MEMORY AS CODE: DECLARATIVE RETRIEVAL OVER CODIFIED LONG-TERM MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

In long-term memory systems, memories are constructed from interaction histories and retrieved in response to queries, enhancing the reasoning ability of LLM agents. In complex scenarios, facts are scattered across sessions and their relationships evolve over time, making it difficult for existing memory systems to compose them to adapt to diverse queries. Motivated by the modularity and composability of code, we propose **M**emory **a**s **C**ode (MaC), a unified memory framework that jointly designs memory construction and retrieval through a shared code abstraction. MaC couples *codified construction*, which establishes executable memory abstractions, with *declarative retrieval*, which translates queries into structured retrieval programs. Specifically, *codified construction* builds a three-layer executable memory structure that progresses from an atomic function layer, through a relational layer, to a modular composition layer. *Declarative retrieval* recasts retrieval from relevance-based fact selection into declarative query processing over memory. Experiments on LOCOMO and LONGMEMEVAL show relative gains of up to 15.91% in overall LLM-judge accuracy over the strongest baseline in each setting, while substantially reducing token consumption, highlighting the effectiveness of memory as code for long-horizon memory tasks.

1 INTRODUCTION

Long-term memory enables Large Language Model (LLM) agents to retain and use information across interactions, supporting personalized assistance and long-horizon tasks (Zhong et al., 2024; Park et al., 2023). A memory system Xu et al. (2026a); Packer et al. (2023) comprises memory construction, which organizes interaction histories into memory data, and memory retrieval, which processes memory queries over memory data. For memory construction, MemoryOS (Kang et al., 2025) organizes dialogue into topic segments and summaries, Mem0 (Chhikara et al., 2025) consolidates facts through update operations, and A-Mem (Xu et al., 2026b) constructs dynamically linked memory notes. These methods support memory organization and maintenance, but provide limited support for encapsulating computations over stored facts for reuse across queries. For memory retrieval, existing approaches combine search with contextual or relational expansion to gather relevant evidence (Borgeaud et al., 2022; Liu et al., 2023), using either predefined pipelines (Han et al., 2024; Rasmussen et al., 2025) or agentic workflows (Yao et al., 2022; Li et al., 2025; Ji et al., 2026). However, predefined pipelines offer limited flexibility for diverse queries, whereas stepwise agentic methods achieve this flexibility through repeated LLM planning, incurring additional inference overhead(Kim et al., 2023; Tan et al., 2025).

The above observations motivate a joint design in which memory construction encapsulates reusable computations over stored facts, and retrieval composes them for each query while reducing the need for repeated planning. Code provides a suitable foundation: shared references preserve consistent fact identities, functions expose reusable computations through explicit interfaces, and programs specify their composition and conditional branches before execution begins. This leads to a code-centric formulation in which memory is a library of callable computations and retrieval is a query-specific program over that library.

However, achieving these goals through a shared code abstraction poses two challenges: (1) How can interaction histories be transformed into executable memory code? Facts expressed across sessions may overlap, supplement one another, or supersede earlier states, distinctions that are not

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.

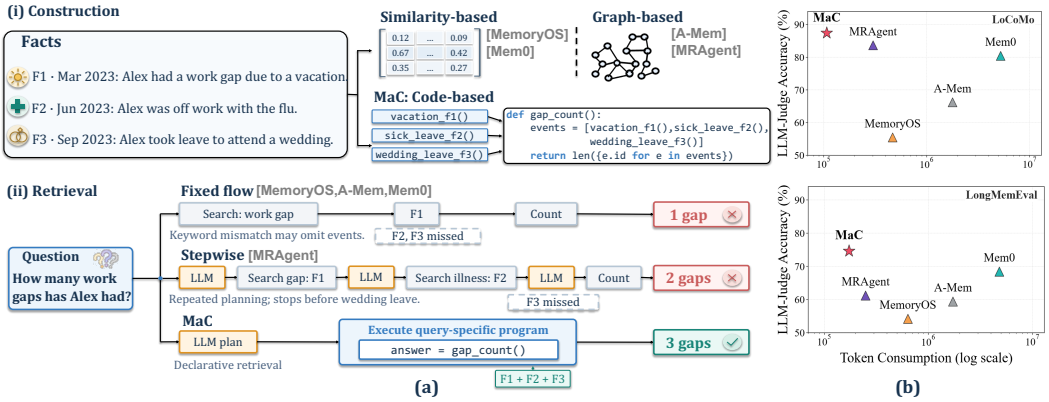


Figure 1: (a) An example illustrating the limitations of relevant methods. (b) MaC achieves the highest LLM-judge accuracy with the lowest token consumption among the compared methods on both LoCoMo and LongMemEval.

automatically resolved by encoding facts as functions. Preserving consistent fact identities and distinguishing current from historical states is a key challenge. (2) How can retrieval compose memory code to accommodate diverse queries? Relevant functions alone do not specify how their outputs should be combined, and subsequent operations may depend on intermediate results. Capturing these dependencies and conditional continuations in a complete retrieval program while reducing repeated LLM planning is another key challenge.

To address these challenges, we propose **M**emory **a**s **C**ode (MaC), a framework that represents long-term memory as executable code and retrieves information through query-specific programs. The framework comprises two components: (1) *codified construction* transforms facts extracted from interaction histories into a three-level executable memory structure. The atomic function layer encodes facts as functions with canonical identifiers and source references; the relational layer connects fact functions through explicit supersession relations while retaining historical evidence; and the modular composition layer composes base functions into reusable current-state and history computations. A compact memory index exposes function capabilities, arguments, and available operators for retrieval. (2) *Declarative retrieval* interprets each query as a high-level specification and compiles it into a complete retrieval program in one LLM planning call. Using the memory index, the program selects and composes the required memory computations, specifying operator arguments, dependencies, and conditional continuations. An executor checks and runs the program, following predefined branches based on intermediate results and assembling supporting evidence without stepwise LLM planning during successful execution. The two components work together through the memory index and a common evidence representation. The index connects constructed capabilities to query requirements, while the evidence representation supports result composition and source tracing. As illustrated in Figure 1, this design connects facts distributed across sessions through reusable functions to answer the query.

Our main contributions are as follows:

- We study long-term memory for LLM agents to support reasoning over evolving information across interactions. We propose **M**emory **a**s **C**ode (MaC), a framework that uses a shared code abstraction to connect memory capabilities with query-specific composition.
- We develop two complementary techniques: *codified construction* organizes facts, their evolving relations, and reusable computations in a three-level executable memory structure with consistent state semantics and source provenance; *declarative retrieval* compiles each query into a complete retrieval program that selects and composes these capabilities in one LLM planning call.
- Extensive experiments on widely used memory benchmarks show that MaC achieves the highest overall LLM-judge accuracy in all four evaluation settings, improving over the strongest baseline by up to 15.91% on LoCoMo and 9.06% on LongMemEval. Ablation studies further support the benefits of both techniques.

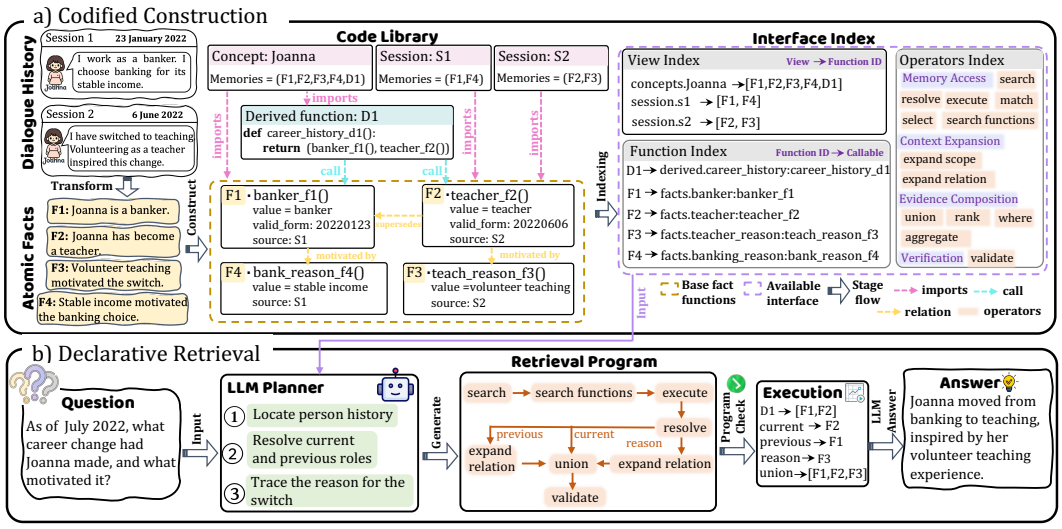


Figure 2: Overview of MaC. (a) *Codified construction* extracts atomic facts from dialogue history and builds a *code library* of fact functions, relations, derived functions, and concept and session views, along with an *interface index* that maps views to function identifiers (view index), identifiers to callable entry points (function index), and lists available memory operations (operators index). (b) *Declarative retrieval* passes the question and indexed interfaces to an *LLM planner*, which generates a *retrieval program*; after checking, an executor runs the program over the library to assemble supporting evidence for the answer LLM.

2 Memory as Code

In this section, we present MaC for long-term agent memory. We first introduce the problem definition and solution framework (§2.1), then describe codified construction (§2.2) and declarative retrieval (§2.3), and finally analyze the time and space complexity (§2.4).

2.1 Problem Definition and Solution Framework

Problem Definition. Given a dialogue history $\mathcal{H} = (s_1, \ldots, s_{n_{\text{sess}}})$ of n_{sess} sessions, memory construction organizes its information into an external memory $\mathcal{M}$. For a query q, memory retrieval accesses and combines information in $\mathcal{M}$ to produce supporting evidence E_q, which is used to generate an answer a_q. Appendix A.1 summarizes the notation used throughout the paper.

Solution Framework Overview. MaC organizes memory construction and retrieval around a shared code abstraction: memory provides reusable functions, and retrieval composes these functions into query-specific computations. As illustrated in Figure 2, the framework comprises two stages. *Codified construction* transforms dialogue history $\mathcal{H}$ into a code library of fact functions and reusable computations over current and historical states. A memory index exposes the callable interfaces and the available memory operators. *Declarative retrieval* uses these interfaces to translate a query q into a complete retrieval program in one LLM planning call. After checking, the program executes over the library to assemble supporting evidence E_q for answer generation. The two stages are connected by indexed interfaces and a common evidence representation, allowing retrieval to discover and compose the capabilities established during construction.

2.2 Codified Construction: Fact-to-Function

Given dialogue history $\mathcal{H}$, memory construction begins by extracting atomic facts to distill salient information and filter out irrelevant conversational details. We then encode these facts as executable memory to support reusable computation. However, directly encoding these facts into code is challenging because they are expressed in natural language, and the relations among them are complex and evolve over time. To address these challenges, we design codified construction to organize

facts into callable functions, explicitly represent their relations, and compose them into reusable operators. We first introduce the codified memory representation (§2.2.1), then describe memory organization and indexing (§2.2.2).

2.2.1 CODIFIED MEMORY REPRESENTATION

We define *codified memory* $\mathcal{M}$ that includes a *code library* $\mathcal{L}$ and a *memory index* I. The *code library* comprises four sets: *base fact functions* $\mathcal{F}$, *memory views* $\mathcal{V}$, *fact relations* $\mathcal{R}$, and *derived functions* $\mathcal{D}$:

$$\mathcal{M} = (\mathcal{L}, I), \qquad \mathcal{L} = (\mathcal{F}, \mathcal{V}, \mathcal{R}, \mathcal{D}).$$

The library organizes memory into three layers. The *atomic function layer* represents individual facts as callable functions in $\mathcal{F}$. The *relational layer* specifies fact connections and state replacements through $\mathcal{R}$. The *modular composition layer* provides derived functions in $\mathcal{D}$ for reusable computations over facts and their relations, including current-state resolution and history retrieval. Base and derived functions return evidence with fact identifiers and source references, supporting composition and provenance.

The set of memory views $\mathcal{V}$ contains *concept views*, which group function references by person or topic, and *session views*, which group them by dialogue session. The memory index I specifies view membership, callable function interfaces, and available memory operators, enabling retrieval to select and compose memory computations.

2.2.2 MEMORY ORGANIZATION AND INDEXING

We construct the three layers of the code library L from extracted facts, then build the memory index I to expose the resulting functions and memory operators for retrieval.

Code Library Organization. To translate natural-language facts into executable memory, we use an LLM to produce structured function specifications. A structured prompt specifies the required fact fields and function interface, and a local generator converts each specification into an executable base fact function.

At the *atomic function layer*, we canonicalize and deduplicate facts and assign each a canonical identifier i. For each canonical fact, the LLM produces a specification containing its structured value v_i, including the subject, predicate, and available time information, together with a statefulness annotation. The local generator uses this specification and the supporting dialogue references S_i to create a base fact function $f_i \in F$ that returns an evidence record $e_i = (v_i, \{i\}, S_i)$. Concept views and session views import shared references to these functions.

At the *relational layer*, we group stateful facts by subject and predicate and ask the LLM to identify explicit replacements. Each replacement becomes a supersession relation $(j, \texttt{supersedes}, i) \in \mathcal{R}$, indicating that fact j replaces fact i while both functions remain available. We also connect facts by temporal order, shared subject, and shared topic.

At the *modular composition layer*, we instantiate derived functions in $\mathcal{D}$ from reusable current-state and history templates over linked facts and register them in the relevant views. The resulting functions invoke base fact functions in $\mathcal{F}$ and use their relations in $\mathcal{R}$ to compute results. For a state group G sharing a subject and predicate, let $G(\tau)$ contain the identifiers of facts whose recorded times do not exceed the time boundary τ. The current-state template computes the active facts $A_G(\tau)$ and resolves the current value:

$$A_G(\tau) = \{i \in G(\tau) \mid \nexists j \in G(\tau) : (j, \texttt{supersedes}, i) \in \mathcal{R}\},$$
$$\text{current}_G(\tau) = \begin{cases} v_i, & A_G(\tau) = \{i\}, \\ \perp, & |A_G(\tau)| \neq 1. \end{cases} \tag{1}$$

Here $\perp$ denotes an unresolved state when there is no unique active fact. The history template retains all eligible facts, including superseded ones; in Figure 2, `D1` combines the banking fact `F1` and teaching fact `F2` into Joanna's career history. Both templates preserve supporting fact identifiers, dialogue references, and applied computation rules as provenance.

Memory Index Construction. The *code library* provides executable memory, but downstream retrieval also needs to discover its contents and determine how to use them. We therefore construct the *memory index* I with two parts: indices over library contents and interfaces to memory operators.

For the library contents, we build indices at two levels. The *function index* I_{func} maps each base or derived function identifier to its entry point, description, and accepted arguments, extracted from function metadata. The *view index* I_{view} maps each concept or session view in $\mathcal{V}$ to the identifiers of the functions it references. Together, these indices support navigation from views to callable functions without inspecting their implementations.

We define four categories of memory operators for downstream computation. The *operator index* I_{ops} records their interfaces. *Memory access* retrieves facts and invokes functions; *context expansion* follows views and relations to gather additional facts and source context; *evidence composition* combines, filters, ranks, and aggregates evidence; and *verification* provides heuristic checks of evidence completeness and unresolved states. The content indices and operator interfaces together form $I = (I_{\text{view}}, I_{\text{func}}, I_{\text{ops}})$, exposing both available memory resources and the operations for using them. Table 6 in Appendix A.2 provides detailed operator descriptions.

2.3 Declarative Retrieval: Query-to-Program

To adapt retrieval to diverse queries, *declarative retrieval* interprets each query as a high-level specification and compiles it into a structured retrieval program that selects and composes the required memory computations. We first describe declarative query planning (§2.3.1), then present retrieval program execution (§2.3.2).

Retrieval Program. Given a query q and codified memory $\mathcal{M} = (\mathcal{L}, I)$, we define a *retrieval program* P_q as an executable query plan over the memory operators exposed by I. The program specifies operator arguments and dependencies, together with any conditional continuations. These operators encapsulate memory access and computation, allowing the program to compose existing memory functions and process their results through predefined operations. Executing P_q over $\mathcal{M}$ produces the supporting evidence E_q specified in the problem definition.

2.3.1 Declarative Query Planning

Diverse queries require retrieval plans tailored to their information needs. A question about a current attribute may need a single function call. Questions combining facts across sessions may require context expansion and evidence composition. We use an LLM planner to interpret each query and reason about the required memory operations.

Initial retrieval selects a set of candidate function identifiers K_q using the memory index I built during construction. Their entries in I_{func} and the operator interfaces in I_{ops} form the planning context $I_q(\mathcal{M})$. Function entries describe what each callable computes and which arguments it accepts. Operator interfaces specify the available operations and their input and output constraints. This compact context supports planning without loading the full *code library*.

Our structured prompt presents the query, function information, operator constraints, and planning guidance (Appendix B). The guidance illustrates how query requirements translate into operations, such as temporal filtering or expansion across sessions. The prompt also specifies evidence compatibility constraints and requires the program to return supporting evidence. The LLM uses the query's constraints to select relevant functions and operators, then links their outputs to subsequent inputs.

The planner combines these inputs to produce a complete retrieval program $P_q = F_{\text{plan}}(q, I_q(\mathcal{M}))$ in one planning call. In restricted Python, it specifies operator calls, arguments, dependencies, and conditional retrieval actions based on intermediate evidence.

2.3.2 Retrieval Program Execution

Before execution, MaC checks the retrieval program's structural compatibility with the available memory operations, including permitted language constructs, operator calls, variable references, and evidence dependencies. These dependencies connect function invocation, any selected view expansion, and subsequent evidence processing, preserving the information required by each operation. The query determines which computations and conditional continuations the program combines.

Table 1: Performance across different question types on LoCoMo. F1 is reported in percent, and LLM-Judge accuracy (J) is reported as mean ± standard deviation in percent over five independent judge evaluations. Improv. denotes the relative improvement of MaC over each corresponding baseline in Overall J. **Bold** and underline mark the best and second-best results.

Model	Method	Multi-hop		Temporal		Open-domain		Single-hop		Overall		Improv.
		F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	
Qwen 3.6-27B	A-Mem	39.44	57.73$_{\pm 2.92}$	37.16	45.23$_{\pm 3.39}$	20.13	45.00$_{\pm 0.87}$	54.40	68.80$_{\pm 1.72}$	45.93	60.38$_{\pm 0.80}$	+43.82%
	MemoryOS	29.42	42.06$_{\pm 2.17}$	35.96	22.12$_{\pm 0.58}$	23.43	43.96$_{\pm 1.71}$	38.93	50.20$_{\pm 0.96}$	35.60	42.47$_{\pm 1.03}$	+104.47%
	Mem0	43.67	71.06$_{\pm 2.19}$	60.27	81.68$_{\pm 0.14}$	30.53	56.88$_{\pm 0.93}$	53.93	75.70$_{\pm 0.35}$	51.91	74.92$_{\pm 0.28}$	+15.91%
	MRAgent	48.88	75.04$_{\pm 2.49}$	58.36	71.03$_{\pm 0.38}$	28.19	57.92$_{\pm 1.58}$	55.01	70.11$_{\pm 0.69}$	52.92	70.44$_{\pm 0.81}$	+23.28%
	MaC	**56.20**	**87.09**$_{\pm 0.74}$	**65.78**	**83.93**$_{\pm 0.87}$	**37.36**	**63.75**$_{\pm 0.47}$	**73.49**	**90.51**$_{\pm 0.41}$	**66.46**	**86.84**$_{\pm 0.35}$	–
ChatGPT 5.6-Luna	A-Mem	41.59	60.43$_{\pm 1.24}$	54.26	62.55$_{\pm 0.81}$	25.32	51.46$_{\pm 2.03}$	53.23	70.06$_{\pm 0.95}$	49.58	65.57$_{\pm 0.94}$	+33.69%
	MemoryOS	32.45	58.65$_{\pm 1.11}$	42.22	38.69$_{\pm 0.14}$	22.01	50.00$_{\pm 1.47}$	41.64	68.80$_{\pm 1.05}$	38.85	59.49$_{\pm 0.75}$	+47.35%
	Mem0	40.96	70.78$_{\pm 3.09}$	55.25	73.96$_{\pm 0.28}$	28.83	57.50$_{\pm 1.36}$	52.54	88.09$_{\pm 0.26}$	49.51	80.06$_{\pm 0.69}$	+9.49%
	MRAgent	51.86	71.56$_{\pm 1.51}$	67.64	76.82$_{\pm 1.28}$	38.60	60.21$_{\pm 1.71}$	**69.15**	86.66$_{\pm 0.50}$	63.76	80.19$_{\pm 0.77}$	+9.32%
	MaC	**52.41**	**82.20**$_{\pm 0.68}$	**69.81**	**86.29**$_{\pm 0.44}$	**39.78**	**64.58**$_{\pm 1.28}$	68.41	**92.65**$_{\pm 0.21}$	**63.99**	**87.66**$_{\pm 0.15}$	–
DeepSeek V4-Flash	A-Mem	41.63	63.69$_{\pm 1.89}$	37.58	42.74$_{\pm 2.07}$	25.15	52.92$_{\pm 1.71}$	55.88	77.53$_{\pm 1.33}$	47.54	66.21$_{\pm 1.40}$	+31.99%
	MemoryOS	27.26	52.06$_{\pm 2.72}$	32.80	32.09$_{\pm 4.70}$	22.96	50.83$_{\pm 2.00}$	40.84	66.06$_{\pm 0.96}$	35.57	55.47$_{\pm 2.11}$	+57.54%
	Mem0	39.44	76.45$_{\pm 0.89}$	58.09	83.49$_{\pm 0.00}$	34.16	**65.21**$_{\pm 0.93}$	50.54	82.28$_{\pm 1.07}$	49.06	80.40$_{\pm 0.79}$	+8.69%
	MRAgent	48.02	76.10$_{\pm 1.47}$	69.67	81.74$_{\pm 0.28}$	27.92	55.42$_{\pm 0.87}$	66.45	90.11$_{\pm 0.35}$	61.35	83.64$_{\pm 0.49}$	+4.48%
	MaC	**52.99**	**86.81**$_{\pm 0.30}$	**70.48**	**84.74**$_{\pm 0.00}$	**40.01**	64.58$_{\pm 0.00}$	**74.07**	**91.20**$_{\pm 0.12}$	**67.34**	**87.39**$_{\pm 0.03}$	–

Table 2: Performance across different question types on LongMemEval using the Qwen backbone. F1 and LLM-Judge accuracy (J) are reported in percent. Sgl.: single-session-user; Tmp.: temporal-reasoning; Pref.: single-session-preference; Upd.: knowledge-update. Bold indicates the best result, and underline indicates the second best in each column.

Method	Sgl.		Tmp.		Pref.		Upd.		Overall		Improv.
	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	
A-Mem	71.74	84.29	32.41	36.84	7.05	50.00	61.81	73.08	47.16	59.40	+25.59%
MemoryOS	70.69	80.00	30.63	32.33	6.43	63.33	54.99	69.23	43.33	54.20	+37.64%
Mem0	82.63	92.86	34.62	54.89	**19.15**	70.00	62.71	76.92	50.95	68.40	+9.06%
MRAgent	73.87	88.57	34.03	56.39	6.94	53.33	43.21	53.85	43.67	61.20	+21.90%
MaC	**84.28**	**94.29**	**53.28**	**68.42**	14.49	**73.33**	**69.69**	**87.18**	**58.58**	**74.60**	–

Once accepted, P_q executes over the code library $\mathcal{L}$, with function calls resolved through the memory index I. Calls to base and derived functions reuse the fact definitions and state computations established during construction, while evidence composition preserves their fact identities and sources. Intermediate results determine which conditional continuations already specified in P_q are followed. Execution can therefore perform additional memory access when indicated by these conditions without a new LLM planning decision. This planning–execution separation combines operator composition tailored to each query with reusable memory computation.

The final evidence assembly combines selected facts, supporting dialogue, and computation provenance into E_q, loading source context as specified by the program; execution also produces a trace $\mathcal{T}_q$. A separate answer model generates the response as $a_q = F_{\text{answer}}(q, E_q)$. A valid first attempt uses one planning call and a separate answer-generation call; checking or execution failures may trigger bounded repair attempts, followed by a fixed retrieval fallback if needed. Algorithm 1 in Appendix C summarizes construction, retrieval, and answer generation using the same notation.

2.4 Complexity Analysis

One-time construction costs $T_{\text{build}} = O(h \log h + n^2 \log n + \mathcal{C}_{\text{build}})$, where n and h are the number of extracted facts and dialogue length in tokens and $\mathcal{C}_{\text{build}}$ aggregates construction-stage model inference. Each query costs $T_q = O\big(\mathcal{C}_q + p(n^2 + h + b^2) + p^2\big)$ time for a retrieval program with p syntax-tree nodes and evidence expansions of size up to b, with $\mathcal{C}_q$ capturing planning and answering. Persistent codified memory occupies $O(n + h)$ space because stored references satisfy $r = O(n)$, and query processing adds $O(pb)$ for program intermediates and execution traces. Appendix A.3 states the assumptions and gives the full time and space analysis.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUPS

In this section, we describe the experimental setup and present comprehensive results to systematically evaluate MaC and address the following research questions (RQs):

- **RQ1 (Overall performance)** How does MaC perform against representative long-term memory systems across benchmarks, question types, and LLM backbones?
- **RQ2 (Cost, runtime, and storage)** What are the token consumption, wall-clock runtime, and storage footprint of MaC during memory construction and retrieval?
- **RQ3 (Ablation study)** How do executable memory and programmatic search individually and jointly contribute to MaC's performance?

Datasets. We evaluate MaC on two benchmarks for long-term conversational memory: LoCoMo (Maharana et al., 2024), which evaluates question answering over long conversations, and LongMemEval (Wu et al., 2024), which evaluates memory across multiple sessions with longer interaction histories per query.

Baselines. We compare MaC with representative baselines: A-Mem (Xu et al., 2026b), MemoryOS (Kang et al., 2025), Mem0 (Chhikara et al., 2025), and MRAgent Ji et al. (2026).

Implementation Details. We conduct our experiments on NVIDIA A100 GPUs. On LoCoMo, we evaluate all methods using three LLM backbones: Qwen3.6-27B, ChatGPT-5.6-Luna, and DeepSeek-V4-Flash. On LongMemEval, we use Qwen3.6-27B as the backbone for all methods. Following prior work, we report F1 and LLM-Judge (J) scores. LLM-Judge (J) scores are obtained using DeepSeek-V4-Flash. Our code is available at:`https://anonymous.4open.science/r/MemoryasCode-8C7B/MaC`.

3.2 MAIN RESULTS (RQ1)

To evaluate the effectiveness of MaC, we compare it with four representative memory systems on LoCoMo and LongMemEval, with three generation backbones evaluated on LoCoMo. Tables 1 and 2 summarize the results, from which we draw two main observations.

Overall performance. Tables 1 and 2 show that MaC achieves the highest overall J in all four evaluation settings. On LoCoMo, it outperforms the strongest baseline for each backbone by 3.75–11.92 percentage points in overall J. On LongMemEval, it improves overall F1 and J by 7.63 and 6.20 points, respectively.

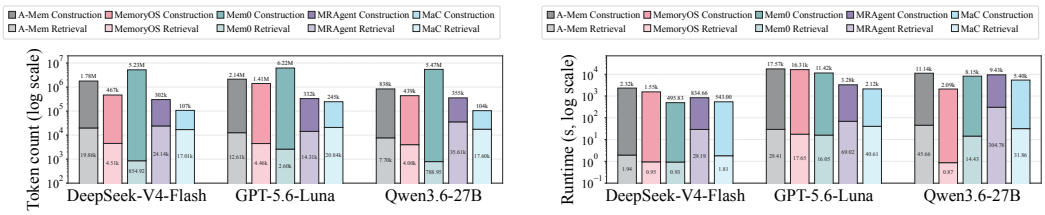
Temporal reasoning and knowledge updates. MaC achieves the best temporal F1 and J across all three LoCoMo backbones. On LongMemEval, it surpasses the strongest respective baselines by 18.66 points in temporal F1, 12.03 points in temporal J, and 10.26 points in knowledge-update J, highlighting its effectiveness on queries involving temporal constraints and evolving facts.

3.3 COST AND EFFICIENCY ANALYSIS (RQ2)

We report construction and query-processing costs separately on LoCoMo in Figure 3, and their combined per-sample costs on LongMemEval in Table 3.

Token consumption. Figure 3(a) shows that MaC requires the fewest tokens for memory construction across all three backbones on LoCoMo. This demonstrates that building an executable memory library and its interface index can be achieved with a lower construction token budget than the evaluated baselines. The advantage is most consistent during construction, while query-stage token consumption varies across backbones and methods. On LongMemEval, MaC also achieves the lowest combined token consumption, using approximately 173k tokens per sample compared with 249k for MRAgent, a reduction of approximately 30.5%.

Runtime. Figure 3(b) shows that MaC achieves competitive runtime across all three backbones on LoCoMo. It ranks first or second in memory construction time and consistently achieves lower



(a) Token Consumption **(b) Runtime**

Figure 3: Normalized token consumption (a) and runtime (b) on LoCoMo across different generation models. Construction is normalized by 10 conversations and retrieval by 1,540 questions. Both panels use logarithmic vertical axes.

Table 3: Per-sample token consumption and runtime on LongMemEval across different memory systems using the Qwen backbone. Results include both memory construction and retrieval. **Bold** indicates the best result, and underline indicates the second best.

Method	Token Consumption ↓	Runtime (s) ↓
A-Mem	1,723k	16,417.98
MemoryOS	636k	11,256.63
Mem0	4,812k	11,952.47
MRAgent	249k	**2,963.90**
MAC	**173k**	4,663.66

query latency than MRAgent. On LongMemEval, Table 3 shows that MaC achieves the second-lowest end-to-end runtime among the evaluated methods, alongside the lowest token consumption. Together with the highest overall LLM-judge accuracy reported in Table 2, these results show that MaC maintains competitive computational efficiency while providing effective memory-based reasoning.

Storage space. As shown in Table 4, MaC maintains a competitive storage footprint among the methods evaluated on LoCoMo. MaC requires 2,859.18 KB of total storage, a 33.5% reduction compared with MRAgent, a strong baseline. Although A-Mem has a smaller storage footprint (1,338.88 KB), it achieves substantially lower accuracy than MaC under the same setting.

3.4 Ablation Study (RQ3)

Table 5 reports sequential ablations on LoCoMo with DeepSeek-V4-Flash. MaC w/o DR retains the codified library but uses a fixed retrieval program; MaC w/o DR+CC applies that program to extracted atomic facts only.

Effect of *declarative retrieval*. MaC w/o DR keeps the codified library but uses a fixed retrieval program that combines lexical recall over facts with execution of candidate memory functions through predefined memory operators. Enabling DR improves overall J by 6.62 percentage points, demonstrating the benefit of adapting retrieval operations and their composition to each query.

Effect of *codified construction*. Under fixed retrieval, CC improves overall J by 8.05 percentage points, supporting the value of reusable memory computations and indexed access. Removing both DR and CC applies the same fixed program to extracted atomic facts only and reduces overall J by 14.67 percentage points relative to full MaC.

4 Related Work

Memory Construction. Memory construction concerns retaining useful information from growing interaction histories and revising it as new interactions arrive. A related foundation is retrieval-augmented generation (RAG) (Lewis et al., 2020), which indexes external passages as dense vectors for generation; GraphRAG (Han et al., 2024) augments this setting with graph structure, community summaries, and neighborhood-based access. For persistent conversational memory, Mem0 (Chhikara et al., 2025) extracts salient facts and reconciles them with existing memory,

Table 4: Storage space usage on LoCoMo (KB). Construction storage is averaged per conversation, and retrieval storage is averaged per question. **Bold** indicates the best result, and underline indicates the second best in each column.

Method	Construction ↓	Retrieval ↓	Total ↓
A-Mem	**1,161.82**	177.06	**1,338.88**
MemoryOS	11,452.02	415.59	11,867.61
Mem0	7,105.06	**13.20**	7,118.25
MRAgent	3,870.32	428.13	4,298.46
MaC	2,565.79	293.39	2,859.18

Table 5: Sequential ablation of declarative retrieval (DR) and codified construction (CC) on LoCoMo with DeepSeek-V4-Flash. F1 and binary LLM-judge accuracy (J) are percentages. Bold indicates the best result in each column. We use ↑ to indicate an improvement in the performance of one module, and ↑↑ to indicate improvements in two modules.

Method	Multi-hop		Temporal		Open-domain		Single-hop		Overall		Improv.
	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	F1 ↑	J ↑	
MaC w/o DR+CC	44.01	70.92	50.30	64.80	36.97	58.33	61.09	78.00	54.21	72.73	+20.17%
MaC w/o DR	50.54 ↑	85.46 ↑	59.63 ↑	69.78 ↑	39.00 ↑	62.50 ↑	67.15 ↑	85.49 ↑	60.79 ↑	80.78 ↑	+8.20%
MaC	**52.99** ↑↑	**86.52** ↑↑	**70.48** ↑↑	**84.74** ↑↑	**40.01** ↑↑	**64.58** ↑↑	**74.07** ↑↑	**91.32** ↑↑	**67.34** ↑↑	**87.40** ↑↑	–

whereas MemoryOS (Kang et al., 2025) organizes information across short-, mid-, and long-term stores. Other methods emphasize relational and temporal structure: A-Mem (Xu et al., 2026b) dynamically links and updates structured notes, and Zep (Rasmussen et al., 2025) records fact validity and historical changes in a bi-temporal graph. More recently, MemSkill (Zhang et al., 2026) learns to select and evolve memory-construction skills. Collectively, these methods improve information selection, consolidation, and structural organization through different memory representations and construction strategies.

Memory Retrieval. Memory retrieval addresses both locating relevant records and deciding how further evidence should be accessed and composed. SeCom (Pan et al., 2025) improves retrieved context through topic-based segmentation and compression-based denoising. Structured retrieval extends access beyond flat text matching: Zep (Rasmussen et al., 2025) combines semantic and full-text search with graph traversal and reranking. In external-knowledge search, Search-o1 (Li et al., 2025) integrates on-demand search into document-based reasoning, and Search-R1 (Jin et al., 2025) uses reinforcement learning to learn multi-turn search behavior. At the execution-interface level, Text2Mem (Wang et al., 2026) translates memory instructions into typed, validated operations and supports workflows with dependencies across backends. MRAgent (Ji et al., 2026) uses intermediate evidence to guide exploration and pruning over a Cue–Tag–Content graph. These approaches improve context quality, structured access, or adaptive exploration; nevertheless, candidate-based retrieval remains bounded by the records selected for access, while iterative reasoning can require repeated context processing and model calls. Thus, retrieving sufficient evidence under a limited budget remains a central challenge.

5 Conclusion

We presented MaC, a unified framework that jointly designs memory construction and retrieval. We develop two complementary techniques: *codified construction* organizes facts, their evolving relations, and reusable computations into a three-layer executable memory structure. *declarative retrieval* compiles queries into complete programs that select and compose these computations, enabling query-specific conditional execution without stepwise LLM planning. Experiments on LoCoMo and LongMemEval show that MaC achieves the highest overall F1 and LLM-judge scores across the evaluated settings while substantially reducing token consumption.

AI Assistance Disclosure

Large Language Models (LLMs) were used to aid in the writing and polishing of the manuscript. Specifically, we used an LLM to assist in refining the language, improving readability, and ensuring clarity in various sections of the paper. The model helped with tasks such as sentence rephrasing, grammar checking, and enhancing the overall flow of the text.

All AI-assisted revisions were reviewed and approved by the authors. The research was conceived and conducted by the authors, who take full responsibility for the accuracy, originality, and integrity of the manuscript.

References

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. In *European Conference on Artificial Intelligence*, 2025.

Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*, 2024.

Shuo Ji, Yibo Li, and Bryan Hooi. Memory is reconstructed, not retrieved: Graph memory for llm agents. *arXiv preprint arXiv:2606.06036*, 2026.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25972–25981, 2025.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5420–5438, 2025.

Lei Liu, Xiaoyan Yang, Yue Shen, Binbin Hu, Zhiqiang Zhang, Jinjie Gu, and Guannan Zhang. Think-in-memory: Recalling and post-thinking enable llms with long-term memory. *arXiv preprint arXiv:2311.08719*, 2023.

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.

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.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Xufang Luo, Hao Cheng, Dongsheng Li, Yuqing Yang, Chin-Yew Lin, H Vicky Zhao, Lili Qiu, et al. On memory construction and retrieval for personalized conversational agents. *arXiv preprint arXiv:2502.05589*, 2025.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

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

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, et al. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8416–8439, 2025.

Leo Wang, Lihai Yang, Boyu Chen, Kerun Xu, Gongyi Zou, Bo Tang, Feiyu Xiong, Siheng Chen, and Zhiyu Li. Text2mem: A unified memory operation language for memory operating system. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2105–2119, 2026.

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

Buqiang Xu, Yijun Chen, Jizhan Fang, Ruobin Zhong, Yunzhi Yao, Yuqi Zhu, Lun Du, and Shumin Deng. Structmem: Structured memory for long-horizon behavior in llms. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 122–146, 2026a.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

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

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

A Appendix

A.1 Notation

The table below follows the problem definition and summarizes the notation for memory construction, retrieval, and complexity analysis. MaC constructs $\mathcal{M}$ from dialogue history $\mathcal{H}$, executes a program P_q generated for query q to assemble evidence E_q, and generates an answer a_q while recording an execution trace $\mathcal{T}_q$. Local variables used only in Algorithm 1 are introduced alongside the algorithm.

Notation used throughout the paper

Symbol	Meaning
Task notation	
$\mathcal{H}$, s_j, n_{sess}	Dialogue history, its jth session, and the number of sessions.
$\mathcal{M}$	External memory constructed from $\mathcal{H}$.
q, E_q, a_q	Query, supporting evidence assembled for it, and the generated answer.
Codified memory	
$\mathcal{L}$, I	Code library and memory index, with $\mathcal{M} = (\mathcal{L}, I)$.
$\mathcal{F}$, $\mathcal{V}$	Base fact functions and views referencing base and derived functions.
$\mathcal{R}$, $\mathcal{D}$	Explicit memory relations and derived memory functions.
I_{view}, I_{func}, I_{ops}	View-to-identifier mapping, function descriptors, and operator interfaces.
i, f_i	Canonical fact identifier and its base fact function.
e_i, v_i, S_i	Evidence record, structured value, and supporting source references for fact i.
G, $G(\tau)$, $A_G(\tau)$	Linked state group, its temporally eligible fact identifiers, and its active fact identifiers.
τ, $\perp$	Requested time boundary and the marker for an unresolved state.
Declarative retrieval	
K_q	Candidate function identifiers supplied to the planner for q.
$I_q(\mathcal{M})$	Compact index context supplied to the planner for q.
P_q, $\mathcal{T}_q$	Retrieval program for q and its execution trace.
F_{plan}, F_{answer}	Model stages for program generation and evidence-grounded answering.
Complexity analysis	
n, h	Number of facts before deduplication and source-dialogue length in tokens.
p, b	Syntax-tree size of P_q and maximum expanded evidence size, counting scalar entries and text tokens.
r	Stored references across views, relations, derived functions, and source annotations.
T_{build}, T_q	Construction time and total processing time for query q.
$\mathcal{C}_{\text{build}}$, $\mathcal{C}_q$	Model inference costs during construction and query processing.

A.2 Memory Operators

Table 6 summarizes the operators exposed to MaC search programs.

Table 6: Memory operators exposed to MaC search programs.

Category	Core Operation	Description
Memory Access	`resolve`	Executes selected memory functions.
	`match`	Matches candidate facts to query keywords without code retrieval.
	`search`	Ranks and retrieves canonical facts using a natural-language query.
	`select`	Filters facts by structured constraints.
	`search_functions`	Finds memory functions matching the query constraints.
	`execute`	Runs a memory function and returns evidence.
Context Expansion	`expand_scope`	Adds related memories from the same concept or session.
	`expand_relation`	Follows graph links to add temporally or semantically related facts.
	`hydrate`	Loads canonical facts and nearby dialogue for answer context.
Evidence Composition	`union`	Merges evidence with its facts and dialogue sources.
	`rank`	Orders evidence by relevance to the query.
	`where`	Filters evidence by a specified time range.
	`aggregate`	Counts evidence while retaining its facts and sources.
Verification	`validate`	Checks evidence completeness and unresolved states.

A.3 Complexity Analysis

This subsection expands Section 2.4. To characterize the cost of reusable memory computation, we distinguish one-time construction from per-query retrieval and account for model inference separately. Let n be the number of extracted facts before deduplication, so $|\mathcal{F}| \leq n$, and let h be the total length of the source dialogue $\mathcal{H}$ in tokens. For a query q, let p be the number of syntax-tree nodes in P_q and b the maximum fully expanded size of an evidence collection materialized during retrieval, counting scalar entries and text tokens in its values and provenance. We assume bounded fact text, structured values, and query length, bounded annotations per extracted fact, and fixed retrieval and repair limits.

Time Complexity. Construction compares facts for deduplication and scans and sorts candidate neighbors for each canonical fact. The latter costs $O(n^2 \log n)$ in the worst case, even though only a bounded number of neighbors is retained per structural relation. Including source preprocessing and model inference, the construction bound is $T_{\text{build}} = O(h \log h + n^2 \log n + \mathcal{C}_{\text{build}})$. During retrieval, candidate matching scans and sorts the memory index, while view expansion and provenance composition can incur quadratic work through list-based deduplication. Structural program checking takes $O(p)$ time, and the loop-free language permits at most $O(p)$ operator calls. Accounting for memory access, source loading, evidence composition, and trace generation gives the conservative bound $T_q = O\big(\mathcal{C}_q + p(n^2 + h + b^2) + p^2\big)$. Here $\mathcal{C}_{\text{build}}$ and $\mathcal{C}_q$ sum the model inference costs in the corresponding stage, including extraction, semantic compilation, retrieval preprocessing, planning, and answer generation as applicable. This accounting includes both operator execution and the model calls used across the query pipeline.

Space Complexity. Let r count stored references across memory views, fact relations, derived functions, and source annotations. The code library, memory index, construction records, and source dialogue require $O(n + r + h)$ space. A query additionally retains intermediate evidence and its execution trace $\mathcal{T}_q$, giving $O(n + r + h + pb)$ space, excluding model weights and inference workspace. Under the stated annotation bounds and the implemented relation limits, $r = O(n)$, so memory storage is linear in the input size. The pb term accounts for the generated program's intermediate results, including matching pairs materialized by joins and their provenance.

A.4 Case Study (RQ4)

Figure 4 presents a temporal-reasoning case in which the system must connect the user's keyboard-playing activity with a later bluegrass-band discovery across separate dialogue sessions. MaC generates a question-specific search program that retrieves the relevant activity memories, expands their temporal relations to recover the event needed by the question, and restores the associated source dialogue for temporal grounding. From the recovered evidence, the answer model correctly infers an interval of *6 days*; its cited evidence exactly matches the annotated evidence. This example demonstrates how programmatic memory search combines structured retrieval, relation-based expansion, and source-context recovery to answer temporally grounded conversational questions.

B Prompt Details

Box 1: Fact to Code Prompt

Task: Convert canonical facts into structured specifications that a local generator turns into executable Python memory functions.
Input: Canonical facts with identifiers, factual content, subjects, temporal information, and source references.
System prompt:
You are a semantic compiler. Produce one function specification for each fact while preserving its identifier and meaning. Specify a concise Python function name, a query predicate, the factual value, optional attributes, and whether the fact describes a time-varying state.
Use only the supplied facts. Do not invent information or infer state-update links in this stage. The specifications will be converted locally into Python functions.
User prompt template:

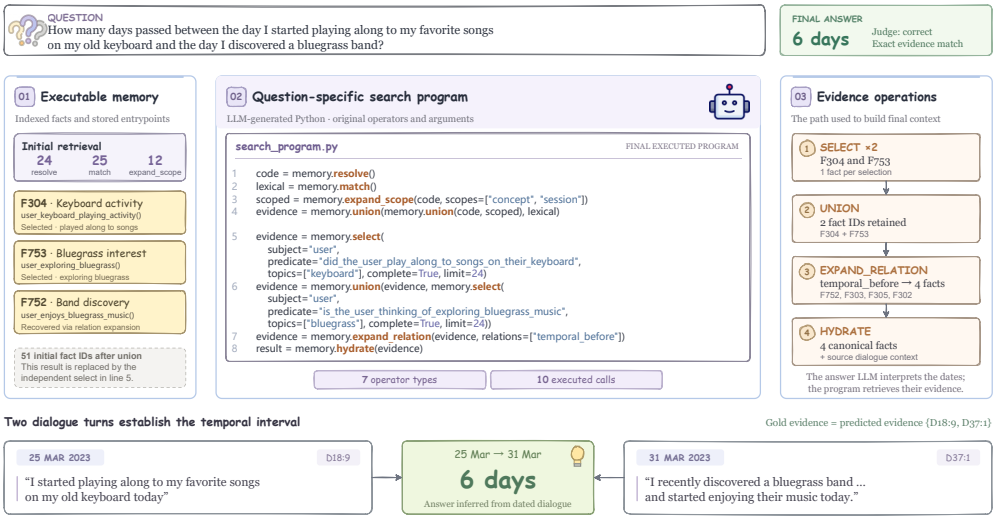


Figure 4: Temporal-reasoning case study on LoCoMo. MaC generates a question-specific retrieval program, expands temporal relations across sessions, and hydrates source dialogue; the answer model infers a *6-day* interval matching the annotated evidence.

Facts:
`{{facts}}`
Compile these facts into function specifications.
Output: A JSON object containing the fact specifications, not Python source code.

Box 2: Declarative Query Planning Prompt

Task: Generate a complete Python retrieval program that selects and composes memory operations for the given question.
Input: The question, compact candidate-function metadata, available memory interfaces, and general human-designed retrieval guidance.
System prompt:
Write one complete program over the read-only executable-memory SDK. Determine the required functions, operations, and dependencies from the question and interface information. Plan the retrieval as a whole rather than requesting further LLM decisions after each operation.
Use the human-designed guidance as reference, not as a fixed pipeline: direct lookup may suffice for a specific fact; distributed evidence may require expansion and composition; temporal questions may require state or history computations and source context. These patterns may not fit every question; adapt or omit them according to the actual evidence requirements.
Use compatible operations to retrieve, execute, filter, and combine evidence as needed. Preserve source information and assign the supporting evidence to `result`. Do not modify memory or generate the final natural-language answer.
User prompt template:
Question:
`{{question}}`
Candidate functions:
`{{functions}}`
Generate a complete retrieval program for this question using the available interfaces.
Output: A Python program whose `result` contains supporting memory evidence.

C Framework Algorithm

Algorithm 1 uses the memory and query notation defined in Section 2. The local collections $\mathcal{X}_{\text{raw}}$ and $\mathcal{X}$ contain extracted and canonical fact records, respectively; a canonical record x_i supplies the value v_i and source references S_i encoded by $f_i \in \mathcal{F}$. For a query q, K_q^{view} contains function identifiers obtained through view expansion, and E_q^{code}, E_q^{view}, and E_q^{lex} denote evidence from code recall, view expansion, and lexical recall. Their merged evidence is cached in $\mathcal{B}_q$; the program returns intermediate evidence E, which is assembled into the final evidence E_q. Here $\mathcal{G}$ denotes the LLM used for extraction, annotation, and repair, while F_{plan} and F_{answer} denote the planning and answering stages defined in the main text. Only values used by subsequent steps are shown.

Algorithm 1: Memory as Code (MaC) Framework

Input: Dialogue history $\mathcal{H} = (s_1, \ldots, s_{n_{\text{sess}}})$, query set Q, LLM $\mathcal{G}$
Output: Codified memory $\mathcal{M}$, answers $\{a_q \mid q \in Q\}$, and traces $\{\mathcal{T}_q \mid q \in Q\}$

▷ *Stage 1: Codified Construction*
$\mathcal{X}_{\text{raw}} \leftarrow \emptyset$
foreach *non-empty dialogue session* $s_j \in \mathcal{H}$ **do**
  Extract source-annotated records with $\mathcal{G}$.ExtractAtomicFacts(s_j) and add them to $\mathcal{X}_{\text{raw}}$
$\mathcal{X} \leftarrow \text{CanonicaliseAndSemanticDeduplicate}(\mathcal{X}_{\text{raw}})$; assign each record x_i a canonical ID i
Store $\mathcal{X}$ in `facts_by_id.json`; initialize $\mathcal{R}$ with temporal, same-subject, and shared-topic relations
Annotate records in $\mathcal{X}$ using $\mathcal{G}$; group state facts by subject and predicate
Use $\mathcal{G}$ to identify explicit state replacements and add $(j, \texttt{supersedes}, i)$ to $\mathcal{R}$
$\mathcal{F} \leftarrow \emptyset$
foreach *canonical record* $x_i \in \mathcal{X}$ **do**
  Add to $\mathcal{F}$ a function f_i returning $e_i = (v_i, \{i\}, S_i)$; emit its applicable `supersedes` references
Construct views $\mathcal{V}$, derived current/history functions $\mathcal{D}$, and code for $\mathcal{R}$
Construct $\mathcal{L} = (\mathcal{F}, \mathcal{V}, \mathcal{R}, \mathcal{D})$, $I = (I_{\text{view}}, I_{\text{func}}, I_{\text{ops}})$, and $\mathcal{M} = (\mathcal{L}, I)$
Persist `memory_code`, `function_index.json`, and `view_index.json`
▷ *Stage 2: Declarative Retrieval*
foreach *query* $q \in Q$ **do**
  $K_q \leftarrow \text{CodeRecall}(q, I_{\text{func}}, 24)$ **in parallel with** $\text{terms}_q \leftarrow \mathcal{G}.\text{ExtractRetrievalTerms}(q)$
  $E_q^{\text{code}} \leftarrow \text{ExecuteAndRehydrate}(K_q, \mathcal{M})$;
   $E_q^{\text{lex}} \leftarrow \text{KeywordRecall}(\text{terms}_q, \texttt{facts_by_id}, 32)$
  $K_q^{\text{view}} \leftarrow \text{ViewExpandAndRank}(K_q, I_{\text{view}}, I_{\text{func}}, q, 32)$;
   $E_q^{\text{view}} \leftarrow \text{ExecuteAndRehydrate}(K_q^{\text{view}}, \mathcal{M})$
  $\mathcal{B}_q \leftarrow \text{Deduplicate}(E_q^{\text{code}} \cup E_q^{\text{view}} \cup E_q^{\text{lex}})$
  Form $I_q(\mathcal{M})$ from $\{I_{\text{func}}(\kappa) \mid \kappa \in K_q\}$ and I_{ops}
  $P_q \leftarrow F_{\text{plan}}(q, I_q(\mathcal{M}))$
  for $\ell \leftarrow 0$ **to** 2 **do**
    if $\text{ValidatePlan}(P_q)$ *and* $\text{RunInNetworkIsolatedSandbox}(P_q, \mathcal{M}, \mathcal{B}_q)$ *succeed* **then**
      $E \leftarrow \text{ExecutionResult}(P_q)$; **break**
    $P_q \leftarrow \mathcal{G}.\text{RepairProgram}(P_q)$
  if *no generated program succeeds* **then**
    $P_q \leftarrow \text{DefaultFallbackProgram}(I_q(\mathcal{M}))$;
     $E \leftarrow \text{RunInNetworkIsolatedSandbox}(P_q, \mathcal{M}, \mathcal{B}_q)$
  $E_q \leftarrow \text{RenderEvidenceContext}(E, \mathcal{B}_q, \mathcal{M})$; $a_q \leftarrow F_{\text{answer}}(q, E_q)$
  if a_q *is malformed, empty, or a refusal* **then**
    $a_q \leftarrow \mathcal{G}.\text{RepairAnswer}(q, E_q, a_q)$
  Record P_q, E_q, a_q, and line-level provenance in $\mathcal{T}_q$
return $\mathcal{M}$, $\{a_q \mid q \in Q\}$, and $\{\mathcal{T}_q \mid q \in Q\}$