aai initiative for applied artificial intelligence

Agentic Memory Realized

From Stateless Models to Cognitive Continuity

A Comparative Study of Memory Architectures for Long Term AI Agents

aai initiative for applied artificial intelligence

2026

Agentic Memory Realized

From Stateless Models to Cognitive Continuity

A Comparative Study of Memory Architectures for Long-Term AI Agents

initiative for applied artificial intelligence

2026

Contents

Foreword 6

About This Whitepaper 8

Key Takeaways 9

Chapter 1 **Why Early AI System Could Not Remember** 10

Chapter 2 **The Evolution of Agentic Memory** 16

Chapter 3 **The Design Problem** 23

Chapter 4 **The Architecture of Agent Memory** 28

Chapter 5 **Measuring What Matters** 38

Chapter 6 **The Experiment** 47

Chapter 7 **Results and Analysis** 55

Chapter 7 **Where Different Architectures Matter** 63

Chapter 9 **Conclusion** 68

References 70

Authors 72

About appliedAI Initiative Gmbh 74

Acknowledgment 75

aci initiative for applied artificial intelligence

Foreword

When we published our whitepaper on Large Language Models Make or Buy Strategy, we were helping organizations navigate their first consequential GenAI decisions. When RAG Realized followed, we gave technical and strategic teams the tools to ground those models in enterprise knowledge. And when Generative AI Agents in Action launched in early 2024, we were among the first to assert, with conviction, that autonomous AI agents would not merely be a feature of the AI roadmap. They would become the primary engine through which enterprises extract transformative, lasting value from artificial intelligence.

That prediction has proven correct.

Through appliedAI's AI Agent Lighthouse Program, we spent 2025 and the beginning of 2026 partnering with European enterprises to move AI agents from controlled pilots into real production environments, redesigning processes, reallocating human effort, and generating measurable efficiency gains across industries from finance and manufacturing to logistics and professional services.

And across every sector, every use case, and every deployment, we encountered the same limiting factor:

MEMORY

Not the language model. Not the interface. Not compute. The agents that faltered, those that stalled before reaching production, or broke quietly within it, shared one architectural weakness: no coherent strategy for how to store, retrieve, and reason over information across time. Without memory, even the most capable agent is, functionally, a chatbot that resets with every conversation. For enterprise use cases where context accumulates, relationships evolve, and decisions build on prior decisions, this is a disqualifying flaw.

Agentic Memory Realized is the direct result of those lessons. It continues our "Realized" series — our commitment to translating the frontier of AI into strategic and engineering guidance that European business leaders can act on. Where RAG Realized connected your agents to knowledge, this paper shows you how to make them retain and build on it over time.

The organizations that invest in memory architecture today are building agent systems that will compound in value over time, learning, adapting, and accumulating institutional knowledge the way the best human colleagues do. Those that neglect it will continue rebuilding the same fragile prototypes, wondering why their agents are expensive to run and impossible to scale.

We wrote this paper to help you build something that lasts.

Mingyang Ma

Head of Agentic AI Solutions Development

aci initiative for applied artificial intelligence

About This Whitepaper

This whitepaper explores why memory has become a core design challenge for long-running AI agents. Its central argument is that memory is not only important, but that choosing the right memory approach depends on the problem an agent is meant to solve.

The paper surveys major memory architectures, explains the tradeoffs behind each, and shows how different designs support different goals, such as factual recall, continuity across workflows, and the preservation of richer contextual information over time.

The comparative experiment is included as a concrete demonstration of this broader point. Rather than standing alone as the purpose of the paper, it serves to illustrate how architectural choices shape agent performance in practice, especially in tasks that require sustained interaction and accumulated context.

How To Use This Paper

Chapter 1-4

Architecture & Design

- Why memory matters for AI agents
- Evolution from stateless to stateful
- Layered memory architecture

Chapter 5-7

Experiment & Results

- LoCoMo-Plus evaluation framework
- Comparative experiment: Mem0, Letta, and Zep
- The cognitive resilience gap

Chapter 8-9

Applications & Outlook

- Where cognitive memory matters most
- Healthcare, finance, education use cases
- Implications for future agent design

Key Takeaways

Memory is an architectural requirement, not a feature

The whitepaper shows that as AI agents move beyond single interactions, memory becomes part of the core system design. It is not just an added capability, but a foundation for continuity over time.

Context replay is not true long-term memory

Early approaches can simulate continuity by reusing prior conversation, but they do not provide durable, reliable memory across extended histories or sessions.

Different memory architectures preserve different kinds of value

Some designs are better at storing explicit facts, while others are better at retaining goals, behavioral patterns, and richer contextual information. The right approach depends on what the agent needs to remember.

Cognitive continuity matters more than simple recall

The whitepaper's central distinction is that storing information is not enough. Effective memory must help an agent recover and use past information in ways that support consistent reasoning and behavior over time.

Memory should be evaluated through sustained interaction

The experiment reinforces that memory systems should be judged not only by isolated recall, but by how well they support performance, coherence, and continuity across longer-running tasks.

aci initiative for applied artificial intelligence

Chapter 1

Why Early AI Systems Could Not Remember

Imagine using an AI assistant throughout your workday. It helps summarize documents, review code, and think through technical problems. In one conversation, you explain your project architecture, describe a deployment issue, and ask for suggestions. The assistant responds helpfully and appears to understand the situation.

The next day, you return and ask a follow up question. The assistant responds as if none of the earlier discussion ever happened.

This was not simply a product flaw or a temporary limitation. It reflected a deeper architectural reality: early AI systems were not designed to preserve memory across interactions. They could process the information provided in a request, but they were not built to carry that information forward over time.

Understanding this limitation is the starting point for understanding why memory became such an important problem in modern AI systems.

Stateless Language Models

The first generation of large language model applications operated as stateless inference systems. A prompt was submitted, the model generated an output, and the transaction ended. On the next request, the model processed a new prompt without any inherent awareness of what had happened before.

At a conceptual level, the interaction could be reduced to a simple pattern:

response = LLM(prompt)

The model could only reason over the information available inside that prompt at inference time. Anything omitted from the input was functionally absent from the model's world.

This follows directly from how transformer based systems were designed.[1] They are highly effective at processing a sequence presented in the current call, but they do not natively preserve persistent state between separate calls.

In practical terms, the model can reason about the text it receives now, but it does not automatically carry knowledge forward to future interactions.

This distinction matters because it separates fluency from memory. A model may sound consistent, informed, and context-aware within a single exchange while still lacking any durable way to retain information across time.

The Context Window as Temporary Working Memory

To make early AI systems feel more coherent across a conversation, developers began reintroducing prior messages into each new prompt. If the assistant needed information from earlier exchanges, the application would resend that material along with the current request.

This approach relies on the **context window:** the amount of text a model can process in a single inference call. As context windows expanded, systems became capable of handling longer conversations, larger documents, and more complex instructions within one request. In practice, this expansion was dramatic: early widely used systems operated with only a few thousand tokens of context, later models extended that capacity to more than one hundred thousand, and eventually some systems pushed beyond the million-token range. These increases made models more useful for extended conversations and large-scale document handling, but did not create durable memory across time .

Expanding the context window did not change the fundamental limitation. It improved short-term recall within an interaction, but it did not create persistent memory across time.

A context window is therefore best understood as temporary working memory rather than long-term memory. On each new request, the system must decide what prior information to include, what to summarize, and what to leave out. The model does not retain the past on its own; it can only reason over whatever reconstructed context it is given in the current call. This analogy was made explicit in MemGPT, which compared the limited context window of language models to the limited capacity of human working memory.[2]

This distinction is important. Early systems could appear coherent across a session, but that coherence depended primarily on prompt reconstruction and application level logic, not on a persistent memory architecture designed for long-term behavior.

The Structural Limits of Context-Based Memory

As developers pushed these systems into real workflows, the limits of context reconstruction became increasingly visible.

First, context overflows. When conversations, documents, or tool traces exceed the available window, earlier information must be removed, summarized, or truncated. The result is selective forgetting. Important project details, prior decisions, or earlier constraints can disappear simply because they no longer fit.

Second, context is not persistent. Session history exists only so long as an application stores and resubmits it. Once the user leaves, switches devices, or starts a new thread, continuity must be recreated externally. Without storage outside the model, the system has no durable memory of prior interactions.

Third, long prompts do not guarantee reliable recall. Even when more text fits into the window, models do not treat all parts of the prompt equally. This limitation was highlighted in *Lost in the Middle: How Language Models Use Long Contexts*, which found that model performance often drops when relevant information appears in the middle of a long context rather than near the beginning or end.[3] In other words, more space does not automatically produce better continuity. It can simply create a larger area in which important facts become harder to retrieve.

It is also worth noting that alternative architectures do not automatically solve this problem. Systems such as Mamba and related state-space models offer linear-time sequence processing and maintain a form of recurrent state, but they still do not provide cross-session memory by themselves.[4] Their internal state resets between separate calls, just as a transformer's key-value cache does. Persistence still requires something outside the model itself.

Together, these limitations revealed an important truth: context alone is an unstable foundation for memory.

Context-based Memory Limits

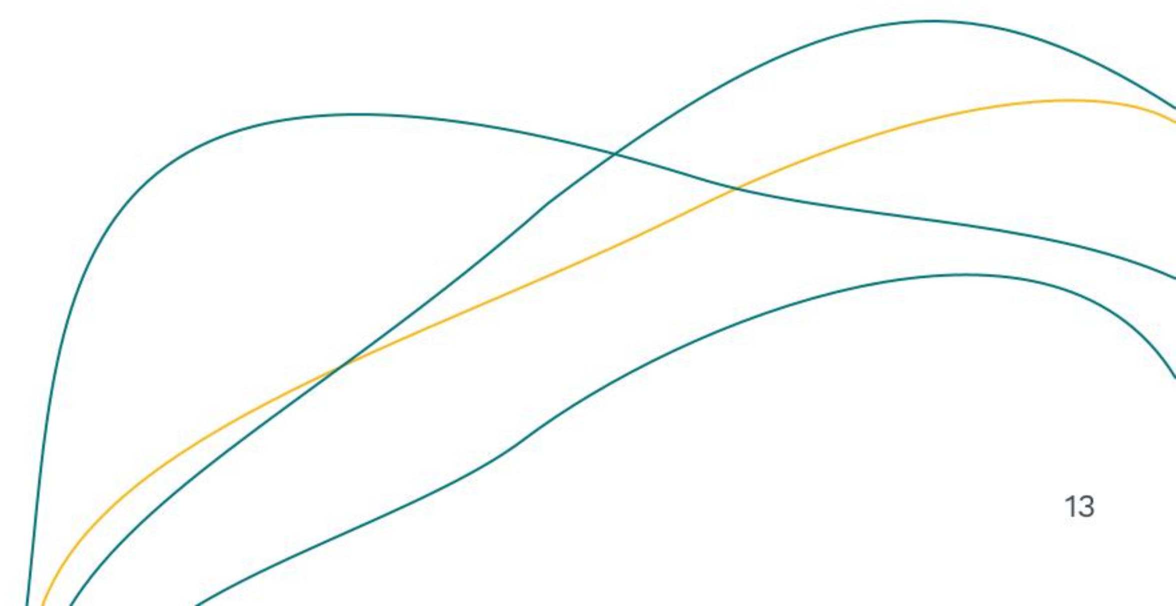
Why This Became a Major Constraint for AI Agents

For narrow, one-off tasks, stateless systems are often sufficient. A model that answers a single question, summarizes a document, or drafts a paragraph may not need to remember anything beyond the immediate request.

That changes once AI systems begin operating as agents rather than isolated responders. An agent is expected to work across multiple steps, maintain objectives, use tools, adapt to intermediate results, and return to ongoing tasks over time. In that setting, memory is no longer a convenience. It becomes essential.

A software engineering agent should remember prior debugging attempts and system constraints. A research agent should retain working hypotheses, source quality assessments, and open questions. A productivity agent should preserve user preferences, recurring priorities, and unfinished tasks. Without memory, each interaction begins too close to zero. The system may remain articulate, but it cannot accumulate understanding.

This is where the limitations of early architectures became operationally significant. What looked acceptable in a chatbot became a serious bottleneck in agentic systems.



AGENT TYPE	MEMORY REQUIREMENTS
Software Engineering Agent	Prior debugging attempts, system constraints
Research Agent	Hypotheses, source quality, open questions
Productivity Agent	User preferences, priorities, unfinished tasks
Customer Success Agent	Communication style, escalation history
Healthcare Companion	Patient concerns, emotional context, treatment history

Table 1.1 — Memory Requirements Across Agent Types

Early Attempts to Compensate

The first wave of solutions treated memory as an application layer workaround. Developers appended chat history, inserted summaries of previous exchanges, and used lightweight frameworks to preserve continuity across short sessions. These techniques were useful and, in many cases, necessary. But they remained extensions around a stateless core.

As systems became more autonomous and workflows stretched over longer time horizons, the limits of these methods became more apparent. Conversation history, summarization, and prompt compression could preserve useful context, and in many systems they remain an important part of the overall design. However, on their own, these techniques often struggle to provide the durable, structured, and selectively retrievable memory required by more demanding long-running agents.

That realization marked an important shift in thinking. Memory could no longer be framed as a prompt engineering trick. It had to be treated as a core systems problem: what should be stored, how should it be retrieved, when should it be updated, and how should it shape future behavior.

This shift set the stage for the next phase in AI system design: the move from temporary context reconstruction to explicit memory architectures.

The Evolution of Agentic Memory

Early language-model applications simulated memory rather than possessing it. When continuity was required, developers inserted previous conversation turns into the next prompt. This improved short-term coherence, but it did not create persistent memory: once the prompt was gone, the system had no durable representation of the past.

As AI systems evolved from conversational interfaces into more autonomous agents, this limitation became much harder to ignore. Agents were expected to pursue goals across multiple steps, use tools, react to intermediate results, and return to unfinished work. In that setting, replaying fragments of conversation was no longer enough. An agent needed to retain information, retrieve it when relevant, and integrate it into later reasoning.

The development of agent memory from roughly 2020 to 2026 can therefore be understood as a transition from prompt reconstruction toward more explicit memory architectures integrated into the design of the agent itself.

Evolution of Agentic Memory

These milestones are not a strict taxonomy, but they clarify the broader shift from temporary context handling toward deliberate memory design.

2020

Retrieval-Augmented Generation (RAG)

Established the pattern of combining language generation with retrieval from an external knowledge source rather than relying only on model parameters.[5]

2021

Early multi-step prompting and prompt chaining workflows

ReAct and early language-agent workflows

Showed why multi-step reasoning, tool use, and action loops require more than single-turn prompt handling.[6]

2022

Generative Agents

Expanded the concept of memory from stored facts to accumulated observations, reflections, and behaviour over time.[7]

2023

MemGPT and memory-layer management

Framed limited context as an architectural problem and introduced explicit management across different memory tiers.[2]

SWE-bench and SWE-agent

Demonstrated that continuity is an operational requirement in realistic software-engineering settings.[8,9]

2024

Emerging adaptive memory systems

Extended the field toward reinforcement learning, belief revision, and broader memory taxonomies.[11,12,13]

2025

Graph-based agent memory (Zep / Graphiti)

Introduced temporally aware graph memory for tracking entities, events, and changing facts over time.[10]

2026

initiative for applied artificial intelligence

Prompt-Level Continuity

The earliest practical technique for maintaining context was conversation replay. Applications stored earlier messages and appended them to new prompts so the model could "see" recent interaction history. Frameworks added buffering, sliding windows, and summarization to make this manageable.

For simple assistants, this was often good enough. It let the model respond as if it remembered what had just been discussed. But the memory remained fragile: the system's apparent recall depended entirely on the context window and on how the application assembled each prompt. Important details could be truncated, summarized away, or omitted.

Conversation replay therefore preserved fragments of dialogue, but it did not create persistent memory.

The Rise of Autonomous Agents

These limits became much clearer when developers began building autonomous agents. Unlike chat assistants, these systems were expected to decompose goals, plan across multiple steps, use tools, and adapt to intermediate results. ReAct is a good early example: it interleaved reasoning traces with actions and showed strong results on ALFWorld and WebShop, making clear that multi-step performance depends on tracking what has already been observed and attempted.[6]

The same pressure appeared in practical software tasks. SWE-bench evaluates agents on real GitHub issues across full repositories, and SWE-agent was built to search, navigate, edit, and test code autonomously inside those environments.[8,9] In settings like these, continuity stops being a UX nicety and becomes an operational requirement: the agent must keep track of what it has inspected, changed, and tried.

Early agent systems also exposed how unstable long horizon behaviour could be without good memory. Projects such as Auto-GPT [14] popularized goal loops and task decomposition, but they also highlighted how easily agents could revisit the same ideas, lose track of prior outcomes, or depend on brittle prompt state.

This is why memory became more than a convenience feature. In Generative Agents, behavior is shaped not only by stored observations, but also by retrieval and higher level reflection over time.[7] The broader lesson was that once agents are expected to operate over extended horizons, **they need a way to accumulate and reuse experience rather than repeatedly reconstructing the present from scratch.**

Retrieval-Augmented Memory

A more durable step emerged when developers began attaching external storage systems to language model applications. Instead of relying only on prompt history, systems could store information independently and retrieve relevant items during later interactions.

This design pattern is closely related to retrieval-augmented generation, or RAG. Lewis et al. introduced RAG as a framework that combines a language model with a retriever accessing an external knowledge index, allowing generation to be informed by retrieved information rather than relying only on parametric memory.[5]

In agent systems, this idea extended beyond document retrieval. Memory stores could hold user preferences, task information, project context, or past events. When a new query arrived, the system could search the memory store for relevant items and inject the results into the model's working context.

This significantly improved continuity across sessions. More importantly, it marked a shift toward treating memory as a distinct system component rather than relying primarily on replaying or compressing prior conversation. At the same time, retrieval-based memory introduced its own limitation: stored information remained useful only if the retrieval mechanism surfaced at the right moment. Relevant context could remain stored but unused.

From Stored Facts to Stored Experiences

A second conceptual shift followed. Researchers and system designers began exploring whether agents should store not only isolated facts, but also experiences.

This changed the role of memory. Instead of preserving only discrete pieces of information, memory could represent events, observations, prior decisions, and patterns of interaction over time. A debugging assistant may need to recall which fixes were already tried and why they failed; a research assistant may need to remember earlier hypotheses and how later evidence changed them.

Generative Agents made this idea especially visible by combining observation, retrieval, reflection, and planning so that later behavior depended on accumulated prior experience rather than on isolated stored facts alone.[7] Facts support recall. Experiences support behavior.

Persistent memory therefore became more than a database optimization. It became a mechanism for maintaining continuity in reasoning and action.

Memory as a Hierarchical System

As memory systems matured, another important design insight emerged: not all memory should be handled in the same way.

The model's context window came to be understood as a form of working memory, useful for immediate reasoning but limited in capacity and duration. External storage served as a longer-term memory layer, preserving information across sessions and over longer time horizons. Some architectures added summaries or specialized stores to bridge the gap between the two.

MemGPT made this shift especially explicit by treating limited context as a systems problem and proposing managed movement across memory tiers rather than assuming that everything relevant should remain in the active prompt at once.[2]

Once memory is framed this way, the design problem becomes more sophisticated. The question is no longer simply what to store, but what belongs in active context, what should remain in persistent storage, what should be abstracted into higher-level knowledge, and what should be forgotten.

Graph Memory and the Move Toward Infrastructure

By the middle of the decade, memory was no longer just an auxiliary feature added to chat applications. It was becoming an infrastructure layer for intelligent systems. Agent frameworks increasingly incorporated persistent state, structured workflows, and mechanisms for maintaining continuity across longer processes and across multiple collaborating agents.

An important part of this shift was the emergence of graph based memory systems. Rather than storing memory only as isolated text fragments or embeddings, these systems began representing entities, events, and their relationships in more structured form. Zep, built on the Graphiti engine, is a useful example: it introduced a temporally aware knowledge-graph approach that can track not only what was stored, but when something happened and how it changed over time.[10] This helps address weaknesses that retrieval-only systems often face, including temporal reasoning, entity resolution, and changing facts across long running interactions.

This change reflected a broader realization across the field: advanced agents cannot rely on prompt tricks alone. They require memory architectures that support persistence, retrieval, abstraction, and adaptation over time. The path from RAG, to ReAct-style agent loops, to reflection-based agents, to tiered memory systems and more structured graph-based memory makes that progression visible.

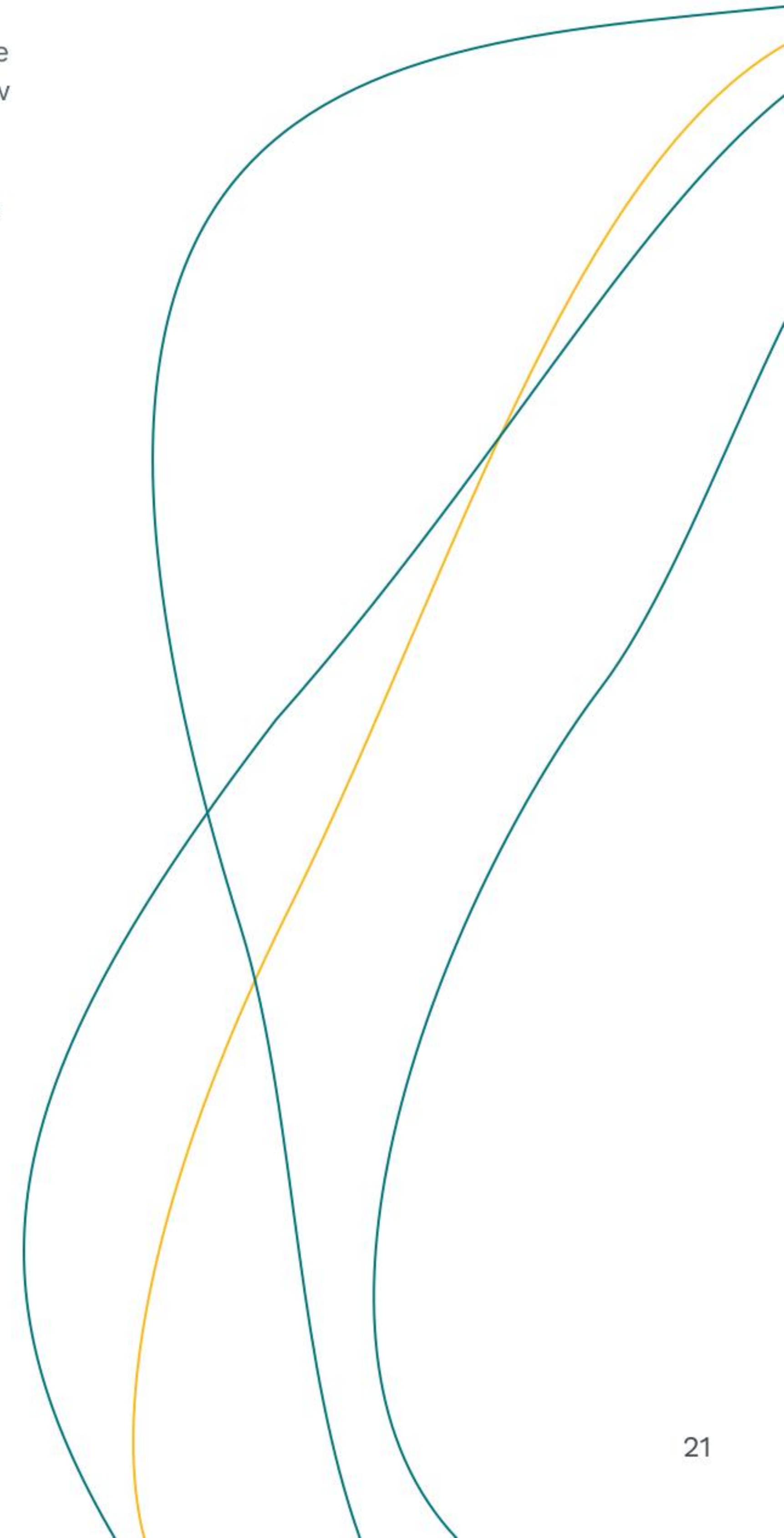
What began as a workaround for stateless models was becoming a foundational component of AI system design.

From Continuity to Architecture

The evolution of agent memory reflects a transition from temporary continuity to durable system design. Early systems simulated memory by replaying conversation history. More advanced systems externalized memory, retrieved it selectively, and increasingly treated it as a structured component of the agent architecture itself.

Memory was no longer just a convenience feature. It was becoming essential to how agents operate. The next challenge, however, was not simply how to store more information. It was how to organize and retrieve information in ways that preserved meaning rather than noise.

That is the design problem addressed in the next chapter.



initiative for applied artificial intelligence

The Design Problem

Introducing persistent storage significantly improved language model continuity by allowing systems to record past interactions, extract information from them, and retrieve relevant context later.

Compared with stateless systems, this was a meaningful advancement. But longer workflows exposed a deeper challenge, storing information is not the same as maintaining usable memory. Systems built mainly around stored text fragments often behaved inconsistently, missed relevant context, or failed to build coherent long term understanding. The design problem was therefore no longer just how to save information, but how to organize and use it so that it supported reasoning over time.

Why Transcript Storage Is Insufficient

Early persistent systems often stored dialogue transcripts or extracted snippets from prior interactions. When a new query arrived, the system retrieved relevant fragments and inserted them into the model's active context.

A transcript preserves what was said, but it does not necessarily preserve what matters.

In long running interactions, only a small fraction of past dialogue remains relevant to future decisions. Without selection, prioritization, and abstraction, a system accumulates text faster than it accumulates understanding.

Retrieval-Based Memory Pipeline

Many memory systems follow a retrieval-based pattern.

This pattern is closely related to retrieval augmented generation, which formalized generation with access to an external retrieved knowledge source [5].

This architecture enables recall across sessions, but as memory stores grow, several structural problems emerge.

Figure 3.1 - Simplified retrieval based memory pipeline

Core Design Problems

Problem	What goes wrong	Why it matters
Fragmentation	Memory is stored as isolated facts or snippets	Correct fragments do not automatically produce coherent understanding
Retrieval Dependence	Stored information is useful only if it is surfaced at the right moment	Relevant memory can remain stored but functionally unused
Context Overload	Too much retrieved material is pushed into active context	More context can reduce clarity and degrade reasoning
Limited Abstraction	Systems preserve details without forming higher level patterns	The model recalls observations but struggles to infer what they mean together
Temporal Blindness	No tracking of when facts were true or how they evolved	Outdated facts can be treated as current, which weakens temporal reasoning
Semantic Drift	Embedding similarity fails when later queries use conceptually related but lexically different language	Relevant memory may remain stored yet still fail to be retrieved

Table 3.1 - Core design problems of retrieval-based memory systems.

The last two problems deserve particular attention because they explain many real world failures. **Temporal blindness** means the system has no reliable way to distinguish a fact that was true a year ago from one that is true today a limitation that temporally aware graph based memory systems such as **Zep** were explicitly designed to address.[10] **Semantic drift** means that when a user describes a long term goal in one conversation and refers to it months later in entirely different language, a retrieval system may fail to connect the two. This is one of the failure modes that newer cognitive memory benchmarks have begun to test more directly [30].

The context overload problem is reinforced by evidence that models often use long contexts unevenly, with performance dropping when relevant information appears in the middle rather than at the beginning or end [3].

A system may separately store that a user prefers Python, works on cloud infrastructure, and values remote collaboration, yet still fails to form a higher level picture of the user's working style. That is the **difference between stored text and usable memory.**

What Effective Agent Memory Must Achieve

1 Persistence
Information survives across sessions and time horizons

2 Selectivity
Trivial observations are filtered out while important patterns are retained

3 Reliable Retrieval
Important memories can be surfaced when needed even when wording changes over time

4 Abstraction
Repeated experiences become higher level knowledge rather than remaining isolated details

5 Adaptation
Memory changes as new situations are encountered and older assumptions become less reliable

6 Temporal Awareness
Facts carry time information so outdated knowledge is tracked rather than silently reused

7 Provenance
Memories remain traceable to their source interactions, improving auditability and trust

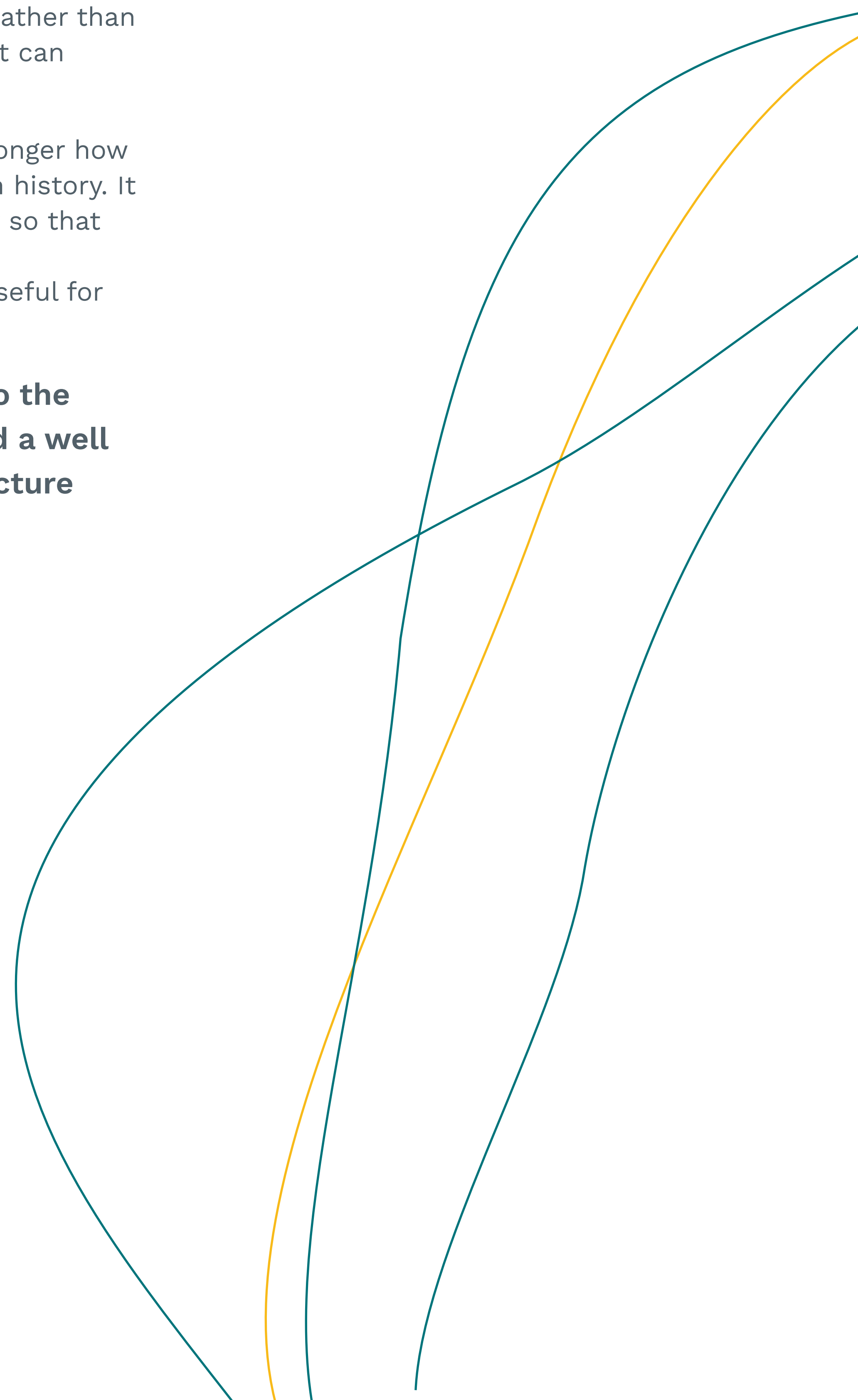
These requirements define the difference between a searchable archive and a functioning memory system.

From Storage to Architecture

As these limits became clearer, memory stopped looking like a simple storage layer and started looking like an architectural problem. Recent frameworks such as CoALA describe language agents in terms of modular memory systems with structured action spaces and decision processes [15] , while systems such as MemGPT [2] frame memory as tier management rather than assuming everything important can remain in the active prompt.

The central question was no longer how to preserve more conversation history. It was how to structure memory so that stored information remained meaningful, retrievable, and useful for future reasoning.

That shift leads directly to the next chapter: what should a well designed memory architecture actually look like?



initiative for applied artificial intelligence

Chapter 4

The Architecture of Agent Memory

By the end of the previous chapter, the central problem had become clear: storing information is not the same as maintaining usable memory. The next question, then, is architectural. How should intelligent agents organize memory so that it remains both durable and usable?

Simple retrieval pipelines helped assistants preserve fragments of prior interaction, but they did not adequately distinguish among the different forms of knowledge required by long running agents. A capable assistant may need to manage immediate conversational context, persistent user facts, records of past events, and learned procedures for solving tasks. Treating all of that information as one flat memory store quickly becomes inefficient and brittle. Modern agent design has increasingly responded to this challenge by adopting structured memory architectures composed of specialized components.

From Storage Layers to Cognitive Organization

The move toward structured memory reflects an important convergence between AI system design and ideas from cognitive science. Human cognition does not rely on a single undifferentiated memory repository. It uses multiple forms of memory that serve different functions in reasoning and action. Agent architectures increasingly borrow this principle. Rather than asking one storage system to handle every kind of information, they divide memory according to purpose. This allows the agent to preserve continuity without overwhelming active reasoning. The result is not simply more memory. It is better organized memory.

A useful conceptual reference here is the **CoALA framework**, which describes language agents in terms of modular memory components, structured action spaces, and decision processes.[15] Its value in this chapter is not that every production system implements CoALA literally, but that it gives a clear vocabulary for thinking about working, semantic, episodic, and procedural forms of memory as distinct parts of an agent architecture.

Memory type	Function	Duration	Agent Example
Working Memory	Active reasoning workspace	Current session	Context window, current prompt, tool outputs
Semantic Memory	Stable factual knowledge	Persistent	User preferences, project configuration, domain facts
Episodic Memory	Records of experiences	Persistent	Past debugging sessions, prior outcomes
Procedural Memory	How tasks are performed	Persistent	Workflows, tool-use patterns, strategies

Table 4.1 — The Four Memory Types in Agent Architecture

This classification follows the modular memory framing used by CoALA [15] and is reflected, in different implementation styles, across current agent systems.

Working Memory

Working memory refers to the information the agent is actively using during the current reasoning process. In most language agent systems, this corresponds to the model's context window or immediate execution state. Everything the model can directly reason over in the present moment must appear there , The current user request, system instructions, retrieved memories, tool outputs, and intermediate state relevant to the task at hand.

Working memory is therefore best understood as the active workspace of the agent. It is essential for immediate reasoning, but it is also limited. Capacity constraints make selection crucial. If too much information is inserted, the workspace becomes noisy. If too little is included, the model reasons from an incomplete picture. This makes working memory central but inherently temporary ,it supports the present moment, but does not by itself provide durable continuity across time.

Semantic Memory

Semantic memory stores stable factual knowledge. In an agent setting, this includes information that remains useful beyond a single interaction: user preferences, project attributes, system configurations, organizational facts, or domain knowledge the assistant should retain. Examples might include a user's preferred programming language, the architecture of an ongoing system, or the deadline associated with an active project. These facts are not tied to one specific moment of interaction. They represent knowledge that should remain available across sessions.

Semantic memory is typically stored in external databases or memory layers and retrieved when relevant. Its role is to preserve durable context that supports future reasoning without requiring full replay of prior conversations. This makes semantic memory a key bridge between stateless model execution and persistent user aware behavior.

Episodic Memory

Where semantic memory stores facts, episodic memory stores experiences. In agent systems, episodic memory captures what happened during prior interactions or workflows: a previous debugging session, a sequence of actions taken to solve a problem, a failed approach, or a meaningful event in the relationship between user and assistant. This form of memory is especially important for long-running agents because it preserves temporal structure. It allows the system to reason not only about what is true, but also about what occurred, in what order, and with what outcome.

That distinction matters operationally. Many tasks depend less on static facts than on prior experience. When an assistant is asked how a similar issue was resolved before, or what happened during the last attempt, semantic knowledge alone is not enough. The system needs access to episodes. In practice, checkpointed states, execution histories, and session summaries often serve as the substrate for this kind of memory. In newer architectures, graph based representations can strengthen episodic memory further by preserving explicit relationships among actors, events, and outcomes across time rather than storing prior episodes only as isolated fragments.[10]

Procedural Memory

Procedural memory stores knowledge about how tasks are performed. In human cognition, this includes learned skills and routines. In agent systems, it appears as workflows, strategies, tool usage patterns, task decomposition methods, and repeated operating procedures. An assistant may develop or be given procedures for reviewing code, diagnosing infrastructure failures, analyzing logs, or coordinating research steps. These procedures are different from facts and different from episodes. They are patterns of action.

This matters because advanced agents are expected not just to answer questions, but to operate. Procedural memory helps convert accumulated experience into repeatable behavior. As agents become more capable, procedural memory may become one of the most important forms of operational improvement. Frameworks centered on flows, routines, reusable tools, and persisted execution patterns increasingly make this layer explicit.

The Logic of a Layered Architecture

When these memory types are combined, a layered architecture begins to emerge. Working memory supports immediate reasoning in the current context. Semantic memory provides stable factual continuity across time. Episodic memory preserves prior events and experiences. Procedural memory guides action through learned or predefined workflows. This layered design allows the system to avoid two failures at once: the statelessness of systems with no persistence, and the chaos of systems that treat all stored information as one undifferentiated mass.

A layered memory architecture also supports more disciplined information flow. Facts can be stored differently from episodes. Repeated episodes can be abstracted into higher level knowledge. Procedural patterns can be refined based on experience. Immediate context can be assembled from multiple memory sources without requiring the entire past to be inserted into every prompt. In other words, **architecture creates manageability**.

Layer/Pattern	Primary Role	Typical Contents	Main Failure
Working Layer	Supports immediate reasoning	Current request, tool outputs, retrieved context, intermediate state	Overload or incomplete reasoning
Semantic Layer	Preserves stable knowledge	User preferences, project facts, system constraints	Stale or noisy facts
Episodic Layer	Preserves prior events and outcomes	Previous attempts, workflows, results, interaction history	Fragmented or poorly retrievable experience
Procedural Layer	Preserves reusable strategies	Workflows, routines, tool-use patterns, decomposition methods	Brittle or outdated behavior
Control Layer	Governs movement across layers	Retrieval, ranking, compression, abstraction, updating	Irrelevant context or missed memories

Table 4.2 - Layered Pattern of Agent Memory

This layered pattern is not a single standard implementation, but it is a useful abstraction for comparing modern systems that separate active context from longer term stored memory and add control mechanisms for retrieval, summarization, and updating.

Architectural Paradigms Evaluated in This Study

The experiment later in this book evaluates three systems that represent three distinct architectural paradigms. Understanding those paradigms helps clarify why memory design cannot be reduced to one universal template.

The first is extraction and retrieval memory, represented by systems such as Mem0. In this design, information is extracted from interactions, stored externally, and later surfaced through retrieval.[16] Its strength is precision .When later queries resemble stored material closely enough, the system can recover useful facts efficiently.

The second is self-managing memory, represented by Letta. Here the agent does not rely only on an external retrieval layer. It also maintains persistent internal structures and participates more directly in deciding what to preserve, compress, or discard over time.[19] This makes the architecture more oriented toward continuity of behaviour and higher level abstraction.

The third is graph based temporal memory, represented by Zep. In this design, memory is not stored only as independent text fragments or vectors, but as linked entities, events, and relationships that can evolve across time.[10] This is especially useful when the agent must track how facts change, distinguish past from present, or reason over connected events rather than isolated snippets.

These paradigms should not be treated as rigid categories. Real systems often combine elements from more than one approach. Even so, the distinction is useful because different architectures optimize for different memory properties: precision of retrieval, continuity of behavior, temporal reasoning, inspectability, or adaptability over time.

Memory Selection and Control

A layered architecture is only useful if the system can manage transitions between layers effectively. This means deciding what should enter working memory, what should remain in long term storage, what should be abstracted into durable knowledge, and what should be discarded. These control decisions are as important as the storage mechanisms themselves. In practice, this control layer determines whether a memory architecture remains usable as the agent's history grows.

Poor selection leads to familiar failures: irrelevant facts clutter the prompt, important experiences are ignored, useful procedures are never activated, and memory becomes a burden rather than an asset. Well designed systems therefore treat retrieval, ranking, compression, and updating as part of memory architecture, not as peripheral utilities.

Frameworks and Implementation Patterns

Modern agent frameworks increasingly reflect these principles, even when they implement them differently. Some emphasize thread or session scoped working memory, some separate in context memory from archival memory, some expose pluggable retrieval layers, and some keep memory fully inspectable as files in the workspace. The technical infrastructure can vary substantially. The key architectural point is not the specific storage technology. It is the recognition that memory is not one undifferentiated repository, but a coordinated system in which different forms of knowledge are stored, updated, retrieved, and activated according to function.

Modern frameworks implement agent memory in notably different ways. Some rely on vector retrieval over extracted facts, some organize memory as graphs of entities and events, some let the agent manage memory more directly, some keep memory fully inspectable as files, and some expose memory as a pluggable subsystem. The important architectural point is that these systems do not merely store information differently ,they also differ in how they decide what becomes retrievable later.

Table 4.2 compares representative frameworks across these approaches.

Frameworks and Implementation Patterns

Framework	How it stores memory	How it retrieves memory
Vector retrieval → Stores facts, finds them by meaning		
Mem0[16]	Extracts facts from interactions and stores them in an external vector database. Qdrant is the default vector backend when no custom configuration is provided.	Uses semantic retrieval over stored memory and supports scoped recall by user, agent, or session.
CrewAI[17]	Uses a unified memory system built around a single memory layer rather than separate short term, long term, and entity memory components.	Ranks results using a mix of relevance, recency, and importance, with deeper retrieval when initial results are weak.
Graph based → Tracks how facts and relationships change over time		
Zep/Graphiti[10]	Stores memory as a temporally aware knowledge graph that preserves not only facts but also when they were true and how they changed over time.	Combines semantic retrieval, keyword search, and graph traversal over linked entities and events.
Cognee[18]	Builds structured memory graphs from documents, entities, relationships, and summaries rather than storing memory only as flat text fragments.	Supports multiple retrieval modes, including keyword search and graph based traversal, for more structured reasoning over stored knowledge.
Self Managing → The agent decides what to remember		
Letta[19]	Separates persistent in context memory from a larger searchable archive, while also maintaining recall memory for conversation history.	Lets the agent itself participate in deciding what to write, update, summarize, or retrieve.

Framework	How it stores memory	How it retrieves memory
Pluggable → Lets developers choose the memory strategy		
LangGraph[22]	Stores short term state as thread scoped checkpoints and supports separate long term stores for persistent memory.	Leaves retrieval strategy configurable: developers can trim, summarize, or search long term memory depending on the application.
AutoGen[23]	Exposes a memory protocol with swappable implementations, including simple list based memory and vector-backed approaches.	Retrieval can be as simple as replaying stored memory into context or as selective as vector style lookup, depending on developer configuration.
Strands Agents[24]	Combines conversation persistence with optional long term memory through Amazon Bedrock AgentCore Memory.	Supports short term conversation persistence as well as long term retrieval of user preferences, facts, and session summaries.
OpenAI Agents SDK[25]	Provides built in session memory that maintains conversation history across multiple runs within a defined session.	Retrieves prior conversation history for the current session, but does not by itself define broader long term memory beyond sessions.
File-backed → Memory you can open in a text editor		
Claude Code[20]	Stores persistent memory in editable Markdown files such as CLAUDE.md, which are loaded at the start of a session.	Retrieves memory by reading those files directly rather than querying a separate database.
OpenClaw[21]	Uses inspectable Markdown based memory files such as MEMORY.md and related workspace notes.	Uses file-based retrieval over stored notes, including keyword and semantic search across workspace memory.

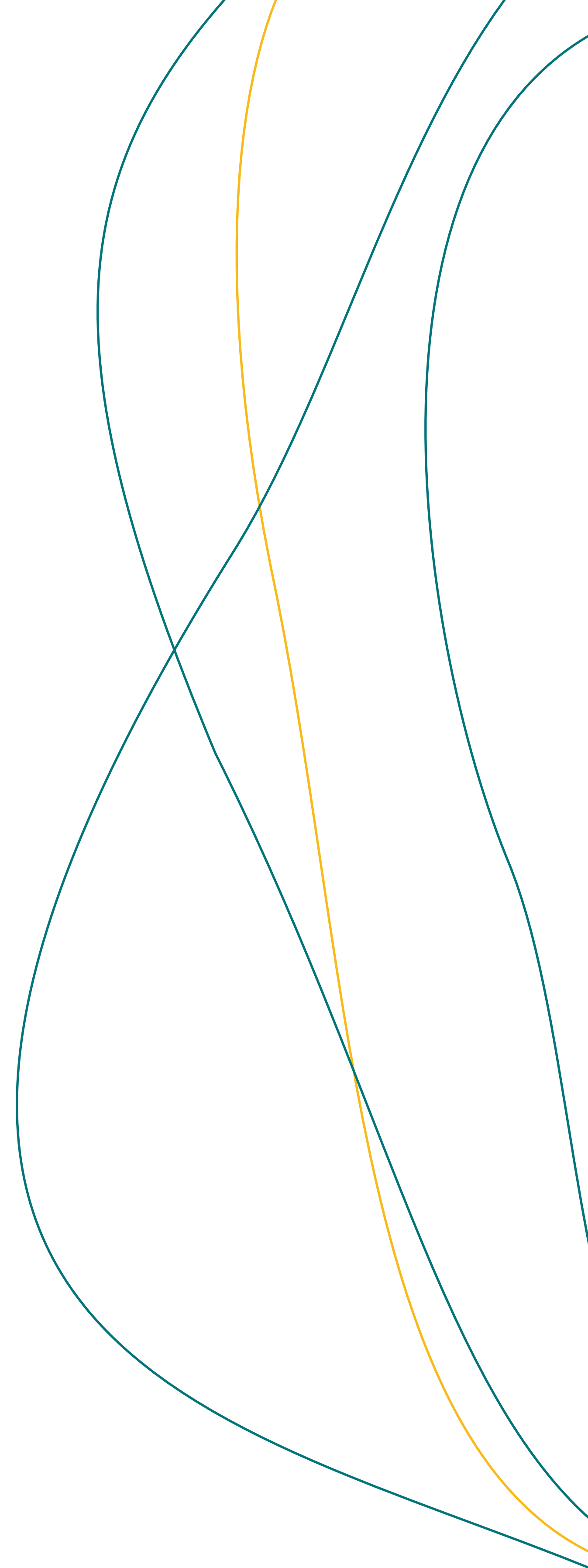
Table 4.2 - Agent Memory Framework Landscape (March 2026). How selected frameworks store and retrieve agent memory, grouped by architectural approach.

Why Architecture Changes the Meaning of Memory

Once memory is organized into functional layers, the role of memory changes. It is no longer simply a repository of past text. It becomes the mechanism through which an agent maintains continuity, accumulates understanding, recalls relevant experience, and translates prior knowledge into present action. That is what separates a memory enabled assistant from a system that merely archives interactions. One stores the past. The other uses the past.

This distinction is central to the future of agent design. As systems become more autonomous and operate across longer horizons, memory architecture will increasingly determine whether they remain useful, adaptive, and trustworthy. Different architectures make different trade offs,
and those trade offs shape what an agent can and cannot remember well.

The next challenge is measurement: if agents can store, retrieve, and organize memory in increasingly sophisticated ways, how should that capability be evaluated in practice?



Chapter 5

Measuring What Matters

The previous chapters traced the evolution of agent memory from stateless prompt processing to layered architectures designed to preserve context over time. That progression raises a practical question: how should memory be evaluated?

It is relatively easy to show that an assistant can store information. It is much harder to determine whether that information remains usable when conversations become long, diffuse, and behaviorally complex.

Consider a user who tells an assistant, early in a relationship, that one of their long term goals is to buy a small cabin near the lake where they grew up. Months later, the same user asks what to do with an unexpected bonus at work. A genuinely useful assistant should recognize that the earlier goal may be relevant to the later financial decision, even though the new question contains no explicit mention of the cabin, the lake, or the original conversation.

This is the difference between simple recall and behaviorally meaningful understanding over time. A memory benchmark that measures only whether information was stored will miss the deeper question of whether the agent can apply earlier context when it matters.

The Benchmark Landscape

Recent research has introduced several benchmarks intended to evaluate memory in conversational and agentic systems. These benchmarks vary not only in scale, but in what they define as successful memory.

Some focus on whether a model can retrieve information from very long contexts. Others examine whether systems can preserve information across multi session dialogue histories. Still others ask whether a system can retain and apply implicit contextual signals such as goals, values, emotional cues, or evolving priorities. These are all useful measures, but they do not test the same capability.

For this study, the key requirement is not simply long context retrieval. It is the ability to preserve and apply implicit user context across extended conversations. That requirement makes the distinction between memory types especially important.

The benchmark landscape can be understood as testing three increasingly demanding levels of behavior.

The first group consists of **long context retrieval benchmarks**. These ask whether a model can locate or recover specific information from very large prompts. Needle-in-a-haystack style tests [32] are the clearest example, and analyses such as Lost in the Middle show that models often use long contexts unevenly, with relevant information in the middle of a prompt proving especially difficult to use [3]. These evaluations are valuable because they reveal the limits of raw context window reasoning, but they mainly test retrieval from a single large input rather than memory across evolving interaction histories.

The second group consists of **conversational memory benchmarks**. These move beyond single long prompts and evaluate whether systems can preserve information across extended multi session dialogue histories. LoCoMo is the clearest example here. Introduced in 2024, it was designed as a benchmark for very long-term conversational memory, with conversations averaging about 300 turns and 9,000 tokens and extending up to 35 sessions. Its tasks include question answering, event summarization, and multimodal dialogue generation, and its results show that even strong models struggle with long-range temporal and causal understanding in dialogue.[27]

Other benchmarks expand this space further. LongMemEval evaluates five core long term memory abilities in chat assistants: information extraction, multi session reasoning, temporal reasoning, knowledge updates, and abstention.[28] MemoryAgentBench broadens the scope from chat assistants to memory agents and evaluates four competencies: accurate retrieval, test time learning, long range understanding, and selective forgetting.[29] Together, these benchmarks make clear that memory quality cannot be reduced to one number or one task type.

The third and most demanding group consists of **cognitive memory benchmarks**. These test whether a system can retain and apply implicit contextual signals such as goals, values, emotional cues, or evolving priorities. This is the level at which memory becomes behaviorally important rather than merely archival. LoCoMo-Plus, introduced in February 2026, was designed specifically for this beyond factual setting.[30] It argues that existing benchmarks focus too heavily on surface level factual recall and introduces a framework for evaluating whether models can retain and apply latent constraints across long conversational contexts.

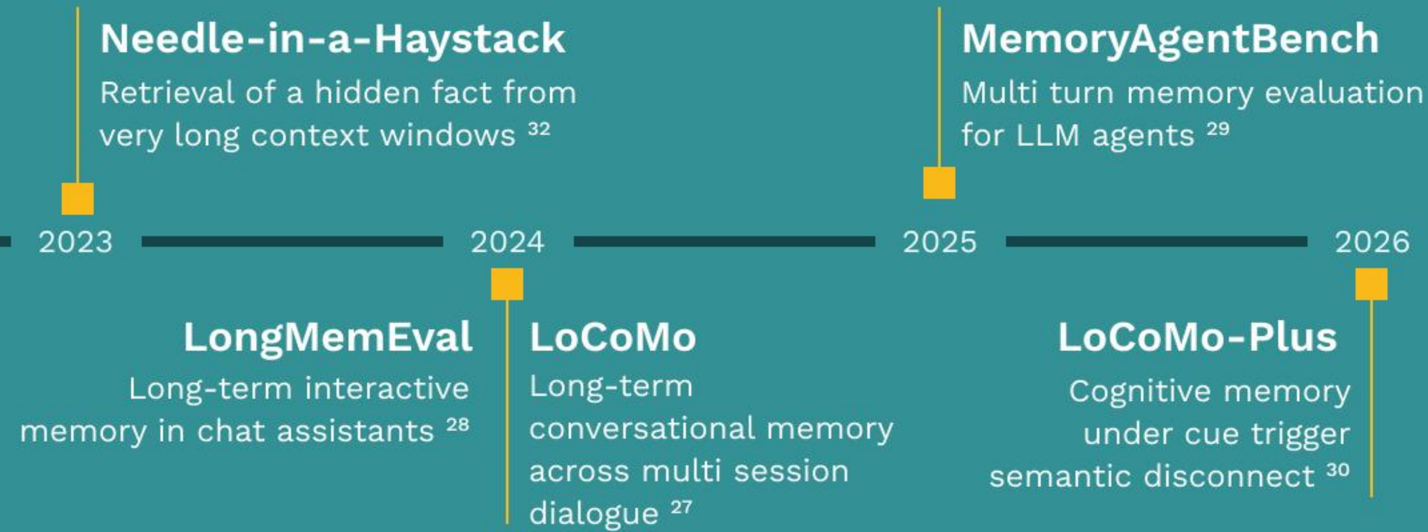


Figure 5.1 - Selected benchmarks for evaluating memory in language models and agents

Factual Memory and Cognitive Memory

A central strength of the LoCoMo-Plus benchmark is that it separates **factual memory** from **cognitive memory**.[30]

Factual memory tests whether a system can recall information that was explicitly stated earlier. Examples include a user's preferred programming language, a stated travel destination, or the deadline associated with a project. These tasks are often well suited to retrieval-based systems because the problem is largely one of finding the correct stored fact and inserting it into context.

Cognitive memory evaluates something more demanding. Instead of asking whether the system can recover an explicit statement, it asks whether the system can retain and apply implicit information that emerges across time. This may include long term goals, values, emotional context, inferred preferences, or causal relationships between earlier events and later decisions. **The critical feature is the cue–trigger semantic disconnect**: the later query does not resemble the original statement closely enough to activate retrieval, yet the earlier context is essential for a good response.[30]

The distinction is consequential. A system may technically possess the relevant information and still fail to use it because the architecture does not preserve it in a form that remains behaviorally available across time. In that case, the memory is present in a technical sense, but functionally unavailable in a behavioral sense.

That is why factual recall alone is an incomplete measure of agent memory.

Example: Factual recall vs. Cognitive memory

A simple comparison makes the distinction clearer. In the earlier conversation, the user mentions a long-term personal goal. Much later, they ask for advice in a way that does not repeat the original wording. A system that only retrieves explicit facts may miss the connection. A system with stronger cognitive memory should recognize that the earlier goal is relevant to the later decision. This is the kind of cue–trigger semantic disconnect that LoCoMo-Plus is designed to test.[30]

Stage	Example conversation	What matters for memory
Early conversation	User: "One of my long-term goals is to buy a small cabin near the lake where I grew up."	The assistant is given a personal goal that may matter later, even if it is not immediately relevant.
Time passes	The conversation continues across many unrelated sessions about work, daily life, and other topics.	The original cue becomes temporally distant and harder to retrieve directly.
Later conversation	User: "I just got an unexpected bonus at work. What should I do with it?"	The later question does not mention the cabin, the lake, or the earlier conversation.
Weak memory response	Assistant: "You could invest it, pay down debt, or spend part of it on travel."	This response is reasonable in general, but it ignores the earlier long term goal.
Strong memory response	Assistant: "Since you've said that buying a cabin near the lake where you grew up is an important long term goal, you might consider putting at least part of the bonus toward that fund rather than treating it only as disposable income."	This response shows that the earlier cue remained behaviorally available and could shape later reasoning.

Table 5.2 - Example of cognitive memory across temporally distant conversation turns

The key point is that the relevant memory is not just a stored fact. It is a prior goal that must be recognized as relevant to a later decision despite weak lexical overlap between the cue and the trigger. A system may technically "have" the information and still fail to use it. In that case, the memory is present in a technical sense but functionally unavailable in a behavioral sense.

How Each Test Sample Works in a Locomo Plus

Every sample follows the same basic structure. A long, realistic conversation contains a buried cue: something the user says or implies. Then, much later, after many unrelated messages, comes a trigger question whose correct answer depends on remembering that cue. The difficulty is that the trigger often uses completely different words. Keyword matching will not help. The benchmark tests whether the system can preserve behaviorally relevant context across semantic distance.[32]

The below table show the four Categories of the samples in the Locomo Plus Cognitve benchmark

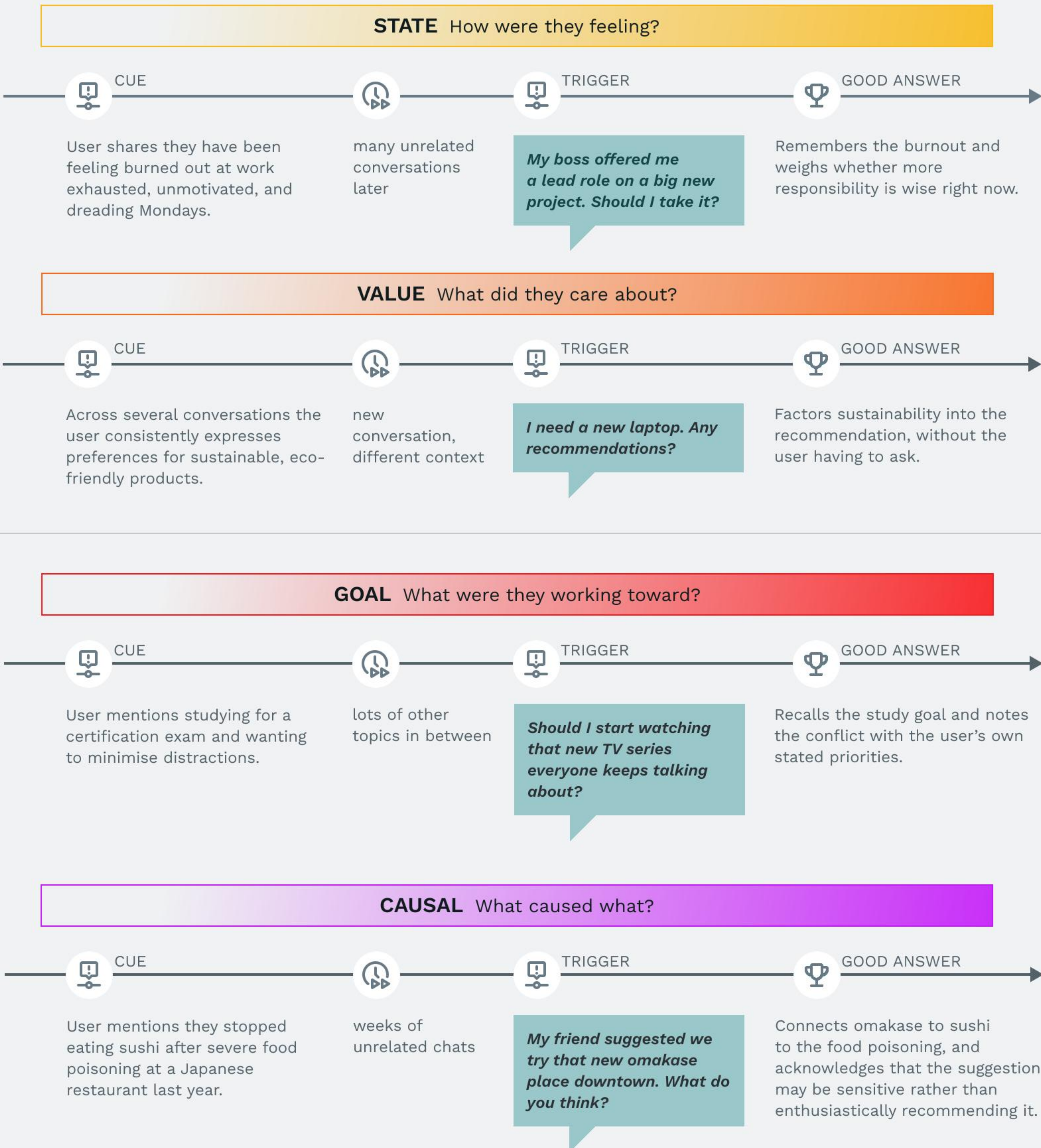


Figure 5.3 - The Four categories of the Cognitive Benchmark

Why This Distinction Matters

For many practical applications, memory is not valuable merely because it preserves facts. It is valuable because it supports continuity of judgment.

An assistant that remembers a favorite programming language is useful. An assistant that understands how earlier goals, constraints, and preferences should shape a later decision is substantially more capable.

This is especially important for systems designed to support long term interaction. Over weeks or months of engagement, the most important context is often not a discrete fact but an inferred pattern, what the user cares about, what they are trying to achieve, what tradeoffs they consistently make, and what kinds of suggestions are likely to be relevant.

Benchmarks that ignore this layer of understanding risk overstating the strength of memory architectures that are optimized for retrieval but weak at preserving behavioral meaning.

Why We Chose LoCoMo-Plus

LoCoMo-Plus is well suited to this problem because it evaluates both factual and cognitive memory using simulated multi-session conversations.[32]

Each evaluation sample consists of a dialogue trajectory between a user and an assistant spanning many sessions. These conversations are long enough to create realistic memory pressure, user goals evolve, facts accumulate, and important cues are often buried within much larger bodies of unrelated interaction.

After ingesting the full trajectory, the system must answer questions that test either explicit recall or implicit reasoning across time.

This structure makes it more representative of long running human AI interaction than benchmarks built around isolated question answer pairs. It allows memory systems to be tested not just for storage capacity, but for their ability to maintain relevance over temporal distance.

It is especially useful for comparing memory architectures that may behave differently when the signal is temporally distant, conceptually relevant, and weakly expressed.

That makes it an appropriate benchmark for distinguishing between architectures that optimize for storage and architectures that preserve behaviorally meaningful memory.

Role in Our Experiment

In the experiment described in the next chapter, the evaluated systems are exposed to the same long conversation histories and then asked questions that require either explicit recall or cognitively grounded reasoning.

This makes it possible to compare not just whether the systems store information, but what kind of understanding they preserve under realistic conversational conditions. If one system performs well on factual recall but poorly on cognitive memory, while another shows the opposite pattern, the difference is not incidental. It reflects the structure of the underlying memory architecture.

What Is Being Measured

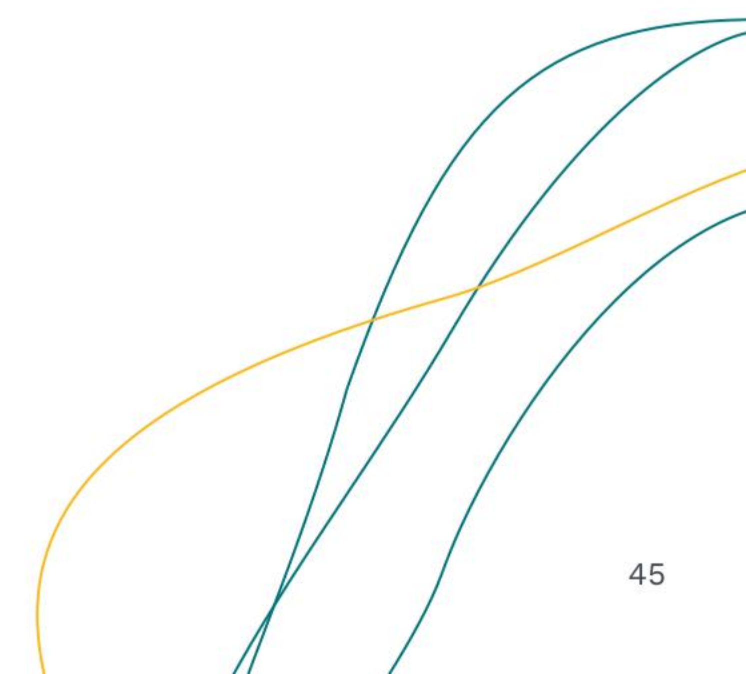
The purpose of the evaluation is therefore not simply to ask whether an agent can remember. It is to ask what kind of memory the system preserves under realistic conversational pressure.

Can it recover isolated facts from storage? Can it retain long term goals and inferred preferences across long temporal gaps? Can it apply earlier context when the later question does not restate that context directly? Can it preserve behavioral relevance when the signal is sparse and the surrounding interaction is noisy?

Those questions matter more than raw storage volume. An agent with access to a large archive is not necessarily better than one with a more selective and more structured memory system. What matters is whether the architecture retains what is genuinely useful.

This is why benchmark choice is not a secondary detail. It determines what kind of memory quality becomes visible. A benchmark focused only on explicit recall may make one architecture look strong, while a benchmark focused on latent contextual reasoning may reveal very different weaknesses. Evaluating agent memory therefore requires not only running experiments, but choosing tasks that match the kind of memory the architecture claims to preserve.

The next chapter turns from evaluation criteria to experimental design and compares three contrasting memory approaches under this framework.



Chapter 6

The Experiment

With the distinction between factual and cognitive memory established, we designed a three way experiment comparing architectures that represent fundamentally different approaches to long term agent memory.

The purpose of the experiment is not merely to measure whether a system can retrieve previously stored facts. It is to examine whether different memory designs lead to different behavior when an agent must reason over implicit user context across time.

This framing follows directly from the motivation of **LoCoMo-Plus**, introduced in February 2026.[30] That benchmark argues that existing conversational memory evaluations are too focused on explicit factual recall and therefore miss a harder and more realistic setting: cases where later responses depend on implicit constraints such as goals, values, state, or causal context that were expressed earlier but are not restated when the trigger arrives. Its **central empirical claim is that even strong current models and memory systems show a large performance drop when evaluation moves from factual memory to cognitive memory.**

That result motivates the present study. If current memory systems already show a sharp decline under cue–trigger semantic disconnect, the next question is whether different memory architectures behave differently under the same conditions. The original paper tested only extraction based and retrieval centered systems such as A-Mem, Mem0, and SeCom; self-managing and graph-based temporal architectures were not included.[30] **Our experiment extends that evaluation by adding Letta and Zep in a matched LoCoMo-Plus style setup,** while retaining Mem0 as a baseline for both architectural comparison and benchmark validation.

Systems Evaluated

The experiment compares three systems that embody contrasting memory philosophies: Mem0 (extraction based), Letta (self managing), and Zep (graph based).

All three systems were run on the same benchmark items within each run, using the same backbone generation model

family, the same final judge model,the same temperature setting, the same maximum token budget, and the same benchmark specific scoring rubric.

This makes the comparison considerably stronger than a paper to paper baseline comparison. At the same time, each system was allowed to operate according to its own native memory design.

The goal was not to force all three architectures into the same internal mechanism, but to hold the task, evaluation, and model family as constant as possible so that observed differences would more plausibly reflect memory design.

In addition, system-specific prompting necessarily differs because Letta, Mem0, and Zep expose memory to the model in different ways. We would now take a look at how each system was setup .

Mem0 - Extraction-Based Memory

Mem0 is built around an extraction-and-retrieval memory pipeline designed to preserve explicit information in a structured and searchable form.

In the benchmark setup used here, conversation batches are serialized into text blocks and stored in a local Chroma vector database under a user specific memory key. At answer time, Mem0 performs semantic search over its memory store and injects the retrieved items into the prompt before generating a response. Search is configured with a top-k retrieval limit, while deduplication and merge behavior follow Mem0's internal defaults.

The architectural logic is straightforward: store useful information explicitly, retrieve it later when it appears relevant, and use the retrieved items to support generation.

Letta - Self Managing Memory

Letta reflects a different design philosophy.

Rather than relying primarily on an external retrieval step, the agent manages memory through its own persistent state. A fresh Letta agent is created for each benchmark sample. Two memory blocks are initialized at the start of each run: a human block with a minimal user identity string and a persona block that instructs the agent to behave as a helpful conversational assistant with long-term memory.[19]

As interaction history is ingested, Letta updates what remains active and behaviorally important in its memory state rather than relying only on an external fact store. In practice, this means the system is designed to preserve not only explicit details, but also patterns, goals, and recurring contextual themes that may matter later even when they are not restated directly.[3]

Zep - Graph Based Temporal Memory

Zep reflects a third design philosophy, centered on temporal knowledge graphs rather than either flat retrieval or agent-managed internal state. In this setup, conversation history is ingested into **Graphiti**, the graph engine at the core of Zep, which incrementally builds entity nodes, relationship edges, and higher-level summaries that evolve over time.[10,31]

At query time, Zep does not rely on a single retrieval mechanism. Instead, it combines graph traversal, semantic search, and keyword search to assemble relevant context. The architectural logic is different from both Mem0 and Letta: rather than retrieving isolated stored facts or depending primarily on retained agent state, Zep attempts to preserve relationships, temporal change, and structured context as first class parts of memory .[10,31]

Dimension	Mem0	Letta	Zep
Memory philosophy	Extract and retrieve stored memory entries	Agent manages memory over time	Temporal knowledge graph
Backbone model	GPT-4o-mini	GPT-4o-mini	GPT-4o-mini
Embedding model	text-embedding-3-large	text-embedding-3-large	text-embedding-3-large
Storage	External Chroma vector store (configured)	Native persistent agent memory state	Graphiti knowledge graph
Update mechanism	Batched memory insertion	Native persistent agent memory state	Incremental graph construction
Query mechanism	Semantic search plus prompt injection	Use of retained agent memory state	Graph traversal plus hybrid search
Expected strength	Recovering explicit stored information	Preserving higher-level context and latent relevance	Temporal and relational reasoning

Table 6.1 - Three way architectural comparison

Architecture Comparison

The contrast among the three systems can be understood as a tradeoff among precision,abstraction, and temporal structure.

Mem0 stores memory externally and relies on search to surface it when needed. Letta keeps memory inside an active agent loop and allows the memory state to evolve over time. Zep stores memory as a temporal graph of entities, relations, and evolving context. These are not small implementation differences. They reflect different assumptions about what long term memory should preserve and how that preserved information should later influence behavior.

Experimental Dataset

The evaluation is primarily centered on the LoCoMo-Plus cognitive track.[30]

The cognitive evaluation uses 401 non overlap samples from LoCoMo-Plus, balanced across relation types: 101 causal, 100 goal, 100 state, and 100 value.[30]

A benchmark sample in this setting is not a short isolated exchange. In the underlying LoCoMo data, conversations average 588.2 turns, 16,618.1 tokens, and 27.2 sessions, with some extending to as many as 32 sessions.

These cognitive items are embedded in long conversational histories composed of multiple distinct sessions, which means that each evaluated sample may contain a substantial amount of interaction before the final trigger is reached. This matters for interpreting the scale of the experiment.

Even what appears to be a modest subset of the benchmark can expand into a much larger effective memory workload at the session level. For example, if one sample contains around 20 sessions, then 100 samples already correspond to roughly 2,000 session-level conversational segments for a single system, and roughly 6,000 across the three systems evaluated here. Using the LoCoMo average of 27.2 sessions, the full 401-sample cognitive track corresponds to roughly 10,907 session-level segments per system.

Cognitive category	What it tests
Value	Whether the system preserves what the user cares about or what principles shape later decisions
Goal	Whether long-term intentions remain available when later choices should be informed by them
State	Whether the system tracks evolving conditions, emotions, or circumstances that should change how later responses are framed
Causal	Whether earlier events continue to shape later interpretation and advice

Table 6.2 - Cognitive categories in LoCoMo-Plus

The dataset should therefore be understood not as a collection of short retrieval prompts, but as a large multi-session memory stress test in which useful context may be distributed across long interaction histories and only become relevant much later.

Within each run, all three systems processed the same sample IDs.

Experimental Procedure

The experiment was conducted in three stages.

Stage	What happens
Dialogue ingestion	All three systems process the same benchmark history up to the final query point
Question phase	Letta answers from agent memory state; Mem0 answers from retrieved memory plus prompt; Zep answers from graph-backed retrieved context plus prompt
Evaluation	Blind judging under the same benchmark rubric

Table 6.3 - The three stages of the Experiment

Dialogue Ingestion

All three systems ingest the same conversation histories for each sample. Dialogue is formatted the same way for all systems: newline-joined "speaker: text" strings. Full dialogue turns are included rather than user only messages.

In all systems, ingestion is batched according to a shared ingestion batch size. This means the benchmark is neither strict one turn at a time replay nor full history at once unless the remaining history fits in a single batch.

For Letta, each batch is sent into the agent as a message input, allowing the agent to update its own memory state. For Mem0, each batch is added into the external memory store through a memory insertion call. For Zep, each batch is ingested into the Graphiti memory engine, which updates its graph representation incrementally as new episodes arrive.[10,31]

For cognitive samples, the **trigger is withheld until the final query**. All three systems ingest the history only up to the trigger point. This is a crucial design choice because it prevents the benchmark from collapsing into simple prompt proximity and forces the architecture to preserve earlier cue information until it becomes relevant later. That design is directly aligned with the logic of LoCoMo-Plus, whose benchmark construction intentionally creates cue–trigger pairs under semantic disconnect rather than explicit query overlap.

Question Phase

After ingestion, each system is asked the benchmark question.

For Letta, the final question or trigger is sent directly to the agent. For Mem0, the system first performs semantic search over stored memory and then generates a response using the retrieved memories and the final user query. For Zep, the system queries the graph-backed memory layer, which assembles relevant context through graph traversal, semantic search, and keyword search before final generation.

No system is simply given the full transcript at answer time. Each one must rely on what its own memory mechanism has preserved and can surface during generation.

Generation settings were matched:

- temperature: **0.0**
- max tokens: **512**

Evaluation

The final consolidated judging uses a single blinded judge model: **gemini-2.5-flash**.

The judge does not receive system identity. It only sees the benchmark inputs and the prediction text. Cognitive items are scored using a binary **correct** / **wrong** rubric. Statistical significance was assessed using Cochran's Q test across all three systems, followed by pairwise McNemar tests with Holm correction for multiple comparisons.

This evaluation choice is consistent with the broader direction proposed by LoCoMo-Plus, which argues that cognitive memory should be judged in terms of constraint consistency rather than surface overlap alone. In that benchmark, cognitive correctness is defined not by exact-match reproduction of a reference string, but by whether the response remains consistent with the earlier cue-induced constraint.

The experiment is fully reproducible. All implementation details needed to rerun it are provided, including the exact prompts, system configurations, judge settings, and evaluation procedures used in the reported results. The QR code links directly to the project repository and the detailed result reports, allowing the full experimental setup and outputs to be inspected and reproduced precisely.

What the Experiment Is Designed to Reveal

Mem0 is not included only as a comparison point. It also serves as an internal benchmark sanity check.

The February 2026 LoCoMo Plus paper reports that modern memory systems, including Mem0, perform reasonably on LoCoMo factual memory but show a sharp decline on LoCoMo Plus cognitive memory. SeCom and A-Mem show similarly large factual-to-cognitive gaps. [30] Our setup is not identical to that original experiment, and the absolute numbers should therefore not be treated as directly interchangeable. But if Mem0 in the matched local setup shows the same broad pattern stronger factual behavior followed by a much weaker cognitive result that would support the claim that the experiment is capturing the same underlying difficulty rather than measuring an unrelated artifact.

The experiment is therefore not meant to ask which system "has memory." All three systems do. The relevant question is whether different memory architectures preserve different kinds of useful information once conversations become too long and too diffuse for naive context replay.

A retrieval centered system should be strong when success depends on recovering explicit stored information. A self managing memory system should have an advantage when success depends on preserving higher level context, latent goals, or conceptually distant cue trigger relationships. A graph based temporal system should have an advantage when success depends on preserving relationships, temporal evolution, and structurally linked context across time.

The next chapter asks whether that difference actually appears in performance, and whether the matched Mem0–Letta–Zep comparison reproduces the broader cognitive memory difficulty identified by LoCoMo Plus while also showing where different architectures may go beyond it.

Results and Analysis

In our evaluation, Letta outperforms Mem0 by 34.4% on the LoCoMo-Plus cognitive benchmark, with Zep close behind.

This chapter turns from setup to outcome. It presents, to our knowledge, the first benchmark comparison of Letta and Zep on the LoCoMo Plus 2026 cognitive benchmark, and asks whether the architectural differences described in the previous chapter translate into meaningful differences in performance.

The answer is Yes.

Across the evaluation, each architecture reveals a different profile. **Letta** achieves the highest overall score on cognitive memory. Zep does not lead overall, but it shows a distinctive advantage on causal reasoning. Mem0 remains weakest on the cognitive track, even though it performs best on factual recall. Taken together, the results suggest that memory architecture does not merely affect how much an agent remembers. It affects what kind of context the agent is able to preserve and use when it matters.

Overall Cognitive Performance

The central question of the study concerns cognitive memory: whether a system can preserve and later apply earlier context when the later trigger does not restate that context directly. The final cognitive evaluation covers 401 non overlap LoCoMoPlus dialogues, with all three systems judged on identical samples.

This is the clearest empirical finding in the chapter. Letta leads at 24.2%, Zep follows at 22.2%, and Mem0 trails at 18.0%.

The overall difference is **statistically significant**. Cochran's Q test across all three systems yields $Q = 7.14$, $p = 0.028$, which rejects the null hypothesis that all three architectures perform equally. In other words, the differences are unlikely to be random noise.

System	Architecture	Correct	Wrong	Accuracy	Wilson 95% CI
Letta	Self-managing	97	304	**24.2%**	**[20.3%, 28.6%]**
Zep	Graph-based	89	312	**22.2%**	**[18.4%, 26.5%]**
Mem0	Extraction-based	72	329	**18.0%**	**[14.5%, 22.0%]**

Table 7.1 -Cognitive memory results (401 non-overlap LoCoMo-Plus dialogues) with Wilson 95% confidence intervals.

Pairwise tests sharpen the picture. The Letta–Mem0 gap is statistically significant after Holm correction ($p = 0.040$), confirming that the self-managing architecture outperforms the extraction-based one on aggregate cognitive memory. By contrast, the gaps between Letta and Zep ($p = 0.466$) and between Zep and Mem0 ($p = 0.171$) do not reach significance at the 0.05 level. Zep therefore occupies an intermediate position: architecturally distinct and descriptively competitive, but not separable from either competitor on aggregate score alone.

This matters because it prevents the chapter from collapsing into a simplistic ranking. The most robust top line conclusion is not merely that Letta finishes first. Self managing memory shows a significant aggregate advantage over extraction based memory on the LoCoMo Plus cognitive benchmark, while graph-based temporal memory reveals a different pattern of strengths that becomes clearer in the breakdowns.

Performance by Relation Type

The aggregate ranking is useful, but it is not the most interesting part of the story.

LoCoMo-Plus organizes cognitive memory around four relation types: causal, goal, state, and value. Once the results are broken down by relation type, the three architectures reveal sharply different strengths.

Relation Type	n	**Mem0**	**Letta**	**Zep**
Causal	101	11.9%	12.9%	20.8%
Goal	100	23.0%	30.0%	21.0%
State	100	15.0%	28.0%	22.0%
Value	100	22.0%	26.0%	25.0%

Table 7.2 — Cognitive accuracy by relation type.

Letta's strength: goals and state

Letta leads on goal reasoning at 30.0%, the highest single category score in the experiment. It also leads on state reasoning at 28.0%.

These are exactly the kinds of cues that a self-managing memory architecture would be expected to preserve well. Goals are forward looking and identity linked; they matter later because they continue to shape what the user is working toward. State cues are often even harder. They may appear only as passing remarks about stress, fatigue, mood, or changing circumstances.

A persistent agent memory that continuously updates its internal picture of the user is well positioned to carry these conditions forward without requiring a near exact retrieval match. Mem0's weakest category is state at 15.0%, which underscores the difficulty of recovering emotional or situational context through retrieval alone.

Zep's strength: causal reasoning

Zep leads decisively on causal reasoning, scoring 20.8%, well ahead of Mem0's 11.9% and Letta's 12.9%.

That pattern is intuitively plausible. Causal reasoning depends on preserving structured relationships among events across time.

A temporal graph is naturally suited to that kind of information. When a later question depends on how one earlier event should shape the interpretation of another, graph structure can keep those links available even when the trigger uses different language.

This is the clearest domain in which Zep's architectural logic shows through directly in performance.

Values: relative convergence

On value reasoning, all three systems are closer together: Letta at 26.0%, Zep at 25.0%, and Mem0 at 22.0%.

That convergence is interesting in its own right. Values may be easier than causal, goal, or state cues for multiple architectures to retain because they are often expressed with some clarity and repetition. Even so, the absolute scores remain modest. None of the systems is close to solving the problem.

Performance by Time Gap

A second useful lens is the temporal distance between cue and trigger.

Time gap	n	**Mem0**	**Letta**	**Zep**
Short (1-2 weeks)	91	15.4%	27.5%	27.5%
Long (3+ months)	300	19.0%	22.3%	20.7%
Unknown	10	10.0%	50.0%	20.0%

Table 7.3 - Cognitive accuracy by time gap between cue and trigger.

On shorter time gaps, Letta and Zep are tied at 27.5%, both far ahead of Mem0's 15.4%. For relatively recent cues, both the self-managing and graph-based systems appear better able to preserve behaviorally useful context.

On longer time gaps - three months or more, which account for most of the benchmark. The systems converge toward a narrower band: Letta at 22.3%, Zep at 20.7%, and Mem0 at 19.0%. The advantage of the non extraction architectures shrinks, but it does not disappear.

This suggests that over very long horizons, all current approaches face the same underlying challenge of memory decay. The difference is not that Letta or Zep avoid the difficulty. It is that they seem to degrade more gracefully. The unknown category is too small to interpret confidently.

Factual Memory Performance

Before interpreting the cognitive results more broadly, the factual benchmark provides an important calibration check. If the cognitive advantage simply reflected one system being globally stronger, the factual results should show it. They do not.

System	Architecture	Score	Sample
Mem0	extraction-based	63.0%	100
Letta	self-managing	59.0%	100

Table 7.4 — Factual memory results (100 sampled LoCoMo QA items)

Mem0 leads factual recall by four percentage points. That ordering makes architectural sense. Extraction based systems are optimized to preserve and recover explicit stored information, so they should perform well when the task is to retrieve a fact that was stated earlier in a relatively direct form.

The value of this result is mostly diagnostic. It shows that the later cognitive findings are not simply the product of Letta being globally stronger than Mem0. In fact, on explicit recall, Mem0 performs slightly better. That makes the reversal on cognitive memory more meaningful.

The Cognitive Decline Pattern

A crucial question is whether the experiment reproduces the broader difficulty identified by the original LoCoMo-Plus paper. It does.

The core claim of LoCoMo-Plus is that many current systems perform reasonably on factual memory yet collapse once evaluation shifts to cognitive memory under cue–trigger semantic disconnect. Our results follow that same pattern.

Letta declines too, but less sharply. Its drop is 34.8 points, roughly ten points smaller than Mem0's. That does not eliminate the cognitive memory difficulty. It does, however, suggest greater resilience under the same general shift from explicit recall to implicit contextual reasoning.

The broader lesson is that the cognitive decline appears to be universal, but not equally severe across architectures.

System	Factual	Cognitive	Gap(pp)	Source
A-Mem	59.64	17.20	−42.44	LoCoMo-Plus paper
SeCom	57.53	14.90	−42.63	LoCoMo-Plus paper
Mem0 (paper)	57.24	15.80	−41.44	LoCoMo-Plus paper
Mem0 (our setup)	63.0%	18.0%	−45.0	This study
Letta (our setup)	59.0%	24.2%	−34.8	This study

Table 7.5 - Cognitive memory Gap

Agreement Across Systems

Another revealing perspective is how often the three systems agree.

Of the 401 dialogues, 237 (59.1%) were answered incorrectly by all three systems. Only 27 (6.7%) were answered correctly by all three.

This makes clear that none of the architectures has solved cognitive memory. Most tasks still defeat all three systems.

The disagreement cases are equally informative. Letta alone was correct in 40 cases. Zep alone was correct in 32. Mem0 alone was correct in 25.

The paper reports that modern memory systems, including Mem0, perform reasonably on factual memory but show those unique correct cases matter because they are not random. They line up with the broader architectural picture. Letta's unique successes tend to appear where goal or state continuity matters. Zep's stand out where causal chains matter. Mem0 still has unique wins of its own, especially in cases closer to explicit recoverable information.

Fleiss' kappa across the three systems is 0.324, indicating fair agreement. The systems broadly agree on what is easy and what is hard, but they diverge enough in their successes to suggest that they are capturing genuinely different parts of the memory problem.

Precision, Abstraction and Structure

Taken together, the results reinforce a broader architectural tradeoff.

Extraction based systems optimize for precision. They preserve explicit information in retrievable form, which makes them effective when later tasks depend on recovering a stored fact.

Self managing systems optimize for abstraction. They preserve a more compressed but behaviorally meaningful representation of what continues to matter over time, which appears to help on goals and state.

Graph based systems optimize for structure. They preserve entities, relationships, and temporal evolution explicitly, which appears to help when success depends on causal linkage across time.

The results do not imply that one architecture is universally best. They imply something more useful: different architectures are matched to different cognitive demands.

Interpreting the Results

Several conclusions follow clearly from the experiment.

First, the overall difference between architectures is real. The three systems do not perform equally on cognitive memory.

Second, the Letta Mem0 difference is the strongest aggregate finding. If the practical choice is between self-managing memory and extraction based retrieval for cognitively demanding long-term interaction, the evidence here favors the self managing design.

Third, Zep's value is not captured fully by aggregate score alone. Its standout performance on causal reasoning suggests that graph based temporal memory may be especially useful when the challenge is not simply remembering something from the past, but preserving the structure of how events relate.

Fourth, all three systems remain modest in absolute terms. The highest category score in the experiment is Letta's 30.0% on goal reasoning. Most tasks still defeat all architectures. Cognitive memory remains genuinely difficult.

That is the central lesson of the chapter. The point is not that one system has solved the problem. It is that the pattern of differences reveals which design properties matter for which kinds of context.

The next chapter turns from benchmark performance to practical implications: where these differences matter most, and why cognitive memory is likely to become increasingly important in real world agent systems.

Limitations

These results should be interpreted with several limitations in mind.

Although the experiment is matched at the level of benchmark items, model family, and scoring protocol, the three systems necessarily expose memory differently at the implementation level.

The study also relies on remote APIs for generation and judging, uses a single blinded judge in the final consolidated evaluation, and depends mainly on one benchmark family for the cognitive analysis.

For these reasons, the results are best understood as strong evidence of meaningful architectural differences, not as a final universal ranking of memory systems.

Chapter 8

Where Different Architectures Matter

Chapter 7 showed that the real question is not whether an agent can store information, but what kind of context it can carry forward in usable form.

Different architectures do not simply remember more or less. They preserve different kinds of relevance. That distinction matters because real systems are not judged by benchmark categories. They are judged by whether they remain helpful over time.

The practical implication is straightforward choosing a memory architecture is really a decision about what sort of continuity the agent is expected to sustain. Some applications depend mainly on explicit factual recovery. Others depend on preserving a user's goals, emotional state, or long running constraints. Still others depend on linking events across time and understanding how one earlier decision shapes a later outcome. The architecture should be chosen accordingly.

Where Letta Matters Most

Letta is strongest in settings where the value of the agent depends on maintaining an evolving picture of the user across repeated interaction. Its best results in the benchmark appeared on goal and state reasoning, where it reached 30% and 28% respectively, the two highest dimension-level scores in the study. It also performed best on short time gaps, scoring 27.5% when the cue and trigger were separated by only one to two weeks.

This makes **Letta a natural fit for personal assistants, planning agents, coaching systems, tutoring tools, and relationship oriented support roles.**

A personal assistant may need to remember that a user is trying to reduce travel, save for a home purchase, or rebalance work and family life, and later use that context when advising on opportunities, spending, or scheduling.

A coaching assistant may need to preserve recurring emotional patterns such as hesitation, stress, or burnout, so that later guidance reflects the user's actual condition rather than only the wording of the latest question.

A tutoring system may need to maintain a persistent sense of the learner's confidence, confusion patterns, and longer-term learning goals rather than simply recalling the last exchange.

Across these domains, usefulness depends on continuity of judgment. The system becomes more valuable when it can carry forward what remains behaviorally important, even when the later prompt does not explicitly ask it to do so. That is the kind of memory problem for which a self managing architecture appears especially well suited.

Where Zep Matters Most

Zep is strongest in settings where memory must preserve causal structure, temporal change, and linked relationships. Its clearest empirical advantage appeared on causal reasoning, where it scored 20.8%, well ahead of Letta's 12.9% and Mem0's 11.9%. It was also the most even performer across the four cognitive dimensions, ranging only from 20.8% to 25.0%.

This makes **Zep a strong fit for technical support systems, incident response workflows, audit and compliance tools, investigative agents, and diagnostic settings**. A support agent may need to connect a current failure to an earlier configuration change or service dependency.

A compliance assistant may need to track when a policy changed, which teams were affected, and how an earlier decision relates to a later requirement. A diagnostic or investigative system may need to connect symptoms, interventions, and outcomes across time rather than treating each observation as an isolated fact.

In these domains, the memory challenge is not simply "what was said before?" but "what changed, what caused what, and how do these events connect?" That is precisely where graph-based temporal memory has the clearest structural advantage. Its value lies not only in preserving information, but in preserving the relationships among pieces of information.

Where Mem0 Still Makes the Most Sense

Mem0 remains attractive in settings where the dominant requirement is explicit factual recovery, preference tracking, lightweight personalization, and operational simplicity. It does not perform as strongly on the cognitive benchmark, but that does not make it a poor choice overall.

It means its strengths lie in a different part of the design space. In particular, it remains competitive on value/preference tasks, where it scored 22.0%, not far behind Letta's 26.0% and Zep's 25.0%.

This makes **Mem0 a good fit for recommendation systems, marketing personalization, lightweight enterprise assistants, and early-stage products** where the main memory value comes from recovering something the user stated clearly before.

A recommendation engine may need to remember preferred brands, budget ranges, or product categories. A personalization layer may need to preserve explicit preferences, stable profile attributes, or account history.

A prototype assistant may simply need to demonstrate that memory improves user experience before a more complex architecture is justified.

In these settings, extraction based memory remains appealing because it is relatively simple, efficient, and well aligned with tasks where memory is mostly a matter of recovering explicit information. The mistake is not using retrieval based memory. The mistake is expecting retrieval based memory alone to behave like cognitive memory when the relevant context becomes conceptually distant from the later query. That limitation is especially visible in frequent-interaction settings: Mem0 scored only 15.4% on short gap items, compared with 27.5% for Letta and 27.5% for Zep.

aci initiative for applied artificial intelligence

A Practical Rule of Thumb

A useful way to decide between architectures is to ask one question:

Is the hard part of the memory problem finding the stored item, preserving the user's evolving context, or linking events across time?

Architecture	Best fit when	What it preserves well	Main tradeoff
Letta (self-managing)	The product depends on continuity of judgment over weeks or months	Goals, evolving state, recurring patterns, what still matters behaviorally	Less transparent than explicit stores; may lose some factual granularity
Zep (graph-based temporal)	The job is linking events across time and preserving "what caused what"	Causal chains, relationships, temporal evolution, connected context	Less naturally centered on user-state continuity unless modeled explicitly
Mem0 (extraction + retrieval)	You need reliable recall of explicit facts and lightweight personalization	Preferences, profile facts, clear statements, retrievable explicit memory	Relevant context may remain stored but fail to surface under semantic disconnect

Table 8.1 - Where each architecture matters most in practice

Why Hybrid Architectures Are the Likely End State

The most important implication of the three-way comparison may be that no single architecture performs best across every kind of cognitive demand.

This is the strongest argument for hybrid designs. A system that combines persistent agent state for goals and emotional context with a temporal graph layer for causal and relational queries could, in principle, cover a wider range of real world memory demands than any single paradigm alone. The absolute performance numbers make clear that cognitive memory remains difficult. But the unique success cases show that there is real headroom in combining strengths rather than committing to one approach exclusively.

The practical takeaway is therefore not that hybrid systems are automatically superior. It is that the results make a compelling case for architectural composition rather than architectural absolutism.

Governance and Trust

As memory becomes more persistent, architecture also becomes a governance choice.

A memory system determines not only what an agent can remember, but also what can be inspected, corrected, deleted, or audited. These are not secondary concerns. They affect trust directly.

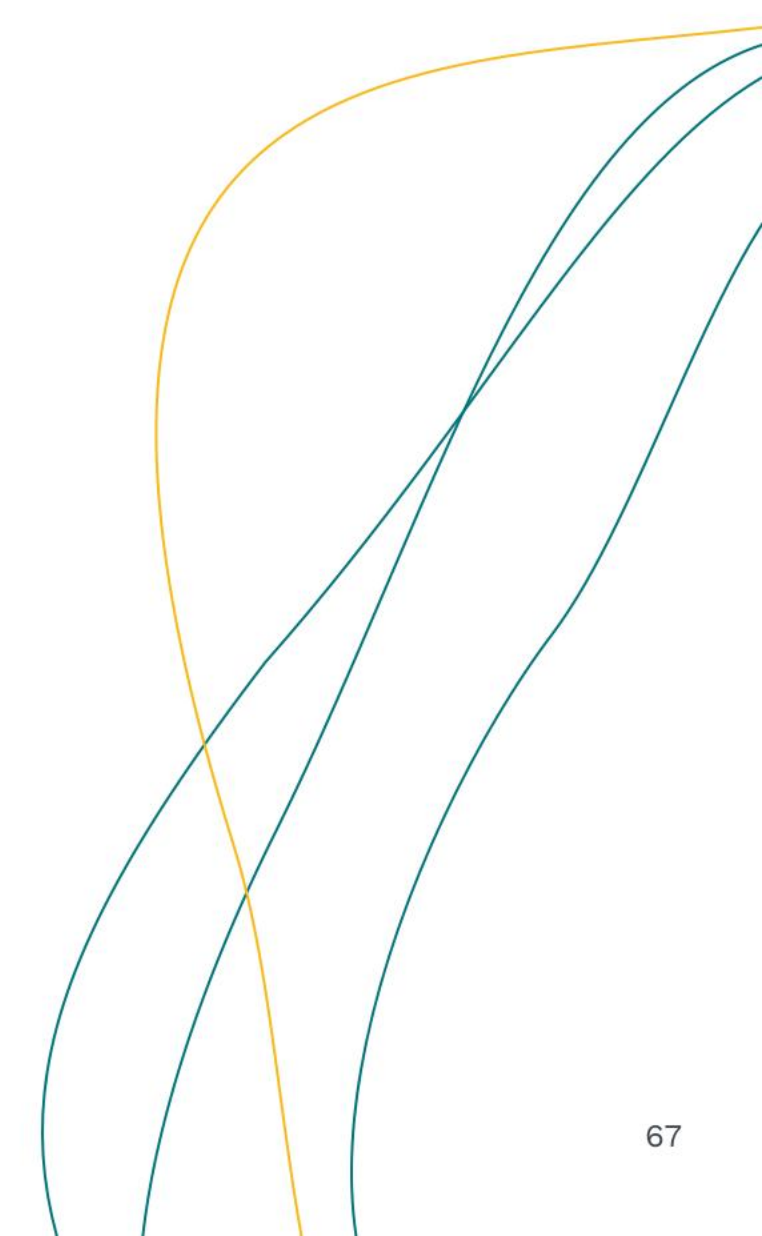
Graph based systems with provenance tracking offer advantages for traceability. File backed systems offer strong inspect ability because their memory remains human readable and directly editable. Extraction based systems often make it easier to add, retrieve, and delete discrete stored items. Self managing systems may perform better on certain cognitively rich tasks, but their internal state can be harder to inspect and explain.

These differences matter in practice. Can a user see what the system remembers?

Can outdated or sensitive information be removed selectively?

The architecture chosen therefore affects more than benchmark performance. It shapes auditability, transparency, temporal accuracy, and the trustworthiness of the system in real deployment.

The next and final chapter draws these findings together. It moves from where different memory architectures matter in practice to what those differences mean for the future of agent design, arguing that memory is not merely a storage feature but a core architectural theory of continuity, relevance, and intelligent behavior over time.



Conclusion

This whitepaper has traced a clear journey in AI system design: from stateless models, to context reconstruction, to retrieval-based memory, and finally to structured memory architectures designed to support long-term agency.

Early AI systems could respond intelligently in the moment, but they could not truly remember. What appeared to be continuity was often only temporary context replay. As AI moved from one off assistance toward more autonomous, long running agents, that limitation became harder to ignore. Memory was no longer just a helpful feature. It became a systems problem.

The central lesson is that memory is not simply about storing more information. It is about preserving what matters in forms that remain useful over time. Facts, experiences, goals, user state, and causal relationships do not place the same demands on an architecture. As a result, different memory systems preserve different kinds of continuity.

That is exactly what the experiment revealed. Mem0, Letta, and Zep are not separated only by implementation details, but by the kinds of context they retain and apply most effectively. Retrieval based memory remains strong for explicit factual recovery. Self managing memory appears better suited to preserving goals and behavioral continuity. Graph based temporal memory shows particular value when events, relationships, and causal structure must remain connected across time.

The broader conclusion is not that one architecture has solved memory. None has. Cognitive memory remains difficult, and the results make clear that storing the past is not the same as using it well. But the differences between architectures are meaningful. They show that memory design helps determine what an agent can understand, what it can carry forward, and what it can still recognize as relevant when the moment changes.

In that sense, the journey described in this paper is larger than a technical shift. It is the movement from storage to continuity, from recall to relevance, and from response in the moment to understanding across time.

The future of AI agents will depend not only on what they can generate, but on what they can carry forward and use with judgment.

References

[1] A. Vaswani et al., "Attention is all you need," in Proc. NeurIPS, 2017.
[2] C. Packer et al., "MemGPT: Towards LLMs as operating systems," arXiv:2310.08560, Oct. 2023.
[3] N. F. Liu et al., "Lost in the middle: How language models use long contexts," Trans. Assoc. Comput. Linguist., vol. 12, 2024. arXiv:2307.03172.
[4] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," arXiv:2312.00752, Dec. 2023.
[5] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. NeurIPS, 2020. arXiv:2005.11401.
[6] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. ICLR, 2023. arXiv:2210.03629.
[7] J. S. Park et al., "Generative agents: Interactive simulacra of human behavior," in Proc. UIST, 2023. arXiv:2304.03442.
[8] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. ICLR, 2024. arXiv:2310.06770.
[9] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," arXiv:2405.15793, Apr. 2024.
[10] P. Rasmussen et al., "Zep: A temporal knowledge graph architecture for agent memory," arXiv:2501.13956, Jan. 2025.
[11] S. Zhang et al., "MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory," arXiv:2601.03192, Jan. 2026.
[12] "Graph-native cognitive memory for AI agents: Formal belief revision semantics for versioned memory architectures," arXiv:2603.17244, Mar. 2026.
[13] Y. Hu et al., "Memory in the age of AI agents: A survey," arXiv:2512.13564, Dec. 2025; revised Jan. 2026.
[14] T. B. Richards, "AutoGPT," GitHub repository, Significant Gravitas, 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT
[15] T. R. Sumers et al., "Cognitive architectures for language agents (CoALA)," Trans. Mach. Learn. Res., 2024. arXiv:2309.02427.
[16] Mem0 documentation, "Vector-database overview and supported backends." [Online]. Available: https://docs.mem0.ai
[17] CrewAI documentation, "Unified memory system and composite recall." [Online]. Available: https://docs.crewai.com
[18] Cognee documentation, "Graph-based memory construction and retrieval." [Online]. Available: https://docs.cognee.ai
[19] Letta documentation, "Core, recall, and archival memory." [Online]. Available: https://docs.letta.com
[20] Claude Code documentation, "Persistent Markdown memory via CLAUDE.md." [Online]. Available: https://docs.anthropic.com
[21] OpenClaw documentation, "File-backed memory and workspace search."
[22] LangGraph documentation, "Persistence, checkpoints, and long-term memory patterns." [Online]. Available: https://langchain-ai.github.io/langgraph/
[23] AutoGen documentation, "Pluggable memory protocol and retrieval options." [Online]. Available: https://microsoft.github.io/autogen/
[24] Strands Agents and Amazon Bedrock AgentCore Memory documentation.
[25] OpenAI Agents SDK documentation, "Sessions and session memory." [Online]. Available: https://openai.github.io/openai-agents-python/
[26] LangMem documentation, "Long-term memory extraction and background consolidation." [Online]. Available: https://github.com/langchain-ai/langmem
[27] A. Maharana et al., "Evaluating very long-term conversational memory of LLM agents," in Proc. ACL, vol. 1, pp. 13851–13870, 2024. arXiv:2402.17753.
[28] D. Wu et al., "LongMemEval: Benchmarking chat assistants on long-term interactive memory," in Proc. ICLR, 2025. arXiv:2410.10813.
[29] Y. Hu et al., "Evaluating memory in LLM agents via incremental multi-turn interactions," (MemoryAgentBench), in Proc. ICLR, 2026. arXiv:2507.05257.
[30] Y. Li et al., "Locomo-Plus: Beyond-factual cognitive memory evaluation framework for LLM agents," arXiv:2602.10715, Feb. 2026.
[31] Graphiti documentation and source code, "Temporal knowledge graph engine." [Online]. Available: https://github.com/getzep/graphiti
[32] G. Kamradt, "Needle in a haystack — Pressure testing LLMs," GitHub repository, 2023. [Online]. Available: https://github.com/gkamradt/LLMTest_NeedleInAHaystack

aci initiative for applied artificial intelligence

Authors

Harsh Gurawaliya

Junior AI Engineering LLM

h.gurawaliya@appliedai.de

Harsh Gurawaliya is a Bachelor's student in Artificial Intelligence at Deggendorf Institute of Technology (DIT). As a working student at appliedAI, he focuses on developing cutting-edge AI solutions involving large language models, AI agents, and memory architectures. He combines a strong academic foundation with hands on engineering skills, applying advanced AI methods to develop prototypes that translate theoretical ideas into practical solutions for real world challenges.

Mingyang Ma

Head of Agentic AI Sollutions Development

m.ma@appliedai.de

Mingyang Ma is Head of Agentic AI at appliedAI, where she leads the design and deployment of multi-agent systems for enterprise transformation. In the past year, her team has partnered with eight organizations to build production-grade agentic solutions that deliver measurable business value. She brings nearly a decade of expertise in NLP, conversational AI, and human–machine interaction, shaped by five years at BMW Group and a career spanning China, the United States, and Germany.

Dr Malte Nalenz

Generative AI Engineer

m.nalenz@appliedai.de

Malte Nalenz is a Generative AI Engineer at appliedAI initiative GmbH, working at the intersection of multi-agent systems and optimization problems. With 8 years of experience, he has developed algorithms across a wide range of domains — from forecasting and classification to document processing. Malte holds a PhD from LMU Munich, where his research focused on interpretable machine learning and ensemble learning methods.

initiative for applied artificial intelligence

About appliedAI

appliedAI is Europe's largest initiative for the application of trusted AI technology. The initiative was established in 2017 by Dr. Andreas Liebl as a division of UnternehmerTUM Munich and transferred to a joint venture with Innovation Park Artificial Intelligence (IPAI) Heilbronn in 2022.

At the Munich and Heilbronn offices, more than 100 employees pursue the goal of making European businesses a shaper in the AI era in order to maintain Europe's competitiveness and actively shape the future.

appliedAI holistically supports international corporations, including BMW and Siemens, as well as medium-sized companies in their AI transformation. This is accomplished through partnership-based exchange and joint knowledge building, comprehensive accelerator programs, and specific solutions and services, such as strategy consulting and Use-Case development.

For more information, please visit

www.appliedai.de/en/

Acknowledgement

We express our sincere appreciation for the excellent work carried out by the authors, reviewers, and designer of this white paper. Their great expertise, motivation, and dedication to every detail made this white paper an exceptional contribution to the AI community.

The collective expertise, exchange, and dedication to advancing the knowledge in generative AI and agentic applications were great inspirations throughout the process of creating this white paper.

ai initiative for applied artificial intelligence

Agentic Memory Realized: From Stateless Models to Cognitive Continuity

appliedAI Initiative GmbH

August-Everding-Straße 25
81671 München
Germany
www.appliedai.de