AI Memory: Comprehensive Review

December 2025

Jonathan Politzki, Founder of Jean Memory

jonathan@jeanmemory.com

Jean

Table of Contents

jonathan@jeanmemory.com

Introduction

Since the publication of General Personal Embeddings, Irreverent has been focused on the general embedding space and how new systems will shape the future of software. Over the last 10 months, this space has evolved into a new sector called **AI Memory**. While a significant amount of talent and investment has gone into the optimization of LLMs, we believe that AI Memory exists as a relatively underexplored area of progress, for the simple fact:

"Bad model with good context > good model with bad context." [Vectorize]

In building systems for consumers, developers, and enterprises, we sought out literature to catch up to the state-of-the-art. To our surprise, there have been no complete overviews written of the memory space. This exposition will aim to be the first comprehensive review.

But to just catch up to the frontier would not be enough. After familiarizing ourselves so deeply with the space, limitations of current technical architectures, market participants, and available solutions was clear. **In order to truly understand memory, we must first forget it.**

—

In **Part 1**, we trace the pre-history of AI Memory, from the ChatGPT moment through the emergence of basic RAG, structured outputs, through context engineering. We examine how memory was initially retrofitted, and why this history shapes the constraints we face today.

In **Part 2**, we establish the technical foundations. We define what memory actually is (and isn't), introduce the Experience → Storage → Recall pipeline, and present a unified theory for evaluating memory architectures. We examine vectors, graphs, hybrid systems, and the emerging paradigm of agentic memory—offering practical guidance for developers building these systems.

In **Part 3**, we map the market. We survey use cases across B2C, B2B2C, and B2B applications, profile the major providers from platforms to frameworks to vertical solutions, and analyze the real tradeoffs developers face when deciding whether to build or buy.

In **Part 4**, we synthesize everything into a future outlook. We present four theses: that memory will become increasingly agentic, that value will flow to platforms and high-value vertical solutions while squeezing the middle, that we are still early in an explosion of use cases, and that the ultimate prize—cross-application context—remains unclaimed.

This guide is written for developers building AI systems, investors evaluating the space, and founders positioning products within it. Our aim is not neutrality but clarity: we have opinions, informed by building in this space, and we share them throughout.

The ChatGPT Moment

When OpenAI released ChatGPT on November 30, 2022, it quickly captured the public's attention. It was incredible that a deceptively simple interface appeared to think as we do.

This simplicity masked limitations that users quickly discovered: ChatGPT forgot everything between sessions. Ask it to remember your preferences one day, and the next day you'd start from scratch.

How LLMs Actually Work: Databases, Not Brains

One large misconception often made by the public is that while Large Language Models (LLMs) seem very intelligent and "smart," they are really more databases than they are brains. This is why LLMs often fail when doing simple operations of arithmetic. These products are not great at solving problems and thinking through new math, they are fundamentally built to parrot things they have been trained on.

A better heuristic for LLMs is as **probabilistic databases of vector programs** rather than reasoning entities [Chollet]. During training, these models encode patterns: how technical documentation is structured, how confident people write versus shy people, how to explain quantum physics at different levels, how poetry flows, how code debugging proceeds.

The activation problem: Without specific context, LLMs default to "average". When tasked with generating an image of a male, the below image is generated. We can think of this person as effectively if you mashed together the average face of billions of images of faces it was trained on.

For example, this is a ChatGPT generated image of the average male face.

Prompt: “Now generate an image of the average wisconsinite, cheese-loving, packer fan”

It is only by introducing context on “wisconsinite” “cheese-loving” and “packer fan” that these feature dials are turned up and a program that resembles the combination of all of these factors generates this image.

The working memory analogy: We can not say that LLM’s don't have memory at all. Clearly, these systems have some level of recall for conversations that fall within your chat window. But this is only a result of how the products we use were built.

Similar to how working memory for humans is around 7 items, LLMs have a working memory or “short-term” memory that only exists because we append prior messages to every new chat. So every time you send a new message, that message, along with all the others before it, are sent as a new prompt to the AI. But there are limits to working memory and LLMs as a whole.

The Three Core Limitations of LLM Recall

1. Statelessness

Every API call to an LLM is independent. The model retains nothing from previous interactions unless explicitly provided in the current context. This creates cascading problems:

- Users must repeat background information constantly
- Applications cannot learn from interaction patterns
- Multi-session workflows require manual context management
- Personalization demands extensive prompt engineering

Developers initially worked around this by storing conversation history and re-injecting it with each request. But this naive approach quickly hits context limits and becomes prohibitively expensive as conversations grow.

This is the most intuitive problem that most people understand. Until recently, these models could not remember across chats, sessions, and applications.

2. Training Cutoff

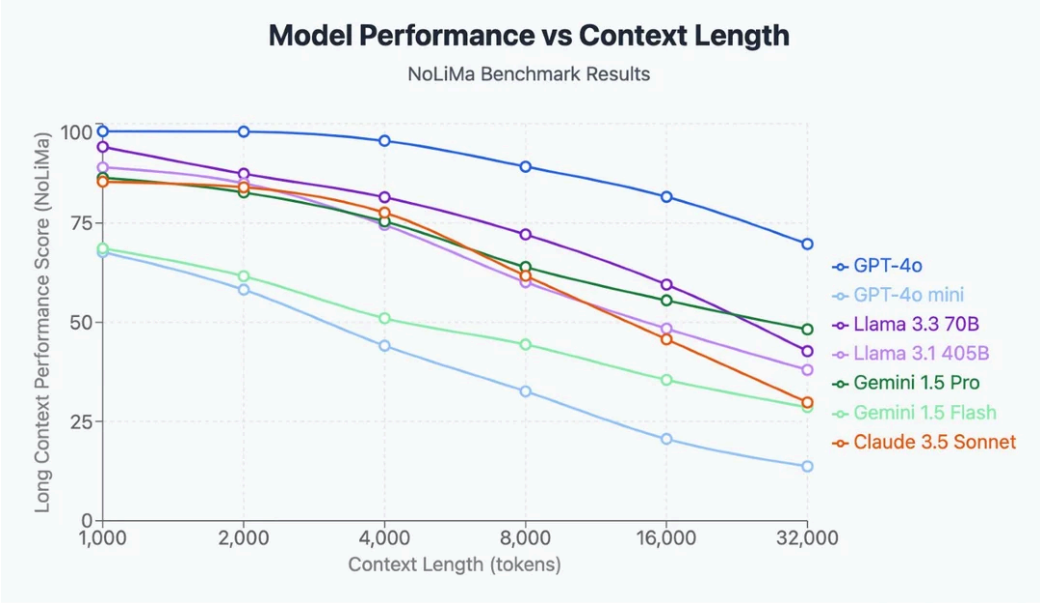
LLMs freeze at training time. GPT-4's knowledge ends in April 2023. Claude 3.5's knowledge ends in April 2024. Anything that happened after these dates—new research, current events, product updates, personal experiences—simply doesn't exist in the model's training data.

For generic knowledge work, this is manageable but inconvenient. For personal AI assistants or enterprise applications dealing with internal documents, it's fatal. The model has no access to the information it needs to be useful.

3. Context Window Degradation

While context windows have expanded from 4K tokens (GPT-3) to 128K (GPT-4 Turbo) to 2M+ (Gemini 2.5), research consistently shows that performance degrades with context length. Historically, performance falls precipitously after only 32k tokens, even with advertisements of ever-extending context windows.

- **The "lost in the middle" problem**: Models struggle to retrieve information from the middle of long contexts, even when that information is directly relevant
- **“Needle in haystack” failures:** Success on benchmarks related to retrieval of specific information within a large wall of context are often inflated and performance suffers in most practical use cases.
- **Attention dilution**: As context grows, the attention mechanism spreads activation across more tokens, weakening the signal from any particular piece of information
- **Latency and cost**: Processing scales quadratically with context length—doubling context quadruples compute



NoLiMa: Long-Context Evaluation Beyond Literal Matching

Models	Claimed Length	Effective Length	Base Score (×0.85: Thr.)	1K	2K	4K	8K	16K	32K
GPT-4o	128K	8K	99.3 (84.4)	98.1	98.0	95.7	89.2	81.6	69.7
Llama 3.3 70B	128K	2K	97.3 (82.7)	94.2	87.4	81.5	72.1	59.5	42.7
Llama 3.1 405B	128K	2K	94.7 (80.5)	89.0	85.0	74.5	60.1	48.4	38.0
Llama 3.1 70B	128K	2K	94.5 (80.3)	91.0	81.8	71.2	62.7	51.8	43.2
Gemini 1.5 Pro	2M	2K	92.6 (78.7)	86.4	82.7	75.4	63.9	55.5	48.2
Jamba 1.5 Mini	256K	<1K	92.4 (78.6)	76.3	74.1	70.8	62.2	52.7	43.6
Command R+	128K	<1K	90.9 (77.3)	77.0	73.5	66.3	39.5	21.3	7.4
Mistral Large 2	128K	2K	87.9 (74.7)	86.1	85.5	73.3	51.5	32.6	18.7
Claude 3.5 Sonnet	200K	4K	87.6 (74.4)	85.4	84.0	77.6	61.7	45.7	29.8
Gemini 1.5 Flash	1M	<1K	84.7 (72.0)	68.6	61.6	51.0	44.4	35.5	28.6
GPT-4o mini	128K	<1K	84.9 (72.2)	67.7	58.2	44.1	32.6	20.6	13.7
Llama 3.1 8B	128K	1K	76.7 (65.2)	65.7	54.4	44.1	31.9	22.6	14.2

Table 3. NoLiMa benchmark results on the selected models. Following Hsieh et al. (2024), we report the effective length alongside the claimed supported context length for each model. However, we define the effective length as the maximum length at which the score remains above a threshold set at 85% of the model's base score (shown in parentheses). Scores exceeding this threshold are underlined. Scores that are below 50% of the base score are shaded in red.

NoLiMa: Long-Context Evaluation Beyond Literal Matching

Simply dumping everything into context isn't a solution. The model drowns in noise.

The First Tool: Web Search as Memory

Before structured memory systems existed, the first "tool" that successfully extended LLM capabilities was web search. Even though most LLMs are trained on the Internet, the success of this tooling showcases the limitations inherent in training alone and the need to access context outside of the chat window.

The cadence of [prompt → external information store + retrieval → context injection] was set.

Google's web index was effectively the first "memory" system for LLMs and soon became native to ChatGPT, Claude, and the other large models. Notably, if you wanted to add search into custom AI models originally, there were no immediately available solutions.

The disentanglement: Initially, web search was tightly coupled to ChatGPT. But within months it separated. Perplexity launched as standalone search-augmented AI, You.com and Phind emerged with specialized implementations, Brave launched an API that could be cheaply integrated into every LLM on the market, and eventually web search became available as infrastructure that any LLM could use.

This disentanglement happened because specialized implementations outperformed generic ones, users wanted choice, and the capability wasn't proprietary. In the case of web search, this became commoditized rather than a core product differentiator.

Memory is not identical to web search. Implementations will vary widely. While we can't expect memory systems to follow this exact path, we do anticipate the disentanglement of memory.

The Early Memory Attempts: RAG and JSON

In order to create effective AI systems and applications that were more useful than chat bots, LLMs needed radical innovation in how they could interact with the outside world.

While LLMs could now read from the internet, they could not access other data stores, such as a company's internal documents. Constant hallucinations and poorly structured outputs also made it difficult for LLMs to call tools that require deterministic inputs. Knowing whether to search and what to search for in order to send that to a tool for searching ended up becoming a problem with structured output.

The Structured Output Problem

Early LLMs were not designed to produce or consume structured data reliably. Getting GPT-3 to output valid JSON quickly became a bottleneck in development. Models would hallucinate extra fields, break syntax with stray characters, or wrap JSON in conversational text ("Here's the JSON you requested: {...}").

This created a chicken-and-egg problem for memory systems: you needed structured outputs to store memories systematically, but the models couldn't reliably produce them. Early developers resorted to:

- Extensive prompt engineering with examples
- Regex post-processing to extract JSON from conversational responses
- Multiple retry loops when parsing failed
- Temperature tuning to reduce creativity (and thus syntax errors)

The breakthrough came from targeted training. By mid-2023, OpenAI introduced function calling in GPT-3.5-turbo and GPT-4, where models were specifically fine-tuned to produce

structured outputs for tool usage. Suddenly, getting reliable JSON clicked. The model had been trained to understand when it should output structured data versus conversational text. Anthropic's Claude models, especially Claude 3 onwards, excelled at structured output and context processing.

JSON-based memory patterns emerged from this capability: store user preferences and facts as structured data, inject them into the system prompt or context, and have the model reference them appropriately.

Early patterns looked like:

```JSON
{
  "user_preferences": {
    "dietary_restrictions": ["shellfish allergy", "vegetarian"],
    "communication_style": "concise and technical",
    "timezone": "US/Pacific"
  },
  "conversation_history": [...]
}
```

This worked for simple cases but revealed immediate limitations:

- Scaled poorly as fact count grew
- Required developers to decide what to store (no automatic memory formation)
- Couldn't capture nuanced information that didn't fit neat key-value structures
- Rigid schemas force information into predetermined categories, preventing models from using generalized pattern matching

Retrieval-Augmented Generation (RAG) emerged as the more sophisticated paradigm by early 2023. The architecture was conceptually simple:

1. Convert documents/memories into vector embeddings
2. Store embeddings in a vector database
3. When a query arrives, find semantically similar embeddings
4. Inject retrieved text into the prompt alongside the query

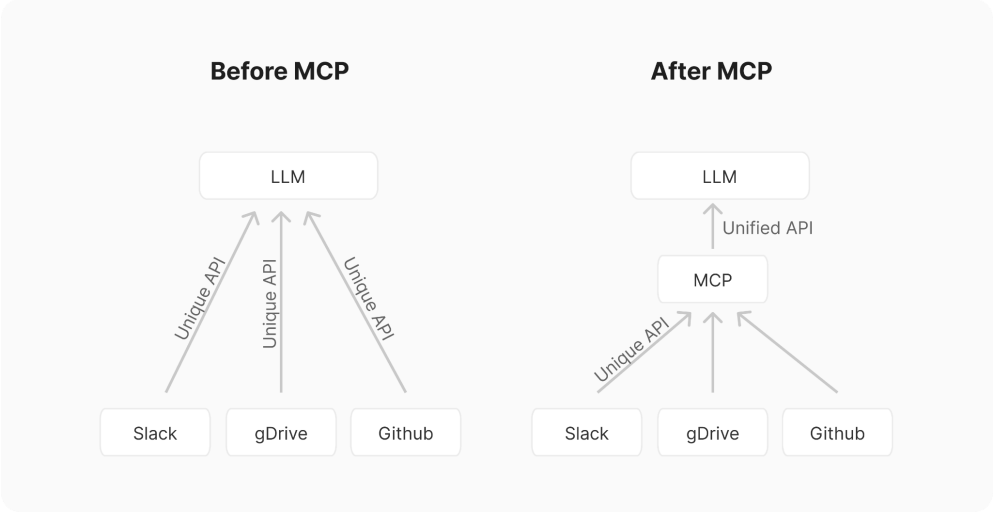
Implementing RAG well required solving multiple non-trivial problems. For instance, how do you know when to pull documents at all? How many? Often, these systems actually hurt intelligence by injecting completely irrelevant context. Some other issues.

- Chunking: How do you split documents so embeddings are meaningful?

- Embedding quality: Which embedding model captures semantic similarity best?
- Retrieval tuning: How many results to retrieve? What similarity threshold?
- Context packing: How to format retrieved information so models use it effectively?

Companies like LlamaIndex and LangChain built entire businesses on abstracting these decisions. Within months, implementing basic RAG went from requiring deep ML expertise to being achievable in an afternoon with a few API calls. But the ease of implementation masked ongoing challenges in retrieval quality and context utilization.

Tool calling and MCP emerged as another early pattern. Rather than trying to put everything in context upfront, give models the ability to request information on demand. This required yet another training breakthrough: models needed to learn when to call tools, how to format tool requests, and how to incorporate tool responses into their reasoning. MCP also promised to standardize the connection between all tools, so developers weren't forced to develop connectors for each.



Function calling became standard by late 2023, but effective tool use remained challenging. Models would:

- Call tools unnecessarily (wasting latency)
- Fail to call tools when needed (missing critical information)
- Misinterpret tool results
- Get stuck in loops calling the same tool repeatedly

Anthropic's Model Context Protocol (MCP), announced in November 2024, attempted to standardize tool calling for context retrieval.

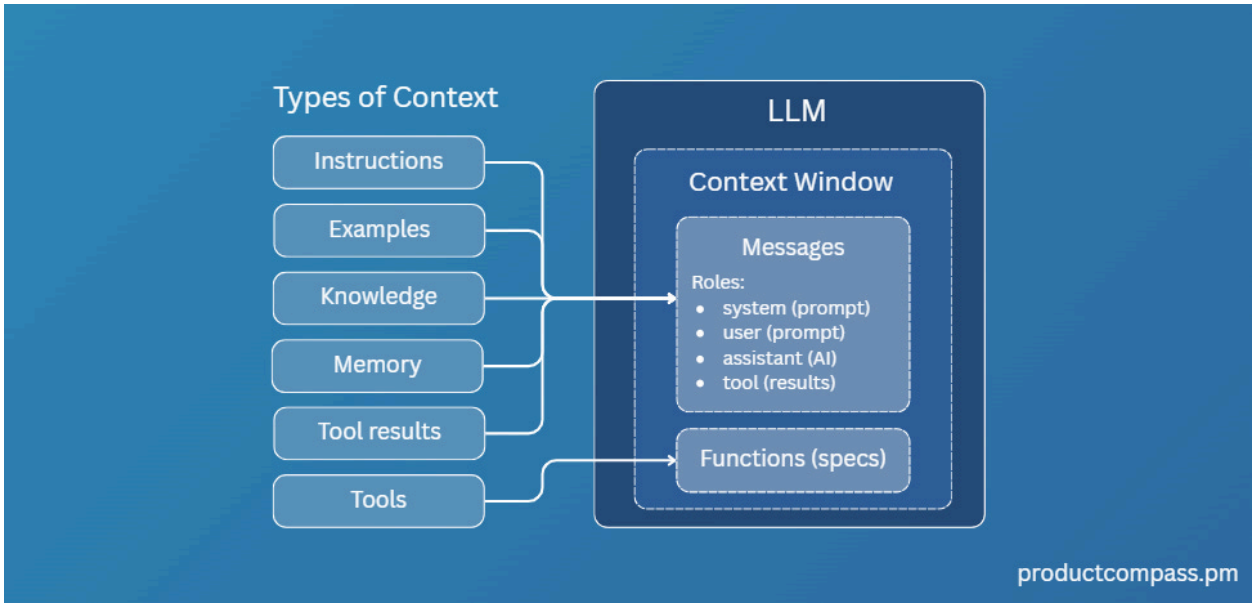
This reveals a subtle but extremely important theme in the context of AI Memory. It is not enough to be able to call tools. You actually have to build systems that are intelligent enough to know when to use what, how to use each tool, and then get rid of polluting context that distracts the AI from completing its task.

The Emergence of Context Engineering

Coined by Shopify CEO, Tobias Lütke, “context engineering” can really be summed up into “right context, right time.” Sounds easy, but it is a dynamic engineering problem involving the orchestration of many moving parts with resurfacing constraints around latency.

Early efforts revealed that memory systems required more than just storage and retrieval. They demanded **context engineering**: the practice of dynamically constructing and managing what information gets injected into the model's context window.

Context engineering is fundamentally a resource allocation problem: with limited tokens available, what information deserves inclusion? Every retrieved memory, every tool call, every piece of injected context consumes budget that could be spent elsewhere. For users of AI systems, we don’t like to wait. We want it to “just work” and fast.



While LLM is the central nervous system of AI. Memory and context management play a significant role in creating effective AI systems. Which brings us back to our pithy quote from the introduction that,

“Bad model with good context > good model with bad context.” [Vectorize]

Context engineering became distinct from prompt engineering:

Prompt engineering (2022-2024): Crafting the perfect static instructions, write once, use repeatedly

Context engineering (2025-present): Dynamic information assembly, what to retrieve, how much, in what order, formatted how, refreshed when

The Compounding Complexity Problem

By mid-2024, the technical building blocks existed. RAG, JSON storage, function calling and MCP, context engineering became household names in AI engineering. But making them work together remained brutally difficult. Each component introduced failure modes that compounded:

- RAG systems retrieved irrelevant documents that polluted context
- Function calling added latency while models decided whether to search
- JSON structures either oversimplified complex information or became unwieldy
- Context engineering required constant tuning as models and use cases evolved

By late 2024, a fundamental architecture had crystallized: **external storage + retrieval + context injection**. The sophistication varied wildly, from simple vector databases to multi-layered memory systems, but this basic pattern has held.

This standardization marked an inflection point. As we'll see, this opened the door to market segmentation and varied solutions.

Part 2: Technical Foundations

See [Part 1] for the history of how memory evolved to its current form.

Introduction

To recap Part 1, AI is no longer just a Large Language Model (LLM). We are now dealing with an entire "context engineering stack," in which AI Memory is a core component. AI applications and agents breathe context as oxygen and we can use memory to provide the right context at the right time to improve performance of AI systems.

We overviewed the humble beginnings of memory from basic RAG. Here, we will explore how this evolved into today's bleeding edge of intelligent, agentic memory. In this section, we will dive deep into evolved memory architectures and offer helpful design principles for building your memory systems.

This section will aim to be objective, but we do include helpful notes for developers that we have picked up from building many different memory products. We will also aim to drive home two opinionated takes:

1. **RAG is a transitional technology.** RAG was retrofitted for memory but will eventually be replaced by hybrid and agentic solutions. Current evals are broken and should be re-evaluated entirely. We stress that what is truly important for performance is recalling the "right context at the right time."
2. **Fragmentation of Solutions**: There's no "one size fits all" solution. We will continue to see different products emerging built for different use cases and tasks.

Section 1: Defining AI Memory

1.1 What Memory Actually Is

Defining AI Memory

Before exploring memory in greater depth, we should properly define it and establish a conceptual foundation. While companies like IBM have defined memory differently in the past, we introduce a new definition that captures all that memory is and can be [IBM].

"AI Memory refers to how AI systems encode, organize, and recall information from experience to improve task performance."

If we consider human memory, we see that the brain not only acts as simple storage and retrieval. The human mind filters, encodes, forgets, and constantly reorganizes itself.

- This functionally defines memory as a means to improve system and task performance.
- This definition does not limit memory to simple storage and retrieval but marks memory as an intelligent process of its own capable of deriving and triangulating new information.
- This definition purposefully restricts memory to information that has been experienced by the system, otherwise we would include the internet as memory. Memory is read/write.

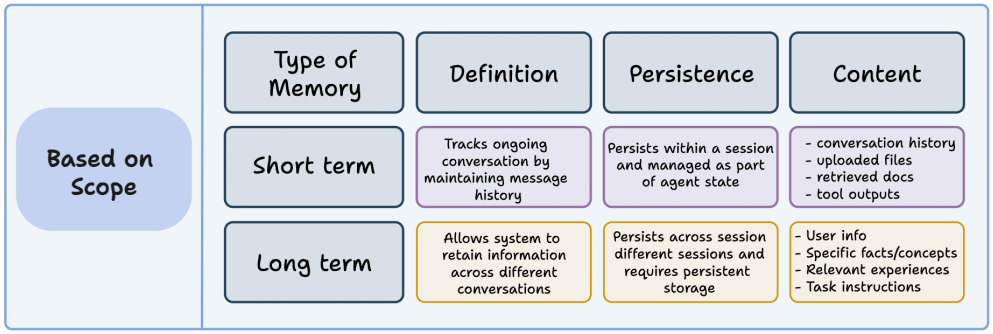
Not all memories are created equal. We will explore different forms of memory and types of context that we are encoding.

Short-Term and Long-Term Memory

As humans navigate the world, we are constantly flooded with information, such as the name of your waitress at IHOP or the date of your anniversary.

We remember information based on perceived future importance and memories degrade when not used over time. Information may be forgotten immediately, enter our short-term memory before being discarded, or become stored in long-term memory. While the future utility of your waitress' name may not hold much weight, forgetting your anniversary may have severe consequences.

The same is true for AI Memory systems. Most information that we experience is transient noise, but we should build systems that are capable of storing signals that will be of future importance.



Short-Term Memory

Generally, short-term memory is considered to be anything that has occurred in the chat or session, such as recent messages or tool calls that may be important to maintain continuity.

By appending recent context to new prompts, session history is maintained. The largest issue with short-term memory is context rot, in which a conversation lengthening means diluting an agent's attention span, often past 32k tokens.

There are emerging methods to improve short-term memory. Even as context windows expand past 1 million tokens, solutions that improve short-term memory include sliding context windows and storing older information in a separate retrievable cache, summarizing and compacting conversations, and maintaining a separate "scratch pad."

Even as context windows expand, there will always be performance improvements to be found in improving short-term memory through these methods. In the words of Anthropic:

"Waiting for larger context windows might seem like an obvious tactic. But it's likely that for the foreseeable future, context windows of all sizes will be subject to context pollution and information relevance concerns—at least for situations where the strongest agent performance is desired. To enable agents to work effectively across extended time horizons, we've developed

a few techniques that address these context pollution constraints directly: compaction, structured note-taking, and multi-agent architectures." [Anthropic, Context Engineering]

Long-Term Memory

While the goal of short-term memory is to improve session continuity, long-term memory is the persistent, externalized knowledge store that allows an AI to learn, personalize, and build understanding across sessions over days, months, and even years. This is the core focus of modern AI Memory systems and this overview.

For example, recent implementations of ChatGPT enable your AI to remember your preferences over time as well as across chat sessions. Customer service calls have historically started each session with a fresh slate but AI voice agents have the potential to remember who you are, your customer information, previous issues, and any other salient information relevant to serving you better.

Types of Memories

Not all memories are created equal. Rather than throw all memories into a giant pile and hope that we can build systems that make sense of the chaos, we can add some organization using principles based on human psychology. Generally the industry categories for memories are considered semantic (factual), episodic (temporal), procedural (process), and associative (linking across memories).

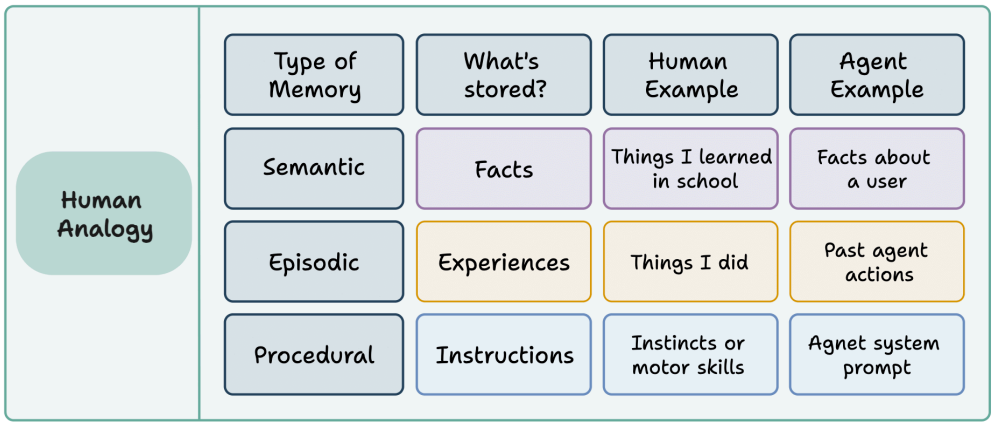
While categories provide useful frameworks, memories are complex and cannot often be contained by such simple organization. A single memory may contain episodic context, semantic facts, and procedural knowledge simultaneously. While this intuition is helpful, we don't necessarily need to implement strict categorization. Too much categorization makes memory brittle. Meanwhile, too little structure hurts performance.

This behavior mirrors the difficulties that Anthropic also calls out in engineering agentic systems. This tradeoff between brittle, complex structure and vague, high-level guidance is often the most important design consideration in engineering AI and Memory systems that we will explore in more detail in the technical principles section.

"At one extreme, we see engineers hardcoding complex, brittle logic in their prompts to elicit exact agentic behavior. This approach creates fragility and increases maintenance complexity over time. At the other extreme, engineers sometimes provide vague, high-level guidance that fails to give the LLM concrete signals for desired outputs or falsely assumes shared context."

[Anthropic, Context Engineering]

You'll notice that each memory type has different challenges. If you know what memory type will predominate your application, consider optimizing for it. Lack of foresight here can leave you months into building a complex architecture before realizing a much simpler architecture would have sufficed.



Semantic Memory: Factual Knowledge

Semantic memory stores facts and general knowledge that are true independent of when or how it was learned. This includes user preferences ("prefers dark mode"), professional details ("works as a product manager at TechCorp"), and domain knowledge ("Paris is the capital of France"). The key characteristic is that these facts exist outside any specific temporal context.

Common challenge: facts can become outdated as users change jobs, relocate, or update preferences, requiring systems to handle versioning and updates.

Episodic Memory: Specific Events and Experiences

Episodic memory captures particular interactions and events with their temporal and contextual details. Unlike semantic memory's timeless facts, episodic memories are inherently tied to when and where they occurred ("our conversation last Tuesday where we discussed the API refactor") or "you mentioned feeling stressed about the presentation last week." This memory type enables conversational continuity and case-based reasoning, allowing AI systems to reference past interactions naturally and learn from specific experiences.

Common challenge: often, we stamp memories with a date and time. But when do you update them if they repeat? What if you learn about a date of birth? What date do you use? What about general time ranges such as the 1970s?

Procedural Memory: How to Do Things

Procedural memory encodes processes, workflows, and learned behaviors. This includes operational procedures, such as communication protocols taken when a patient enters the ER, and personal workflows, such as starting every working session with a summary of your inbox.

Often learned through repeated experience or reinforcement learning, procedural memory allows agents to perform complex sequences efficiently without re-reasoning through each step.

Common challenge: procedural knowledge tends to drift and fork. The real process often lives partly in documents, partly in tools, and partly in undocumented team habits. This makes it hard to keep one canonical version up to date, detect when behavior diverges from the procedure, or safely evolve workflows over time. Encoding procedures too rigidly makes agents brittle; encoding them too loosely means they silently revert to ad-hoc behavior.

Associative Memory: Connections Between Memories

We are proposing a new category of memories that exist from forming connections between existing memories. Associative memory represents the web of relationships between different pieces of stored information. For instance, knowing that a user is working under Professor X and a separate memory that Professor X works at the Computer Science Department at Stanford may create a novel connection that the user is affiliated with Stanford. Often, associative memory excels when creating abstract understanding on top of individual memory data points. Such as a user who is constantly trying new things being categorized as high in personality openness and novelty-seeking.

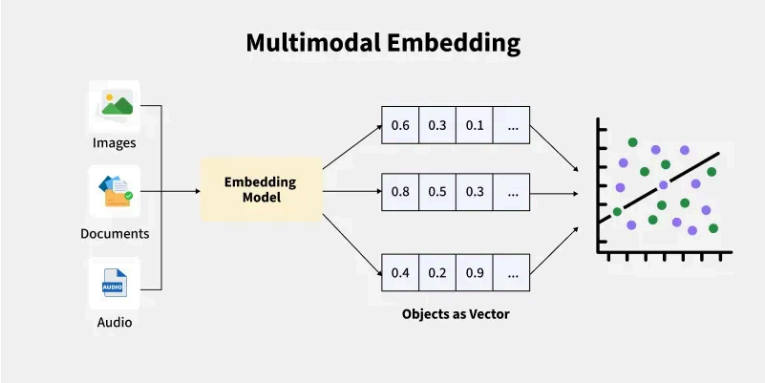
Common challenge: computers perform poorly at making proper connections between memories. Graph relational databases have proven to be too rigid when scope is broad. The nature of forming new memories is also unstable and may introduce false memories. This is why naive graph DBs struggle.

1.2 Types of Context

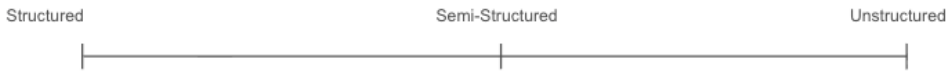
Memories can often be thought of as simple storage of context. When designing systems that should retain information from their experiences, we should consider that different systems are designed to navigate different environments and contexts.

By Modality:

Yoshua Bengio, AI researcher, recently stated that we likely shouldn't call modern AI LLMs anymore since they are really multi-modal and not language based. While they started as language-based, they can now take inputs of images, for example. [The Minds of Modern AI] In other words, memories are not necessarily just textual. They can be images, videos, or other types entirely.



Structure and Precision

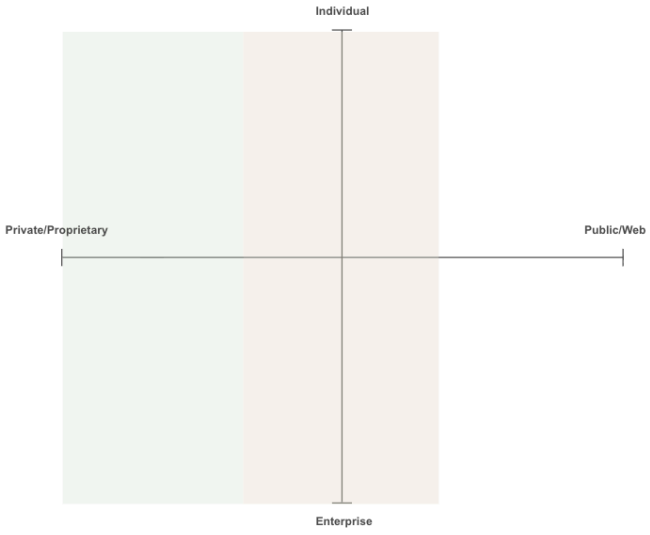


Note that there is a tradeoff between structure and unstructured context. We will also save much of this discussion for the section on technical design principles.

Some memories require strict rules in how they are saved and recalled. When dealing with legal documents or financial matters, we often need to be exact and can not afford to lose precision in understanding. Many professionals build automations that reuse templates that are core to a firm's value proposition and intellectual property. For tasks that require this precision, we must build strict systems. We may prompt systems to format and verify text exactly as it is stored and to interact with the memory system using JSON formatting.

On the other hand, when creating a memory system that recalls previous zoom conversations, we may at first design the system with the ability to quickly review summaries of various conversations. “What did I talk about with Josh Brener last week?” Too much structure may unnecessarily hinder performance.

Source and Propriety



Context may be categorized across axes of propriety (private or public on the web) and what entity it relates to, whether an **individual** or **enterprise**.

- Individual memory is often heavily semantic, unstructured, and related to factual data and preferences about the individual or user.
- Enterprise information is often procedural, structured, and related to workflows, Slack, internal data, etc. Enterprise memory is often heavily dependent on integrating with internal applications where value can be unlocked by connecting context across siloed data stores.

There is a significant amount of context residing on the web and accessible by tooling such as web browsers. This context is largely commodified and not very valuable. However, we can still build agentic systems that navigate the web and use a memory store to complete tasks. The reason why we have purposefully restricted the definition of memory to information that is experienced by the AI is that there are separate tools more suited for dealing with this commodified information (including basic web search).

The line between context related to an enterprise or individual is often blurry. Each individual has a “*severance effect*” where context related to work is separate from the rest of their daily context. Even within each category, we often segregate context further between individual projects.

Work Self: This context is professional and task-oriented. It includes your projects, code, meetings, professional relationships, and team dynamics.

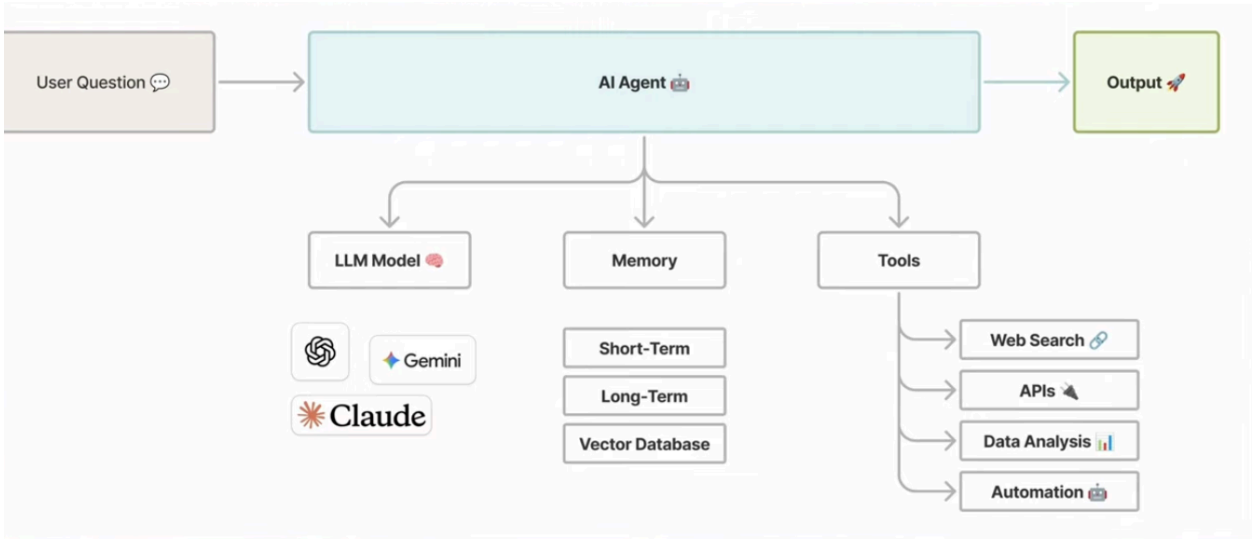
Life Self: This context is personal and private. It includes your preferences, habits, personal history, family relationships, and private thoughts or goals.

As we move further to the left in the green shaded area, where individual and enterprise data becomes more proprietary, context becomes more valuable. As you can imagine, sensitive individual and enterprise data leads to privacy considerations and consumer protections.

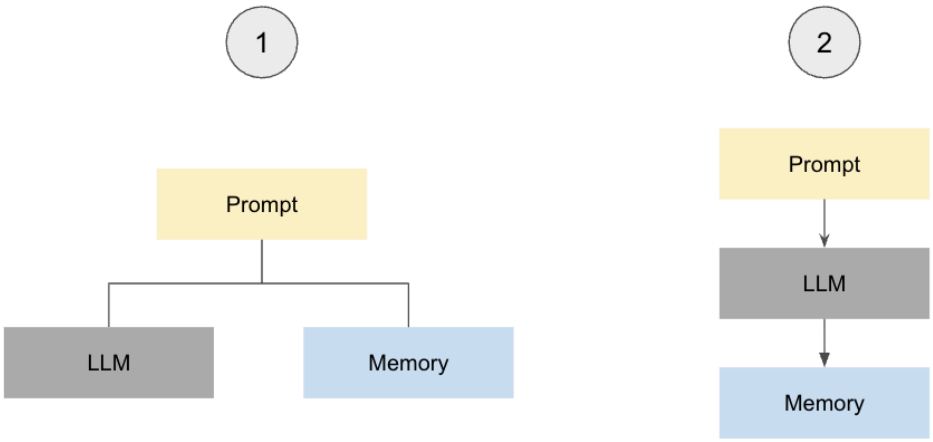
It took years for many companies to scaffold secure LLMs running in the enterprise due to risk of leaking proprietary information. We should anticipate similar precautions with memory systems.

High-Level Memory Processes

To provide a general outline, we can break memory into distinct processes. Pragmatically, we will write this section through the lens of modern agentic systems, with a simple depiction below.



Before we jump into each individual process, we first need to decide “how” and “when” a memory is passed along. There are fundamentally two decision pathways here.



Pathway 1 runs parallel, where we pass along the full prompt to the LLM and memory system in tandem. We can make our AI system faster and more efficient by designating this as a separate background process that does not interfere with LLM inference. In many cases, memory systems must validate if there are conflicting memories to reconcile. This process may take time and will create a bottleneck that we prevent by keeping processes independent.

We can pass along every message in full or process memories before sending them to memory. Processing can be agentic/intelligent or programmatic/rule based to decide what is sent. For instance, if we always want to remember when customers send messages with item numbers, we can program this directly.

Pathway 2 is unified, where the memory system itself presents as a tool to be autonomously called by the LLM, often through the model context protocol. An example of this is coding on Cursor. A coding agent may enter a loop to fix a bug. After a major milestone has been reached, the milestone may be deemed meaningful to save down as a progress report and the coding agent will call memory directly.

This method is especially useful when the context of the entire chat or session is required for deciding what to save and when. An example of this is Claude's current memory.

Memory is a 3-Part Process

To understand where current systems fail, we must move beyond the idea of memory as a static storage system. Memory in AI is a computational pipeline designed to recall *“right context, right time.”*

On this point, many memory products now advertise scores on retrieval benchmarks like LoCoMo. LoCoMo is a question-answering benchmark over long, synthetic conversations: it measures whether an agent can pull the right fact from a transcript, not whether it has “good memory.” Letta’s work shows that a simple filesystem-based setup reaches 74% accuracy with GPT-4o mini, outperforming specialized “memory” stacks. [Benchmarking] Zep also accused Mem0 of similar benchmark hacking and poor testing. As we move into context-heavy, task-specific agents, LoCoMo is best treated as a useful smoke test, not a sufficient eval; systems should be judged on end-to-end task performance for the actual use case.

On this point, while many memory products grade themselves on evals such as LoCoMo, this eval is often too naive and exploited by overfit products. As we enter a regime defined by context rather than retrieval, this is simply an outdated eval. Each task is very different.

We can break this pipeline down into three relatively independent downstream phases:

1. **Experience (E)**
2. **Storage (S)**
3. **Recall (R)**

Ultimately, the quality of recall for a defined task is what makes an effective memory system. This is an important point, because recall must also be designed with the agent's unique **Task (T)** in mind. There are no perfect evals other than your unique case.

Here is an overview of what these abstract processes look like in a programmatic flow.

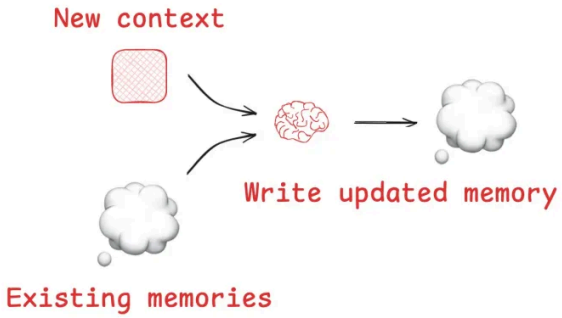
```python
graph TD
    A[User Interaction] --> B{Decision 1: Pathway}
    B -- Pathway 1: Parallel Process --> C[Background Listener]
    B -- Pathway 2: Active Tool Use --> D[Agent Calls 'Save_Memory']

    subgraph "1. Experience (The Trigger)"
    C --> E[Filter & Extract]
    D --> E
    end

    subgraph "2. Storage (The Organization)"
    E --> F{Processing}
    F -- Semantic --> G[Update User Profile]
    F -- Episodic --> H[Vector/Time-Series DB]
    F -- Associative --> I[Update Knowledge Graph]
    G & H & I --> J[Consolidation/Forgetting]
    end

    subgraph "3. Recall (The Synthesis)"
    K[New Query] --> L{Retrieval Strategy}
    L --> M[Fetch Context]
    M --> N[Rerank & Synthesize]
    N --> O[Inject into Prompt]
    end
```

Phase 1: Experience (Input Layer)

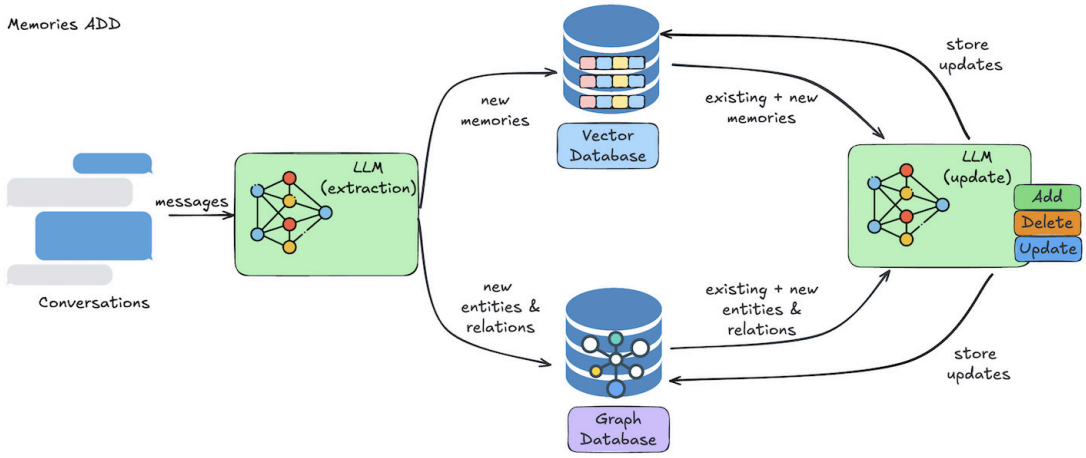


Experience is the point of contact between the data stream and external memory. This is where the system decides what is worth remembering. In 90% of current industry implementations, this step is passive and the system simply passes along every interaction. This is a mistake.

Effective memory systems distinguish between signal and noise at the point of entry. Remember, we want to create systems that distinguish what will be useful in the future. We use memory in the first place to fight against context pollution. We do not want to also pollute our memory store.

High-performance memory is non-blocking. While novice systems force the agent to pause and save data, efficient architectures run as an asynchronous background process that distills insight without introducing latency. We can avoid this bottleneck by running a secondary background process that scans the interaction for valuable signals. It de-duplicates repeated information, discards conversational filler, and extracts core facts or intent before they ever reach storage.

Phase 2: Storage (The Organization Layer)



Once information passes the gate, it must be organized. In our everyday lives, we expect our clothes to be in our closet. When they are in the pantry, it confuses us and leaves us stuck. The same goes for memory systems. We must organize memories for future utility.

This layer is responsible for the architecture of storage. It answers questions like:

- Does this new information contradict old information?
- Is this a standalone fact, or does it modify an existing memory?
- What entities or other memories might have relationships with this memory?
- Should we log this as a precise fact, a summarized memory, or link it to an artifact?

We will explore the specific technical architectures (Vectors, Graphs, etc.) that achieve this in the next section, but the high-level goal remains constant: building memory that recalls “right context, right time.”

Sleep-Time Compute

Sleep-Time Compute is a term coined by Letta. [STC] The human mind reorganizes our memory while we sleep. Well-designed memory does the same. We are not only storing. We must also edit, reorganize, compact, prune, distill, etc. to make sure that our storage (S) is always organized and ready for recall (R). If we expect there to be some upper limit for the amount of memories that a system can handle, we must design our systems to balance the adding of new memories with keeping space free for new memories.

Phase 3: Recall (Retrieval Layer)

If storage is making sure everything is ready and findable, recall is the process of querying and retrieving this data. It is where the system pulls the information that is useful for answering the task at hand.

While most modern memory systems do nothing more than simple retrieval, we make the argument in this paper that creative synthesis and reasoning on top of memory data points will be critical in the design of next-generation memory products.

Our rationale is that if the goal of memory systems is to optimize Recall (R) for Task (T), and we can achieve more accurate recall by reasoning over memory data points, we should.

Levels of Recall Depth:

1. **Static Context Injection (list_memory):** The system blindly dumps the last ***N*** messages or pinned notes into the prompt. Fast, but zero intelligence.
2. **Semantic / Graph Retrieval (search_memory):** The system takes the user's query ("Help me fix this bug") and searches the database or prior messages for mathematically similar concepts to the message or relevant words such as “bug”.
3. **Intelligent Query Generation (deep_memory):** The system pauses to ask: *"What do I need to know to answer this?"* Then locates and packages important information for the agent.
    - Includes *iterative retrieval:* Sometimes the first retrieval reveals that we are asking the wrong question. An agentic memory system allows for multi-hop retrieval: fetching A, realizing A requires B, and then fetching B.
    - *Example:* If the user asks "Why is my deployment failing?", a naive system searches for "deployment failing." An intelligent system generates new queries: "What is the user's tech stack?", "What were the last 3 error logs?", and "Check recent changes to config files."
    - There is no reason that we can not build multi-layer, long-context, coordinating agentic memory that we coin “deep memory.” It is just often prohibitively slow or expensive.

To summarize, the ultimate measure of a memory system is not how much it remembers, but how well it uses stored information from experience to improve performance of the agent’s task. We should design our memory pipeline to optimize Recall (R) for the defined Task (T).

Section 2: Technical Architecture

2.1 Overview:

The technical foundation of memory and its pipeline is set. But different uses of memory require different architectures for their specific use cases. Some may prioritize latency and semantic precision while others prioritize intelligence, quality of recall, and the maintenance of relationships across memories.

2.2 Technical Design Principles

Before we choose between specific tools (like Vector vs. Graph), we must understand the constraints that lead to our decisions between them.

To evaluate these architectures objectively, we have synthesized a unified theory based on **Politzki's Law**, **The Memory Frontier**, and **The Routing Constraint**.

These formalize the chain of logic behind every memory decision:

1. **AI's Limitation:** AI struggles with complex problems. We need relevant information to help our AI navigate problems and tasks (Politzki's Law).
2. **The Constraint:** Recalling context through inference is expensive at scale, so we must partition it (The Memory Frontier).
3. **The Optimization:** To find data within these partitions, we need a clean organizational ontology (The Routing Constraint).

1. AI's Limitation: Politzki's Law of Complexity

To set the stage, we lay out the limitations of AI. While humans are able to find solutions to complex problems with limited information, AI struggles to generalize at the same level. We define this relationship as [Politzki's Law].

Simply put, we can't easily improve the intelligence of the LLMs we use, so the only variable we can reasonably improve is the context provided to the system. Memory is an important tool to do so.

1. **The Premise:** Modern AI lacks Generalization (**G**), the ability to reason with limited and disparate information, relative to humans.
2. **The Problem:** When Complexity (**C**) of problem space is high and Data (**D**) is low, AI fails because it cannot generalize (**G**) a solution like a human can.
3. **The Solution (Memory):** Since we cannot readily increase a model's generalization (**G)**, we are forced to use **Memory** to provide useful information (**D)** by injecting relevant context with the aim of maximizing performance of task (**T)** by effectively guiding the model to find more appropriate vector programs for the task.

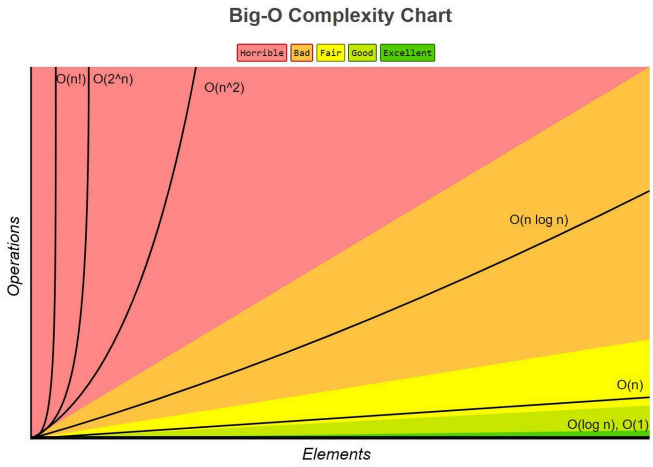
2. The Constraint: The Memory Frontier

In order to retrieve the correct Data (D), we need to build an effective recall system. Most engineers frame memory as a database selection problem ("Vector vs. Graph"), but this is not the correct framing and perhaps the most dangerous misconception in today's discussion on AI Memory systems. In an ideal world, we would recall only via inference.

For example, imagine a scratchpad containing every event on your calendar. The most elegant solution is to feed this entire context into an LLM and ask, "What time is lunch today?" This guarantees 100% recall because the model "sees" everything simultaneously and picks it out efficiently.

While we aspire to throw everything in the context window, the cost of attention scales quadratically $O(N^2)$ as the number of tokens (N) expands. This results in:

1. Increased latency.
2. "Lost in the middle" phenomena (forgetting).
3. Prohibitive cost.



To mitigate this, we commit the **"original sin"** of memory: we segment context to lower search space.

We start partitioning by introducing structure via directories, embeddings, and graphs. By dividing N tokens into P partitions, we eliminate the quadratic scaling cost of context. However, this introduces the problem of **fragmentation.**

Fragmenting context severs connections between context. For instance, if you have a memory that you have lunch today at 1 PM, but you have another work-related memory from a different partition that a client requested you reschedule and you agreed, the two cannot be reconciled.

The Efficient Frontier

Therefore, memory architecture is not just about choosing a database. A database and framework is only as helpful as how well it balances **Partitioning** and **Recall (R)** quality along the memory frontier:

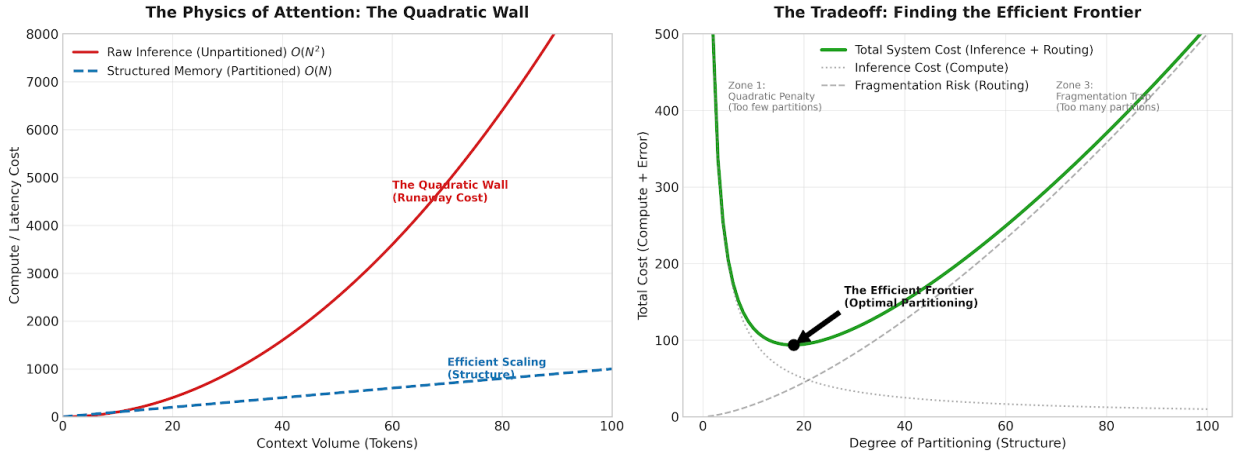
Fast Memory (High Partitioning)

- *Tech:* SQL, Key-Value, Vector RAG.
- *Physics:* Splits context into thousands of tiny shards.
- *Pros:* Linear scaling, instant retrieval, low cost.
- *Cons:* **High Fragmentation Risk.** Logical connections between shards are severed. If vector similarity fails, the context is lost.

Slow Memory (Low Partitioning)

- *Tech:* Agentic Reasoning, Long-Context Inference.
- *Physics:* Maintains large, contiguous blocks of context.
- *Pros:* **High Synthesis.** Can connect disparate facts (e.g., Clause A + Clause B) because they are processed together in inference.
- *Cons:* Expensive and slow.

We should aim to find the **Minimum Viable Partitioning**, which segments data enough to enable fast recall, but keeps partitions large enough to preserve logical connections.



3. The Optimization: The Routing Constraint

The above principle tells us that there is a tradeoff between computational cost and the quantity of partitions. We also need to consider how we organize our memories amongst our partitions.

For example, we may find optimal partitioning=10. However, how do you distribute context across them? We should design the directories so that they each contain approximately the same quantity of tokens *N*. This distributes the search load across each partition and improves the quality of Recall (R).

We define this as **The Routing Constraint**:

"Partitioning effectively lowers search space but introduces fragmentation. To minimize the cost of fragmentation, we must apply efficient organization to maintain logical connections across context."

This segmentation is often more of an observational art than a science. A user's memories may fall under "work," "personal preferences," "family," "recent context," so we should design the system with these partitions. However, for very general products, this is not always straightforward as different users use the product in drastically different ways. Giving the agent the leniency to self-organize through dynamic partitioning is an active area of research for us.

Example and Summary

Imagine a debugging agent trying to diagnose a production outage. By Politzki's Law, the model alone cannot generalize a fix from a vague error message ("502s on checkout") without more Data (D): it needs logs, recent deployments, runbooks, and incident history. The Memory Frontier forces us to partition that context so that it can realistically process context. The Routing Constraint then tells us we need these organized into logical categories: logs in one store, PRs in another, and runbooks in a third.

To summarize, AI relies on relevant context. We can use memory to make sure that it is recalling the right context at the right time. However, as context size N increases, we need to partition P the memory set to ensure that the cost of operations are not prohibitively expensive. To optimize this further, we need to make sure that our organization distributes context across categories to distribute the search load evenly.

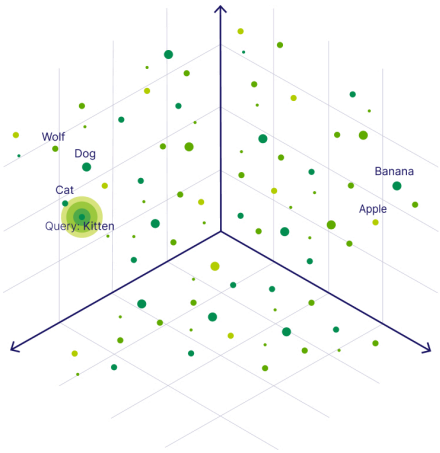
2.3 Technical Architectures

As we see above, technical architectures are a means to an end. For low-context tasks, a simple file paired with inference suffices. But as memories grow and become more complex, architecture matters.

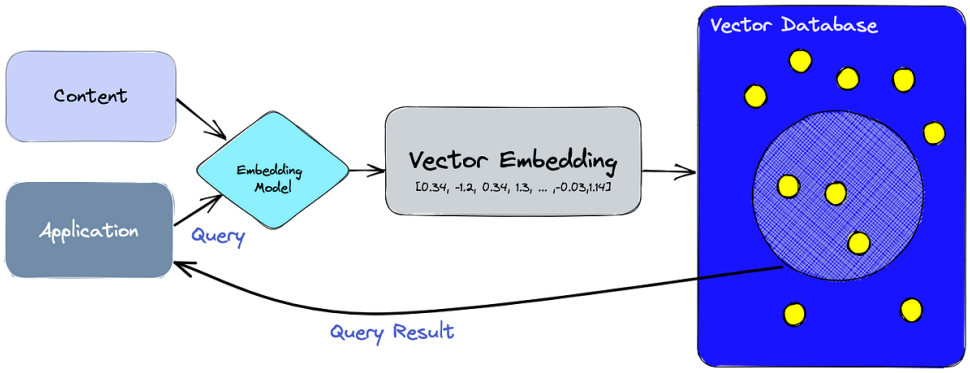
Vector/Embedding-Based Systems (RAG)

In part 1, we explored the history of memory. Retrieval Augmented Generation (RAG) was the first solution to be retrofitted to be used as memory. Notably, this development came before the introduction of MCP and effective structured output. We believe this led us down a path dependency that should be reconsidered. [RAG is not Memory] RAG was the obvious initial solution for memory. It is excellent for retrieving facts, but terrible for deriving understanding.

Today, RAG remains the most popular architecture. Vector databases work by converting text chunks into mathematical vectors (lists of numbers) and plotting them in multi-dimensional space. When a user asks a question, the system finds the "nearest neighbors," the chunks of text that are mathematically most similar to the query. See below that “wolf,” “cat,” and “dog” all live near each other and are ranked similarly when searching for “kitten.”



Vector databases maximize partitioning while maintaining a clean, fast search for closely related data points. However, saving memories in embedding space is far from perfect. Embedding models mostly capture semantic meaning and much is lost in translation when converted to embeddings, especially with larger chunk sizes. Moreover, when you move data into high-dimensional embedding space, you strip away the adjacency that exists in a contiguous block of text. You may find the precise "chunk," but miss the logical neighbor required to make sense of it.



Graph-Based Memory

Graph databases (Knowledge Graphs) attempt to solve the fragmentation problem by enforcing structure. Instead of isolated chunks, data is stored as **Entities** (nodes) and **Relationships** (edges).

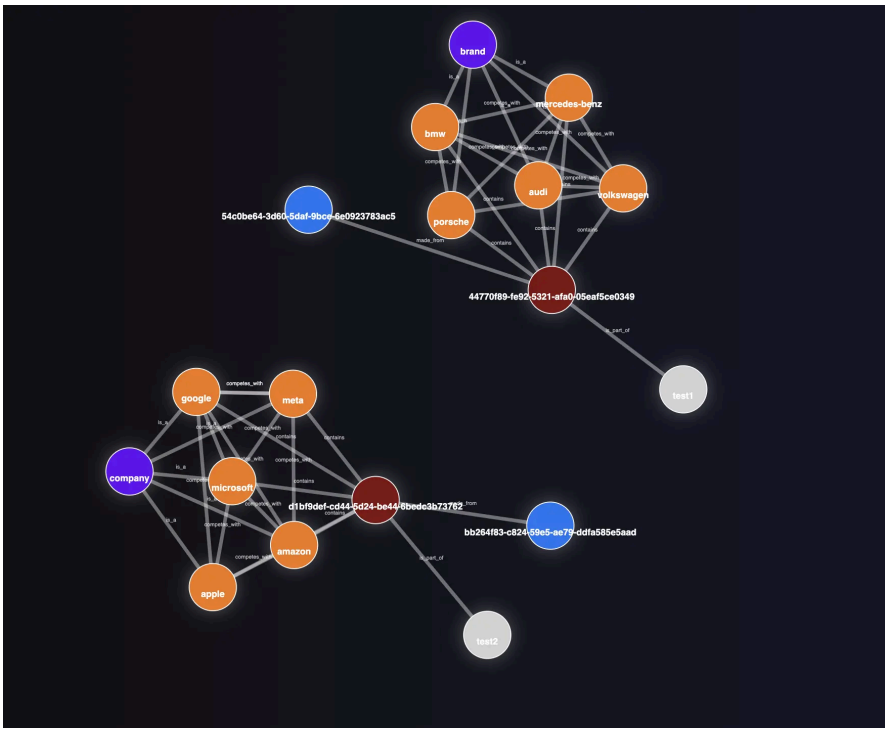
“Alice” [WORKS_FOR] “CorpX”.

“Jordan” [WORKS_FOR] “CorpY”.

In theory, this allows for "multi-hop reasoning" traversing from Alice, to her Employer, to her Employer’s Stock Price, in a way that Vector search cannot.

Graphs are conceptually elegant but difficult in practice, since you are effectively trying to brute force a relational structure of the world. Graph memory can work well in narrow, bounded domains where entities are rigid and predictable. If you are building a bot to query an inventory system (SKUs, Prices, Warehouses) or an HR system (Employees, Departments, Managers), a graph is unbeatable. However, conversations are messy. Trying to force the complexity of the real world into a rigid graph schema results in a brittle system that breaks the moment a user says something that doesn't fit the pre-defined ontology.

While not explicitly built for temporal purposes, some architectures claim to be able to capture date and time for temporal tracking (when memory was saved and when it occurred) through metadata stamping. In practice, this really just means stamping metadata of time with memories and doesn’t perform extremely well and bloats context.

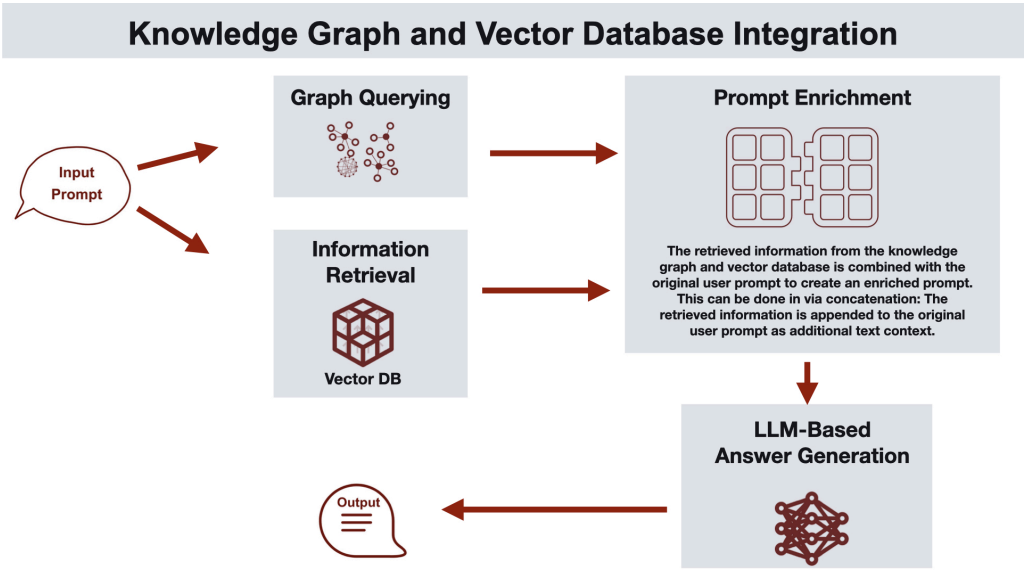


Hybrid Systems

The industry is converging on Hybrid approaches. The most promising patterns emerging in 2025 were combining **Vector + Graph (GraphRAG)** and **Agentic RAG.**

GraphRAG

While GraphRAG is often touted as the final state of AI Memory, we do not believe this will be the case. It frequently inherits the weaknesses of both systems. Adding a rigid graph to a broad, messy agentic task adds latency without improving recall. Many companies have emerged as a simple wrapper around a Graph + RAG system, but we find that they are often over-engineered and ineffective. A use case where this is valuable is when a company has a large quantity of memories where graph relationships are narrow enough to harness. In other words, when complexity is still manageable and applications have narrowly defined tasks.

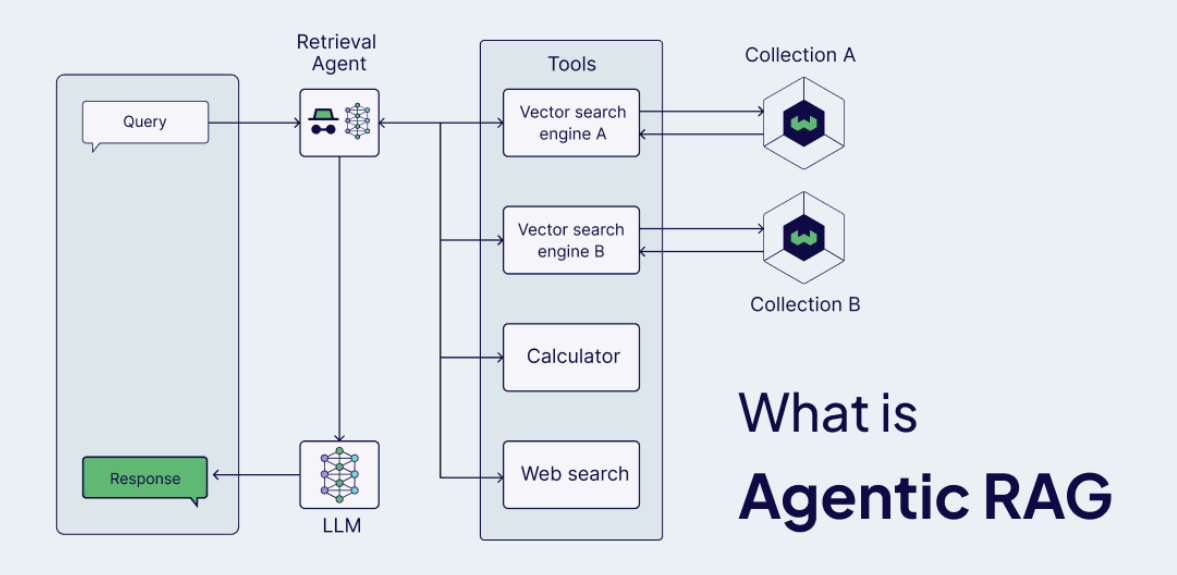


Agentic RAG

Agentic RAG is effectively running an LLM over the output of a RAG's search. The system retrieves a broad set of potential memories using Vector search (casting a wide net), but instead of dumping them directly into the final prompt, it passes them through a cheap, fast "reasoning layer" (like Gemini Flash or Haiku). This intermediate layer acts as a sanitizer, checking if the retrieved memories are actually relevant before passing them to the main model.

This was the "sweet spot" we found when building more modern memory systems where memories per entity exceeded 50,000 tokens. Standard RAG suffers from a lack of intelligence and an abundance of context. You are often forced to decide between a targeted search that may not pull the correct memory or a broad search that returns too much context and distracts the agent. By using a cheap, fast model to distill the retrieved memories first, we mimic the human ability to glance at a notebook and decide what is relevant *before* starting the task. It balances the speed of vectors with the discernment of an LLM.

But even Agentic RAG, we believe, is simply a precursor to futuristic agentic memory systems.



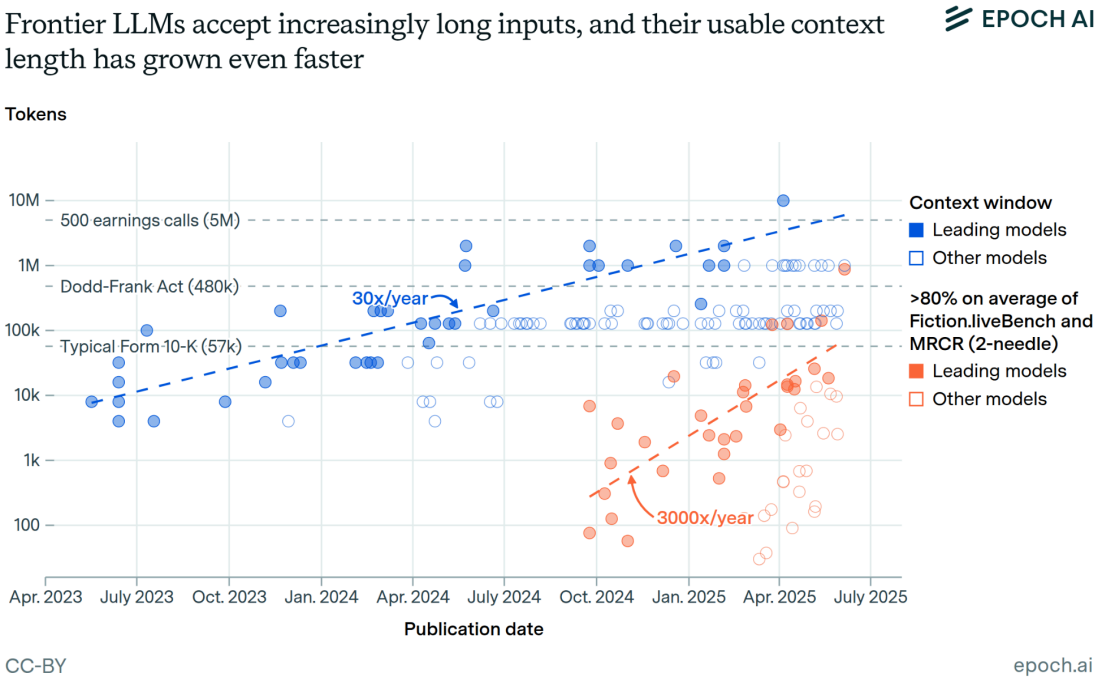
2.5 Long-Context as "Memory"

With context windows expanding to 1M+ tokens, some argue we don't need memory systems at all. Just dump the entire history into the prompt every time.

While effective for some one-shot tasks (e.g., "summarize this 200-page PDF"), it is economically and technically ruinous for persistent memory. Cost scales quadratically which translates to increased price per model call and Gemini models may run for 45+ seconds before returning.

Paying to re-process the same 500,000 tokens of history for every single "Hi, how are you?" message is unsustainable. Second, the "Lost in the Middle" problem is very real and proclaimed industry benchmarks on retrieving needles in haystacks are often overblown. [Arxiv] Attention is a finite resource.

One task we have found to be very useful is in running “deep memory” background jobs across all memories to create up-to-date summary for an AI. We can then store this summarization into a cache that is called at the beginning of every conversation. This provides you the processing depth of long-context without the recurring charges and 45-second bottlenecks.

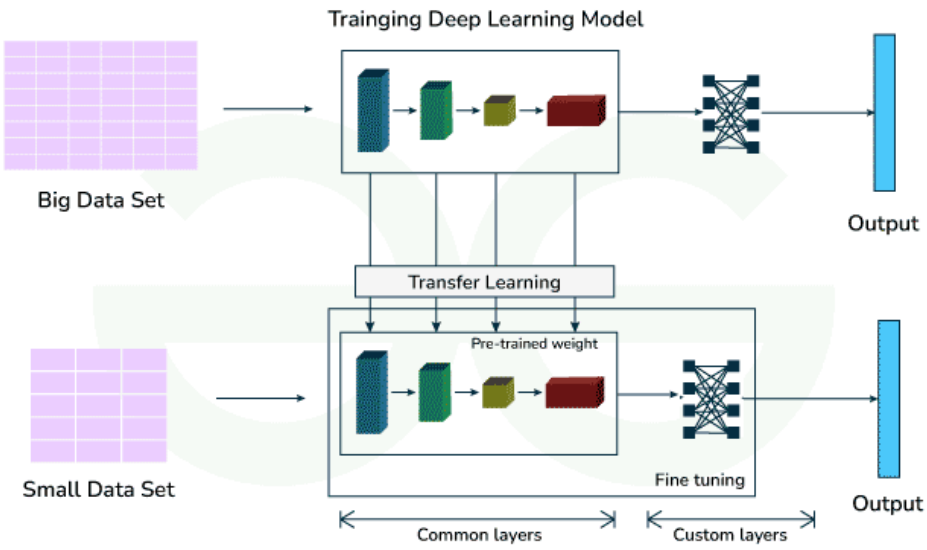


2.6 Fine-tuning for Memory

While theoretically possible to fine-tune a model so that the parameters themselves hold knowledge/memory, it is almost never practical to do so.

Fine-tuning is for teaching a model *how to speak* (style, format, tone), not *what to know*. Baking facts into model weights is the most expensive, rigid form of memory possible. If a fact changes (e.g., a user moves to a new city), you cannot simply update a database row; you must re-train the model.

This is useful, however, for procedural memory that is not straightforward how you might build memory systems for. For instance, if Google wants to make sure their AI coding agents follow general company design principles, they can bake that into their models by training on company code.



2.7 Agentic Memory Systems

Intelligent Orchestration

Modern

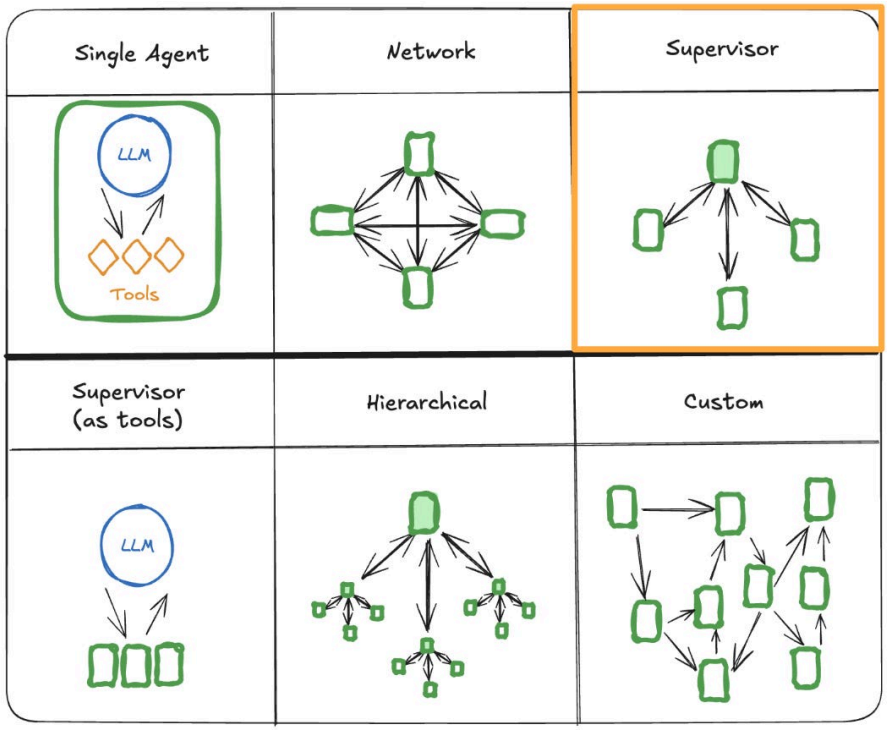
Until now, memory was synonymous with a static database. Eventually, memory will flip from storage and retrieval to intelligent reconnaissance agents that reason over stored memories, find useful information and compact it to just the right size for the agent, similar to how a chief of staff might support a senior executive.

Over time, we project latency and cost of inference to approach zero, where agents will overtake RAG in efficacy. Memory will work similar to how deep research is used across the web today, yet instantaneous.

1. **The Scouts:** You ask a complex question. Instantly, 50 lightweight agents spawn. They don't just "search." First, they ask what would be helpful to know before responding, they think for themselves how deep they should search. Finally, they split up and read across your memories. One agent reads your last 100 emails, another scans your Slack, a third reads the codebase.
2. **Iterative Discovery:** One Scout finds a Jira ticket mentioning a "migration bug." It doesn't just return the text; it *realizes* this is important and triggers a secondary wave of agents to search specifically for "migration bug logs."
3. **The Synthesis:** These Scouts feed their findings to a "Deep Reasoner" (like Gemini 2.5 Pro or OpenAI o1), which spends 45+ seconds (or minutes) thinking, filtering, and synthesizing the answer.

This is the "end game" and “meta” of memory. Deep Memory Swarms assume the question is just a starting point for an investigation. Agents will determine how deep to search as well as

how and where to route to find useful context. Even this coordination process will have different orchestration architectures.



Opinionated Synthesis of Architectures

While different use cases may find each of these architectures useful for different reasons, usually there are only a couple that are useful today. Graph, RAG, hybrid, and agentic memory when effective.

If you're building memory in 2025, start simple. Build a simple agentic memory across directories. If context bloats, add a separate vector database.

Section 3: Market Participants & Solutions

3.1 Overview: Mapping the Stack

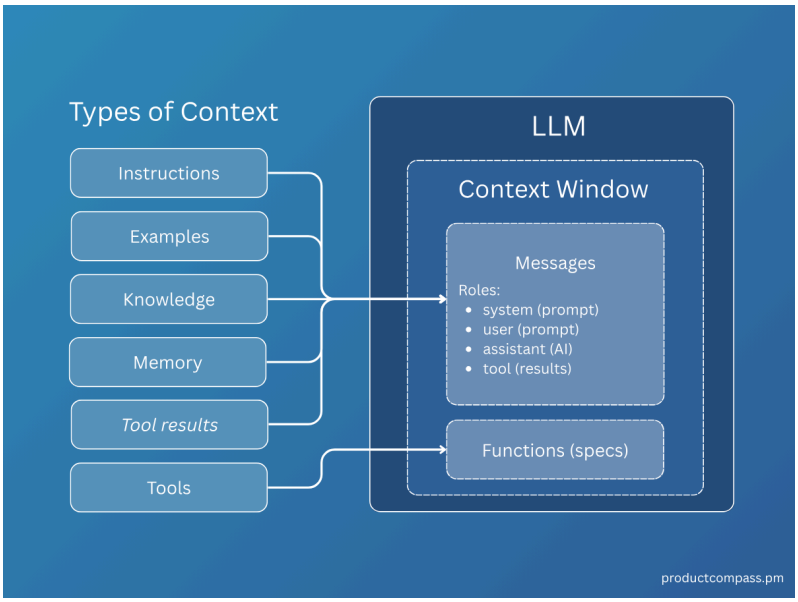
For the final section of this overview, we map how memory fits into the broader AI ecosystem. Memory is not a single tool; it is a stack within a stack. Before we discuss use cases and providers of memory (Part 3), we will map out where it lives in the AI stack.

Originally, AI applications were viewed as simple "Wrappers," thin interfaces built around a core model like GPT-4.

However, as models commoditized, the "Model Layer" shrank in importance. It has been replaced by the **Context Engineering Layer**. This is the new nervous system of the modern AI application, responsible for orchestration, tooling, and crucially, **Memory**.

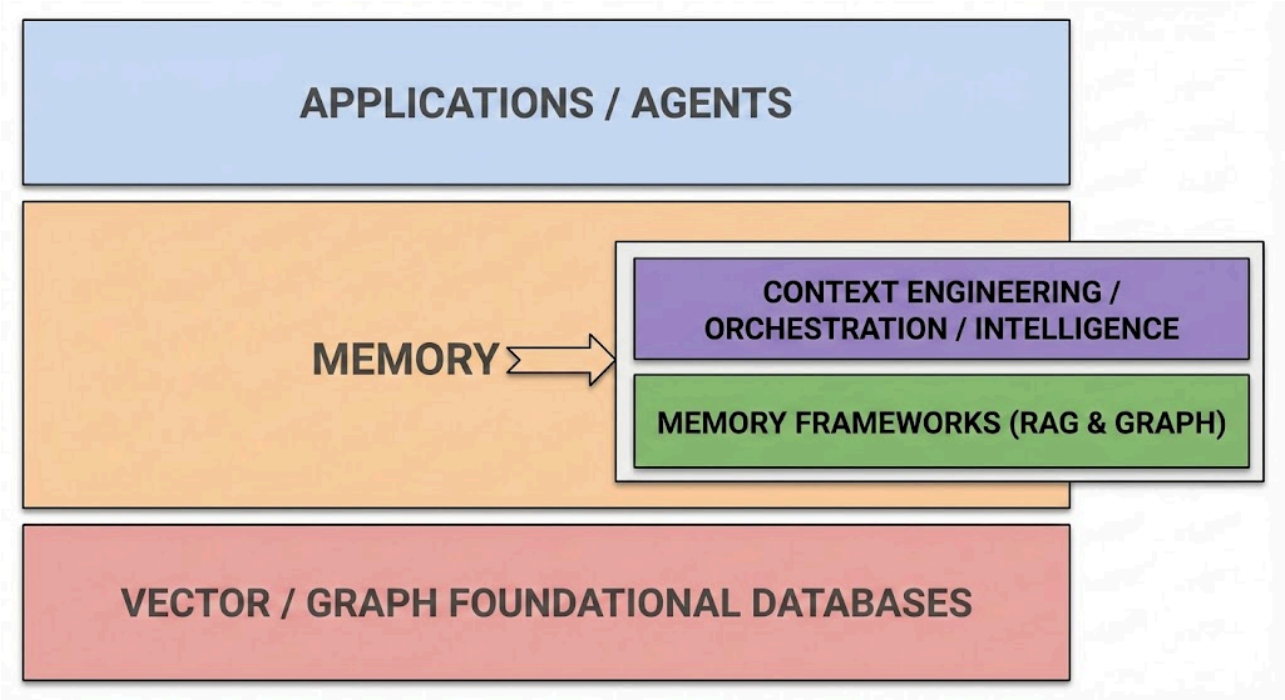
The New AI Stack

1. **Application Layer:** The interface (Chatbot, Agent, Dashboard). The user's touchpoint.
2. **Context Engineering Layer:** The "Brain." This is where the system decides what information to fetch, which tools to call, and how to format the prompt. **Memory is called here.**
3. **Infrastructure Layer:** The "Body." The GPUs, hosting, underlying databases, and foundational models (LLMs) that execute the compute.



Memory

If we zoom into the "Memory" slot of the Context Engineering layer, we find a deeper sub-stack. Memory is not just a database; it is a hierarchy of needs ranging from raw storage to intelligent reasoning.



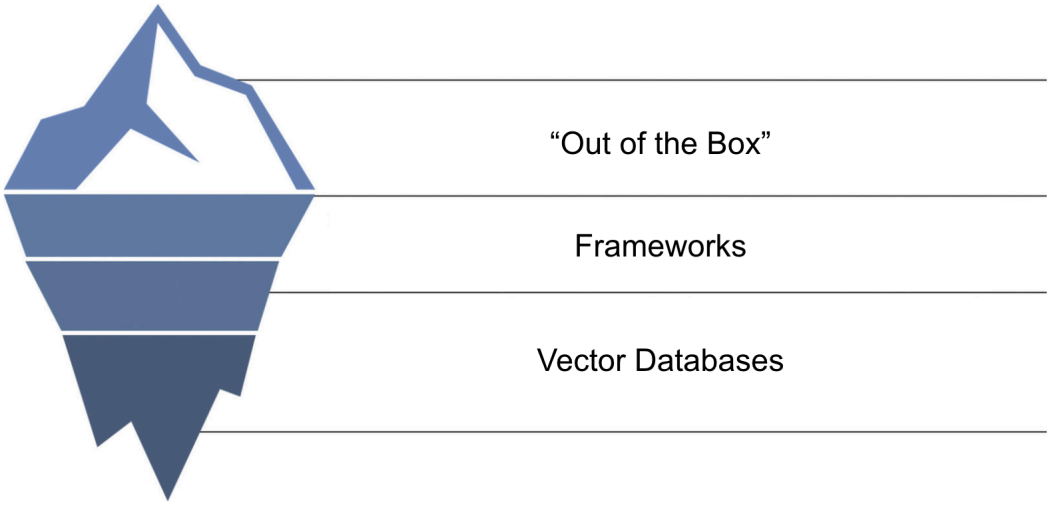
Many memory systems come out of the box with the promise of being able to set it up in “5 lines of code,” including ours. Let’s dive deeper into the different categories of memory companies. Early memory products have diverged in how abstract they are. Ideally, developers want memory to “just work” out of the box for their tasks.

Abstraction Tradeoff

There is an inherent trade-off between simplicity of use and how "opinionated" a memory product is. We can picture this as an iceberg. As you pack more complexity into products that work out of the box, you gain speed but lose control of the underlying building blocks.

In a world where there is one "perfect architecture," we would always opt for the simple-to-use memory product. But that is not the case. A coding agent requires a fundamentally different retrieval logic than a customer support bot. Because each agent has unique requirements, developers must choose where they want to sit on this stack.

Here is the memory stack, organized from the bottom (Raw Control) to the top (High Abstraction).



1. Low Abstraction: Foundational Databases

At the bottom of the stack lies raw storage. These are the vector and graph databases that hold the bits and bytes.

- **Pros:** Total control. You define the ontology, the embedding model, and the retrieval logic perfectly for your specific use case (e.g., highly specific legal discovery vs. casual chat).
- **Cons:** You must build the entire pipeline from scratch. You handle the race conditions, the scaling, and the chunking strategies.
- **Use Case:** Ideal for large tech companies with sophisticated internal engineering teams or highly verticalized agents where the "standard way" of doing memory isn't good enough.

2. Medium Abstraction: Memory Frameworks

Sitting in the middle is a layer of frameworks that abstract away de-duplication, timestamping, managing metadata, and standard functions like add_memory or search_memory. Some are open-source so developers and companies retain some control over tuning the product for their use.

- **Pros:** You don't need to reinvent memory from scratch while maintaining relative control. You can tweak the chunking algorithm or swap the underlying database if needed.
- **Cons:** It is not "plug and play." It still requires a developer to integrate these blocks into their product and make architectural decisions.
- **Use Case:** The sweet spot for most serious developers who need velocity but cannot afford to be locked into a black box.

3. High Abstraction: Developer Tools & "Batteries Included"

At the top of the stack are managed services and developer tools. These solutions promise "Memory in 5 lines of code." Sometimes this is a complex orchestration engine; other times it is simply a hosted tool that abstracts the different tools from the frameworks.

- **Pros:** It “just works.” The provider handles the embedding models, the infrastructure, and the logic. You simply make an API call.
- **Cons:** You lose control. If the provider's chunking strategy cuts off critical context, or if their retrieval logic prioritizes the wrong memories, you often cannot fix it. You are locked into their "opinion" of how memory should work. May not allow self-hosting.
- **Use Case:** Perfect for MVPs, simple chatbots, or applications where memory is a feature, not the core product differentiator. If you can find an abstract, opinionated memory solution that is perfect for your use case, this is ideal.

Conclusion

Memory is not RAG. It is a complex pipeline filtering, storing, and reorganizing experiences for future recall. As memory systems become more sophisticated, we are forced to make very real design decisions across different architectures.

For developers, consider your downstream use case before implementing any product you find online. Start simple and experiment.

In Part 3, we will explore more tangible use cases of memory and evaluate the AI Memory market as a whole. With such a quickly developing space, we are left questioning where exactly value will accrue.

Part 3: Use Cases and Market

See part 1 for the pre-history of AI Memory and part 2 for the sector's technical foundations.

Introduction

We just covered a lot, so let's take a step back. So far, we briefly overviewed the history of "memory" from basic RAG to today's architectures. Then, we dove deeper into the technical weeds of how to build these systems. We will now re-emerge from this depth and review what memory is practically used for today as well as who is actually providing memory products in the market.

We make no attempt to prophesize what chunk of the trillion dollar AI market memory will take home. What we can be sure of is that there will be millions-billions of agents and AI applications by 2030, each requiring memory.

While we kick the can down the road on market sizes, we connect the upstream use cases to downstream providers and ask where exactly the majority of value will accrue in this space.

Section 1: Use Cases

With the exception of some edge cases, the majority of applications and agents requiring memory fall under 3 categories:

1. B2C – Life Self: the individual is the center. Memory creates personalization and persistence.
2. B2B2C – Mediated Self: the consumer is the center, but used to provide better core services.
3. B2B – The organization, system, or process is the center. Often process automation.

It would not be possible to review every use case and implementation of memory, so we will give a high level review and an example for each, while providing market maps here to let your imagination run for the breadth of use cases.

B2C

In consumer applications, the center of gravity is the individual. In these applications, memory may be the product itself or memory may be core to the value proposition offered. Users demand an application that knows and *understands* them and can recall prior conversations. In order to build a companion, coach, "second brain," or a personal assistant, there must be continuity so you aren't starting from scratch each conversation.

- **Companions & Therapists:** AI that builds a psychological profile to offer emotional support (e.g., remembering trauma or relationship dynamics).
- **Coaches (Health/Finance):** Agents that track longitudinal progress (e.g., "Your spending on takeout has dropped 15% since our talk last Tuesday").
- **"Second Brains":** Tools that ingest your entire digital life (emails, notes, reading history) to act as a searchable extension of your mind.
- **Personal Shoppers:** Agents that remember your size, style evolution, and the fact that you hate polyester.
- **Tutors:** Education bots that track your learning curve, remembering exactly which concepts you struggled with three weeks ago.

Example: Kin AI Kin is a personal AI designed to be a supportive peer. Unlike standard chatbots, Kin used a memory architecture called Semantic Spacetime.

- **Priority:** Most RAG systems are "timeless" and just find matching keywords. Kin treats *time* and *causality* as core to memory. If you tell Kin you are nervous about a presentation on Friday, and you log back in on Saturday, it proactively asks, *"How did it go?"*
- **The Architecture:** It doesn't just store the text; it stores the event on a temporal graph, linking the "Date" event to your "Anxiety" entity. This allows the AI to "wake up" with an understanding of where you are in your life's narrative, not just what you last typed.

B2B2C

When memory is built into an existing business, context is used to improve the value of products or services. The culture of different companies and industries are subtly different and the value must be clearly proposed for how it will improve their business (i.e. conversions for e-commerce). This is where memory, data density, and clear ROI line up most naturally.

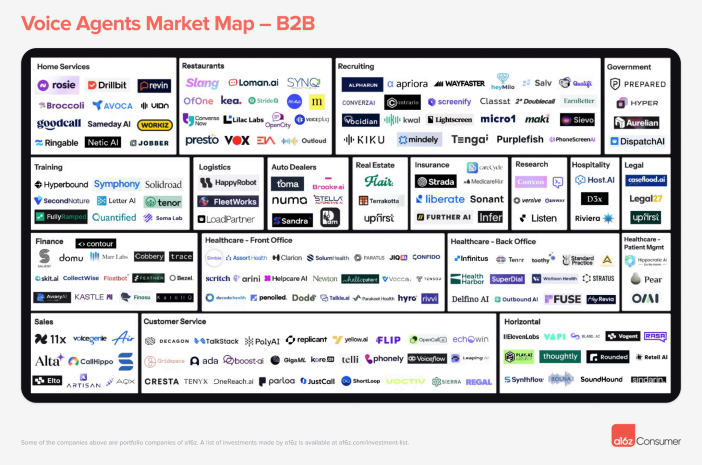
Many companies will use memory as a logical extension of their existing data collection and personalization practices. Others will reimagine them entirely. X has notably begun using LLMs in their recommendation systems, which was seen as heretical to literature published by Twitter's previous RecSys teams.

Common Use Cases:

- **Chat Bots/Voice Agents:** The new frontier of support. Voice agents require ultra-low latency (sub-300ms) to feel natural. Memory must be retrieved faster than a human can blink.
- **Dynamic UI/UX:** Interfaces that generate themselves on the fly based on user history. Instead of a static website, the AI renders a custom dashboard for *you*.
- **Transactional Support:** Agents that instantly recall order history and return policies to close tickets without human intervention.

Example: The Enterprise Voice Agent Investors are pouring capital into Voice AI (e.g., Bland, Vapi, Retell). The constraint here is physics.

- **Priority: Latency.** A natural conversation has a "turn-taking" gap of ~500ms. If the AI takes 2 seconds to remember who you are, the illusion breaks.
- **The Architecture:** These systems *cannot* use complex, multi-hop reasoning during the call. They rely on caching and background processing. Before the call connects, the memory system pre-fetches the user's profile (name, last order, sentiment) and injects it into the immediate context window. The memory architecture is optimized for read-speed (milliseconds) rather than depth. It trades nuance for velocity.



B2B

In the enterprise, AI agents are always on, automating ever-larger chunks of the "services" work that powers the global economy. Memory here is about coordination and continuity of work. It is the difference between an intern who needs to be retrained every morning and an expert employee who knows the ins and outs of the business.

Common Use Cases:

- **Coding Agents:** Autonomous developers that navigate complex repositories, remembering dependency trees and architectural decisions.
- **“Swarm” workflows:** Multiple agents coordinating across sales, ops, or support. One agent’s action becomes another’s context.
- **Back-office automation:** Invoices, reconciliations, approvals, escalations—all of which have state that extends beyond a single request.

Example: The "Always-On" Service Agent Consider an AI responsible for reconciling invoices.

- **Priority:** Precision is non-negotiable. A hallucination means financial fraud or audit failure. Logs are required for auditing.
- **The Architecture:** These systems rarely use simple vector search. They rely on strict ontologies. The memory must track the *state* of a task (e.g., "Waiting for approval from Bob"). It compresses the complexity of 10,000 emails into a single "State" object that the agent can act on reliably.

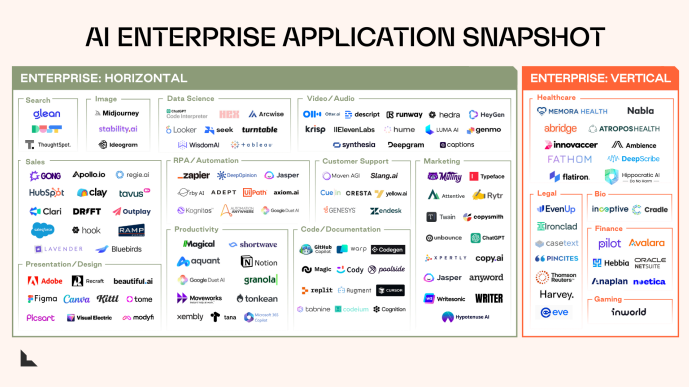
Hardware / Networks / Miscellaneous

Hardware is becoming a hotbed for new products infused with AI. Devices that see what you see and hear what you hear, pave the way for a future where we can build “proactive AI” and devices that act as support for the human brain’s limitations.

New social networks will be built in which context is the memory link across people, ideas, and organizations. Boardy is an interesting “superconnector” application that connects individuals with similar interests proactively, as you would imagine LinkedIn would.

Example: Limitless (Acquired by Meta in December 2025) The recent acquisition of Limitless (formerly Rewind) by Meta represents that “hand off” of human memory to machines. Limitless built a "pendant" that records and transcribes real-world audio.

- **Priority:** Quantity matters. Memory here is a firehose: the core challenge is not retrieval, it’s deciding what to keep, how to compress it, and how not to leak it.
- **The Architecture:** This requires massive Audio-First Ingestion and privacy filtering at the edge. The system must process 16 hours of audio a day, identify speakers, and index it all without sending sensitive data to the cloud unnecessarily.



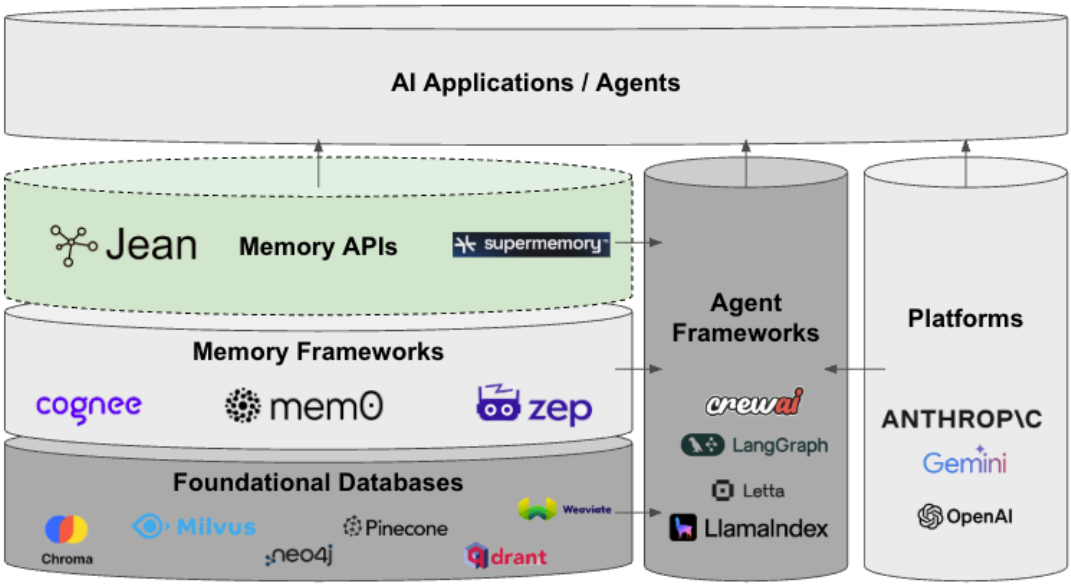
Section 2: Market Providers

We have established there will be many different use cases of memory. The examples provide a glimpse into how the value each application aims to provide demands different memory architectures. We can think of this as the demand side of memory.

We will now overview the supply side. For developers today, what options are available for building performant AI systems?

We generally categorize market providers of memory by:

1. Platforms
2. Agent Frameworks
3. The “Memory Stack”
    a. Layer 1: Foundational Databases
    b. Layer 2: Memory Frameworks
    c. Layer 3: Opinionated Memory Products



jonathan@jeanmemory.com

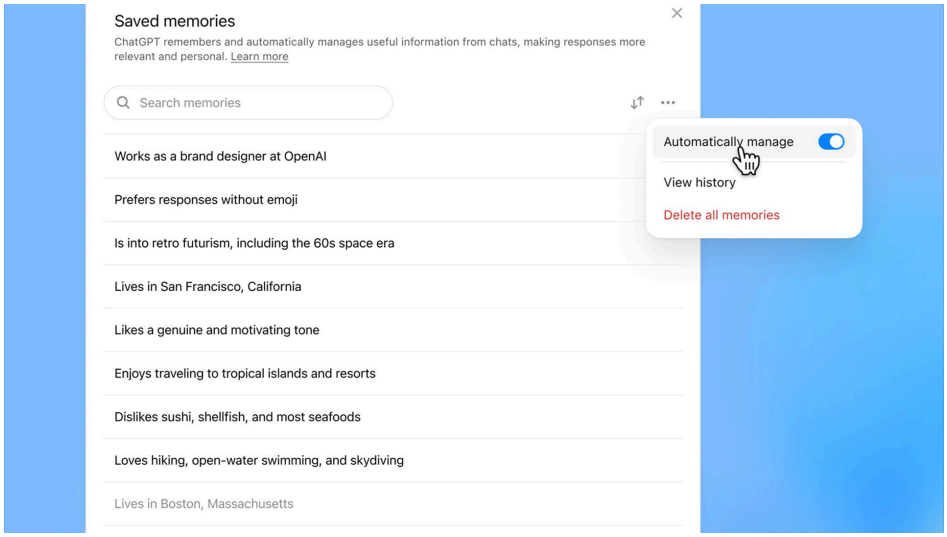
1. Platforms

The platforms who provide LLMs have the natural distribution to provide memory services. If we have learned anything from the development of these companies, it is that incumbents shouldn't be ignored, their capacity to innovate and move fast is unlike previous cycles. The "Big Three" have each taken a different philosophical approach to how they offer memory to developers.

OpenAI

OpenAI's bet is mostly consumer-first and they are clearly aiming to be the next big "walled garden" of consumer data. If you have used their product, you likely have an intuition for how it works. ChatGPT mostly uses semantic memory to store:

- Explicit facts you tell it ("I'm a lawyer in SF")
- Patterns across your chats (topics, projects, preferences)
- Recent conversation history and usage metadata.



For developers, they're slowly pushing toward a more stateful API, but it's still OpenAI's memory, inside OpenAI's world. This is a general purpose memory with extreme vendor lock-in.

Anthropic

Anthropic's Memory SDK, we believe, is a glimpse into the future of what memory systems will look like. Notably, developers have the control to store memory on internal systems. Anthropic is not trying to own this layer.

Here, memory is designed to be relatively task agnostic and is a move towards *unified memory.* The operations of deciding what to read and write are given to the LLM itself through the model context protocol. The memory operations are agentic and not as simple a retrieval product as ChatGPT's system.

This memory is opinionated, but it is still relatively simple and can not scale to large context tasks. Its generality, while a feature for many simple tasks, makes it unable to perform for many tasks that require more complex architectures.

```
XML
IMPORTANT: ALWAYS VIEW YOUR MEMORY DIRECTORY BEFORE DOING ANYTHING ELSE.
MEMORY PROTOCOL:
1. Use the `view` command of your `memory` tool to check for earlier progress.
2. ... (work on the task) ..
- As you make progress, record status / progress / thoughts etc in your memory.
ASSUME INTERRUPTION: Your context window might be reset at any moment, so you
risk losing any progress that is not recorded in your memory directory.
```

Google Gemini: The "Managed Platform" Model (Vertex AI)

Google's bet is managed enterprise memory. Gemini personalizes using your Google data, and in the cloud they offer Vertex AI Memory Bank, a service that extracts and stores "memories" from agent conversations and serves them back via API.

It's convenient if you're all-in on Google Cloud, but the memory lives inside Google's schema and stack, not as a portable asset you can freely shape and move.

2. Agent Frameworks

For most agent frameworks, memory is not their core value proposition, and you can bring your own external memory stores. These frameworks are built to provide scaffolding for easier development of agent orchestration and development. CrewAI, LangChain, and LlamaIndex compete here.

The exception where memory is core to the agent framework is Letta, a Berkeley spinout initially focused on memory. Letta later shifted gears to compete against frameworks like LangChain. They provide "memory blocks" and other opinionated memory architectures including sleep-time compute. The obvious limitation here is that if you aren't building with Letta, their internal memory isn't very useful.

3. The Memory Stack

The memory stack offers the basic building blocks of databases all the way up to abstract, opinionated memory solutions.

A) Foundational Databases

At the most basic level, you can build memory from the ground up using the building blocks of existing vector and graph databases.

Vector Databases

- Pinecone, Weaviate, Qdrant, Chroma, Milvus
- Optimized for similarity search at scale
- You own the chunking, embedding model selection, and retrieval logic

Graph Databases

- Neo4j, Graphiti (Zep)
- Enforce entity-relationship structure
- Powerful for bounded domains; brittle for open-ended conversation

B) Memory Frameworks

The middle layer abstracts away the plumbing: de-duplication, timestamping, metadata management, and standard operations like `add_memory` and `search_memory`. These are the building blocks that let you assemble a memory system without starting from zero.

Memory Frameworks

- **Mem0**: Lightweight memory layer with built-in entity extraction and conflict resolution. Open-source core with managed offering.
- **Zep**: Long-term memory for AI assistants. Focuses on graph-based relationships and providing temporal structure to memory.

C) Opinionated Memory Products

Opinionated memory works out of the box. Claude's memory and the platforms would also fall into this category. While at the top of the stack, they do not necessarily need to be abstractions of vector or graph databases. With opinionated products, you lose the control of building your own stack from the ground up but you get the ease of implementation and a product that has been stress-tested widely.

Supermemory was one of the first companies that stumbled into this category after initially coming up as an indie product to save down bookmarks, they eventually offered a wrapper around a RAG database as a developer tool to abstract away much of the complexity of implementation.

Jean also lives in this category. We initially gained traction building intelligent memory across applications. Now, we are building more intelligent, next-generation memory for high-value use cases.

Section 3: Developer Choices

Ultimately, the decision of which architecture to implement sits with the developer. They face constraints around time, budget, and how core memory is to their application.

The Naive View

In an ideal world, there's a single product at the top of the stack that "just works" for every use case. You call an API, memory happens, and you focus on your application.

This is not the world we live in.

As we've shown throughout this series, a voice agent requiring sub-300ms latency has fundamentally different memory needs than a coding agent tracking dependency trees. A companion app building psychological profiles over months requires different architecture than a customer service bot recalling order history. The use case dictates the architecture. General-purpose memory is a compromise.

The Real Tradeoffs

Build from scratch (Foundational Databases)

- *When it makes sense*: Your use case is sufficiently unique that no existing solution fits. You have engineering capacity to build and maintain the full pipeline. Memory is your core product differentiator.
- *The cost*: You're solving chunking strategies, embedding selection, retrieval logic, conflict resolution, temporal handling, and scaling—problems that have already been solved elsewhere. Most teams underestimate the ongoing maintenance burden.
- *The trajectory*: The database layer is commoditizing. Pinecone, Weaviate, Qdrant, Chroma are converging on similar feature sets. Building here means competing on implementation details that will matter less over time.

Buy from platforms (OpenAI, Anthropic, Google)

- *When it makes sense*: Memory is a feature, not your product. You're already locked into a platform ecosystem. Speed to market matters more than architectural control.
- *The cost*: Vendor lock-in. Opacity—you can't see or modify how memory works under the hood. Architectural mismatch—platform memory is general-purpose by design, optimized for the median use case rather than yours.
- *The trajectory*: Platforms will continue bundling memory as a retention mechanism. The memory itself becomes a moat for them, not for you. Your user data lives in their systems.

Assemble from the stack (Frameworks + APIs)

- *When it makes sense*: You need more control than platforms offer but don't want to build from zero. Your use case is complex enough that general-purpose memory underperforms but not so unique that nothing in the market applies.
- *The cost*: Integration work. You're making architectural bets on which layers to own and which to outsource. The frameworks and APIs you choose shape your system's ceiling.
- *The trajectory*: This is where most serious developers land. The question becomes: which layers do you assemble, and from whom?

The Horizontal vs. Vertical Dimension

The stack we've outlined is one axis: abstraction level. But there's a second axis that matters just as much: **specificity**.

Horizontal memory is general-purpose. It works reasonably well across many use cases but isn't optimized for any particular one. The frameworks and databases live here. The value proposition is breadth and ease of integration.

Vertical memory is domain-specific. It's designed for a particular class of problems: voice agents, coding assistants, therapeutic companions, enterprise automation. The value proposition is performance on the specific task.

The honest reality is that the memory space is still early. Vertical memory solutions don’t really exist and products claim generality because they haven't discovered their natural domain. Over time, we expect specialization. Just as databases fragmented into relational, document, graph, time-series, and vector stores for different workloads, memory will fragment into architectures optimized for different use cases.

Where Value Accrues

If databases commoditize downward and platforms consolidate upward, where does value accrue in the memory stack?

We see three defensible positions:

1. **Platform lock-in**: OpenAI, Anthropic, and Google will capture value through ecosystem control. If you're building on ChatGPT, you'll use ChatGPT memory because it's frictionless. This is a distribution advantage, not a technical one.
2. **Vertical depth**: Products that go deep on a specific use case—understanding the unique requirements of voice latency, or coding context, or therapeutic continuity—can build moats through domain expertise. Horizontal players can't easily replicate this without sacrificing generality.
3. **Infrastructure primitives**: At the framework layer, products that become the "standard" way to handle memory operations, can capture value through developer adoption and ecosystem effects.

The middle ground, general-purpose memory APIs that aren't deep enough to be vertical and aren't standard enough to be infrastructure, is the most contested and most questionable position.

Conclusion

Memory is not a solved problem. The space is fragmented because use cases are fragmented, and we expect this to accelerate. The companies that win will be those that provide high-value, performant memory that is not commoditized.

For developers: start with your use case, not your database. Understand what "right context, right time" means for your specific application. Then work backward to the product that delivers it.

Part 4: Future Outlook

Introduction

We have now reviewed the history of how AI Memory formed (part 1), a technical overview for developers building memory systems (part 2), and a review of the current state of the market (part 3).

We will now synthesize all of this information into a future outlook. We will briefly review each section we have written and extract some core themes, before weaving them into 4 forward-looking theses.

Mainly, history will be rewritten. RAG was a band-aid and will be replaced with agentic, active memory over time. Value will flow towards platforms and vertical, high-value memory products, and the space is growing unpredictably fast with many emerging use cases and opportunities.

Part 1: Pre-History

In our first section, we looked into the past to understand how we arrived at today's idea of AI memory. Mostly, AI Memory was born out of the simple problem that LLMs are “stateless” and do not remember. The first simple fix was to just throw the entire conversation into every new prompt, forming a “short-term memory” of what has occurred in the current chat.

The first appearance of long-term memory came in the form RAG and web search, where one could access context outside of the chat window. These were still primitive at the time.

RAG was retrofitted for memory, but it was clear this was only because function calling was unreliable and expensive. Structured output and Model Context Protocol (MCP) had not arrived yet. Over time, prompt engineering evolved into context engineering. It became clear memory was now a core part of every AI Agent, yet we were still using the unintelligent memory systems of the past.

Part 2: Technical Foundations

In the second section, we evaluated memory from first principles. What is memory and what is its purpose? We posit that memory is not destined to be passive (as it currently is), it is actually supposed to be active and propose the E → S → R pipeline, where memory is an intelligent process of its own.

We claim that we wouldn’t segment memories at all in a perfect world, and that partitioning was the “*original sin*” of AI Memory. This ties into the narrative that RAG was retrofitted for memory.

It was the obvious solution. But as we approach a new age, memory should be agentic by default.

We reviewed each architecture for memory and their strengths / weaknesses, including anecdotes from building these systems. Lastly, we propose the idea of the memory stack, where foundational databases lie at the bottom and are abstracted away into more opinionated memory products.

Part 3: Market Dynamics

In the third section, we take a step away from the code to touch grass and look at how people are actually using memory today. We ask where the value is accruing in the market. We acknowledge the present use cases, how developers decide whether to build or buy, and the gaps in the market.

It was clear in section 2 that different architectures have different strengths. In section 3, it also became clear that different use cases required drastically different architectures. This leads us to reject any “one-size-fits-all” solution, which opens the door to rich market fragmentation and competition.

We conclude that foundational storage has been commoditized. Value is accruing from AI applications downwards. So in order to understand where value accrues, we must look to developer activity.

The logical memory choice for developers is to inherit the memory of the LLM provider like OpenAI or Anthropic, especially when memory is not core to the value of the application. However, these general solutions are designed to be general and do not fit for every different use case.

In the event a company has a unique use case, they may decide to build their own memory, build on top of memory frameworks, or purchase a vertical memory solution. The largest question we have today is where, among these three choices, will developers lean the hardest? We attempt to answer this question in thesis 2 of this write-up.

Claude's agentic memory and Letta's storage blocks point the direction: memory as an active, agentic process, not passive retrieval. Mem0 won developer mindshare through simplicity and enterprise focus, creating a de facto standard even as questions about monetization linger. Zep pushed downstream into context engineering, attempting to carve out their own niche after getting squeezed from the middle. Supermemory offered simplicity but lacked vertical focus to capture premium value. Each teaches something about where value does and doesn't accumulate. Collectively, they reveal an industry fragmenting and becoming more opinionated.

Thesis 1: The Future of Memory is Agentic

Since segmentation is the “original sin” of memory and RAG was simply retrofitted for this use case, we posit that RAG is being demoted in the memory stack. For low context applications, we will start by building agentic memory. We will add memory blocks and RAG only as context size increases above manageable thresholds. We can learn the most from the design philosophy of Anthropic’s Memory SDK and Letta in this regard.

Many enterprises may still have narrowly defined tasks. Graph and GraphRAG will still perform well in domains where ontologies can be effectively harnessed.

Thesis 2: Waterfall of Value

We anticipate that the value accruing to memory will flow to those with obvious distribution advantages and only flow downwards to those who solve high-value problems that incumbents cannot.

Platforms

Platforms such as OpenAI, Anthropic, and Gemini have distribution advantages, and will capture the majority of simple value. We expect that general platforms will be the rising tide whose memory will work for most general cases. Their generality, however, will leave their products unsuited for use cases that require specific memory solutions. After building our own products with memory, we saw the limitations of the basic memory solutions.

Memory Frameworks (Squeeze)

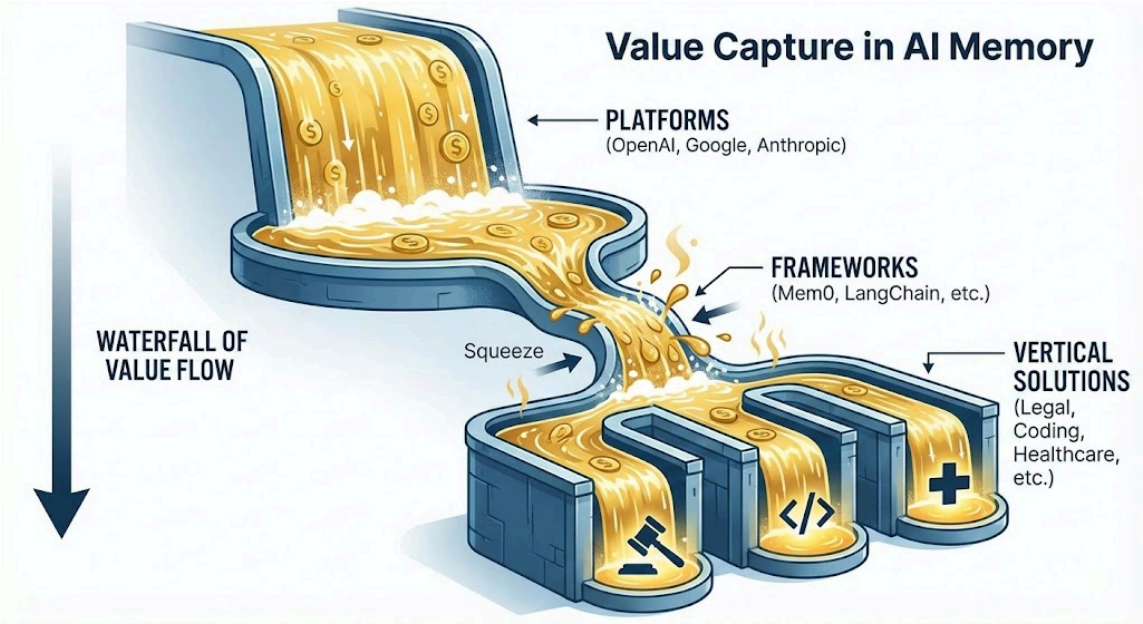
Memory frameworks have won developer mindshare first, abstracting some of the more high-level decisions in building memory systems, but they are getting squeezed by applications who choose to build themselves and narrower, high-value vertical memory solutions that either build on top of their open-sourced components or build separately.

Mem0 was the breakaway leader in this category, but long-term profitability concerns remain. They have also built a product that assumes RAG becomes the core primitive, rather than an agentic memory. As we have reviewed extensively, RAG isn’t going anywhere, but its core status is called into question.

Vertical Memory

For now, vertical memory as a category is nascent but growing fast. Over time, we expect the vertical category to expand with demand and increasingly divergent memory requirements. Many vertical solutions may build on Mem0 and other open-sourced frameworks and take their lunch.

Many vertical memory solutions will naturally slot into various domains (legal, finance, healthcare) as well as different use cases (enterprise agents, personalization, voice agents).



Thesis 3: We Are Early

Memory is not just recall. After you have built large memory banks, we can then ask important questions about how we can use those stores. We are building the next generation of persistent, personalized products, long-running intelligent agents, and more. Increasing variety and sophistication of AI Applications and Agents lead to an increase in demand for new memory products. Indeed, the introduction of new memory products also paves the way for new applications. New interfaces will call you by name and generate UI on demand for your unique personality and interests at that moment.

Thesis 4: The Prize is Cross-Application Context

We started down this road by recognizing that the biggest prize in the age of AI was going to be who would own the user context layer. Our context is fragmented across siloed applications, each seeing only a narrow slice of our holistic identities. This has been a tough nut and already many companies have lived and died attempting to crack it.

Memory is the logical form factor for this cross-application context. Judging from the actions of OpenAI, this is likely where they see their largest opportunity and are gearing up to capture it.

Users want this in principle but are also privacy conscious. As memory solutions fragment increasingly, the likelihood that OpenAI becomes the ubiquitous winner falls.

The winner may not be a platform at all. It may be a neutral identity layer that connects fragmented memory systems—letting users port their context across applications without any single provider owning them. This is the Plaid play for AI: not building memory, but connecting it.

Conclusion

While AI Memory as an industry is beginning to take shape, it is young. There is no telling how the future can unfold. But we have made some projections after evaluating the space holistically from the ground up.

1. Memory will become increasingly agentic over time.
2. The majority of value in memory will accrue to the big platforms and high-value opinionated memory products.
3. We are seeing an explosion of use cases on top of the foundation of memory. We will continue to see new use cases requiring new memory solutions.
4. Whoever wins cross-application context / user identity will become one of the most valuable companies in the world.

Thanks for sticking with us for our comprehensive review of AI Memory. As always, if you have any questions or are interested in building AI Memory, reach out to Jean.