall accuracy falls from 47.8% to 34.8%. Its Code-OK accuracy is 31.6% without repair and 33.3% with repair, and the corresponding Code-FAIL accuracies are 61.4% and 54.1%. Because repair changes which questions belong to each subset, these conditional accuracies do not identify the cause of the overall change. Across all ten configurations, Code-FAIL accuracy exceeds Code-OK accuracy by 0.6–29.8 points; this descriptive association does not establish that

Interp.	Serial.	Rep.	Exec. fail (%)	Acc. (%) Code-OK	Acc. (%) Code-FAIL	Overall acc. (%)
Qwen2.5-7B						
Read-time	JSON	✗	54.4	31.6	61.4	47.8
Read-time	JSON	✓	7.4	33.3	54.1	34.8
WRITE	custom	✗	16.0	41.0	58.8	43.8
WRITE	custom	✓	5.4	40.0	48.1	40.4
WRITE	JSON	✓	25.0	40.0	51.2	42.8
Gemma-3-12B						
Read-time	JSON	✗	26.6	52.3	66.9	56.2
Read-time	JSON	✓	21.8	54.2	63.3	56.2
WRITE	custom	✗	63.4	59.6	60.2	60.0
WRITE	custom	✓	47.4	57.4	62.4	59.8
WRITE	JSON	✓	71.8	54.6	62.7	60.4

Table 19: **Python baseline configurations and failure analysis on LongMemEval-S.** Single-run diagnostics over 500 questions per condition. *Rep.* indicates one automatic repair attempt on execution error; in that condition, an execution failure triggers one additional code-generation call containing the previous program and its execution error, with no reference answer or correctness feedback supplied. Code-OK and Code-FAIL report final-reader accuracy on the execution-success and execution-failure subsets; these subsets contain different questions, so their difference does not isolate the effect of supplying code outputs. On failure, the reader answers from its existing context, and the question remains in the overall denominator. The no-repair rows share the retrieved turns, final reader, and scoring procedure of Table 8.

suppressing code outputs would improve accuracy on the same questions. Execution success should therefore not be treated as evidence of correct interpretation or a correct final answer.

Table 8 reports the no-repair single-attempt configuration, which BINDMEM exceeds by 11.0 points for Qwen and 6.1 for Gemma. Across the configurations in Table 19, the strongest Python scores are 47.8 for Qwen (read-time JSON without repair) and 60.4 for Gemma (WRITE records, JSON, one repair), below BINDMEM by 7.0 and 5.7 points, respectively. Gemma's highest-scoring Python configuration also has its highest execution failure rate (71.8%), reinforcing the distinction between execution success and final-answer quality.

Source interpretation pipeline and serialization format. With JSON serialization and one repair attempt fixed, using deterministic WRITE records instead of read-time LLM normalization increases accuracy by 8.0 points for Qwen and 4.2 for Gemma. This comparison changes both the interpreter and its execution stage, and therefore does not isolate an effect of interpretation timing. With WRITE records and one repair attempt fixed, changing the custom serialization to JSON increases accuracy by 2.4 and 0.6 points, respectively. None of these configurations reaches BINDMEM; read-time Python with repair also falls below the raw-reader baseline for Qwen.

F.2 PAIRED SCORE CHANGES UNDER DISCLOSURE CHECKS

Table 20 reports paired changes in per-question benchmark scores between Without checks and Full, using the same scoring and run aggregation as Table 6. Improved and Worsened count questions whose scores increase and decrease, respectively; Net is their difference. These counts summarize the direction of score changes, not their magnitude.

G COMPILER AND EXECUTION RELIABILITY DETAILS

This section evaluates operation selection and compilation behavior under question reformulation. The complementary full-trace audit in Table 5 evaluates operations, grounding, dependencies, and computed values.

Benchmark	Reader	Improved	Worsened	Net
LoCoMo	Qwen	74	8	+66
	Gemma	65	15	+50
LME-S	Qwen	31	10	+21
	Gemma	27	12	+15

Table 20: **Paired per-question benchmark score changes when enabling scope and obligation checks.** Improved and Worsened count score increases and decreases, respectively; Net is Improved minus Worsened.

G.1 Aggregate Question-Reformulation Diagnostics

For each compiler configuration, agreement compares the operation selected for a reformulated question with that selected by the same configuration for the original question.

Metric	Grammar-only	Cascaded
Compile coverage	66.7	**81.9**
Strict agreement | compiled	**85.7**	80.8
Family agreement | compiled	87.9	**89.5**
Joint coverage × strict agreement	57.1	**66.2**
Joint coverage × family agreement	58.6	**73.3**
Ill-typed executions	0	0

Table 21: **Question-reformulation diagnostics.** Percentages use 42 executable originals and five transformations; the final row is a count. Joint metrics account for both compilation coverage and agreement.

The cascade increases compilation coverage and improves joint operator-family coverage by 14.7 points. Table 21 reports both strict and family-level agreement. No ill-typed programs execute in this evaluation.

G.2 Human Operator-Intent Audit

We additionally checked whether the compiled operation expresses what the question asks. From the questions the frontend compiles, we sampled 72 on LoCoMo and 61 on LifeMem uniformly at random; an annotator labeled the intended operation family for each without seeing the compiler output, and we counted a match when the compiled family agreed with that label. Table 22 reports 72/72 and 60/61. This measures operation selection only: a matched family does not establish that grounding bound the intended records or that the executed value is correct, which the trace audit in Table 5 evaluates separately.

Benchmark	Audited	Matched	Agreement
LoCoMo	72	72	**100.0%**
LifeMem	61	60	**98.4%**

Table 22: **Human operator-intent agreement.** A match requires the compiled operation family to express the question's annotated intent.

G.3 Question-Reformulation Breakdown

Table 23 breaks down compilation coverage across five question-reformulation types, each applied to the same 42 executable source questions.

Reformulation type	Grammar-only	Cascaded
Lexical substitution	61.9	61.9
Word-order change	54.8	**83.3**
Indirect interrogation	47.6	**76.2**
Conversational	90.5	**92.9**
Implicit temporal	78.6	**95.2**

Table 23: **Compilation coverage by reformulation type (%).** The cascade yields the largest coverage gains on word-order changes, indirect questions, and implicit-temporal reformulations.

H TRANSFER ACROSS RETRIEVAL BACKENDS

This control evaluates the deterministic staged representation-and-execution core with the structured proposer disabled. For each backend, a per-question ranking is computed once and then held fixed across the Base and deterministic core arms. Retrieved turn identities, ordering, reader instruction, decoding configuration, and judge are therefore identical within each comparison.

Retriever	Temporal gain Qwen	Temporal gain Gemma
Dense	$+21.2$	$+17.5$
HippoRAG	$+18.7$	$+14.0$
A-MEM	$+14.6$	$+10.6$

Table 24: **Temporal gains across retrieval backends on LoCoMo.** Score gains in percentage points. Within each retriever–reader pair, both receive exactly the same retrieved evidence.

The targeted temporal effect of the shared staged core remains large under every retrieval backend, ranging from $+10.6$ to $+21.2$ percentage points. The HippoRAG and A-MEM rankings are recomputed with their corresponding retrieval implementations. Each ranking is then held fixed and shared exactly across its matched arms.