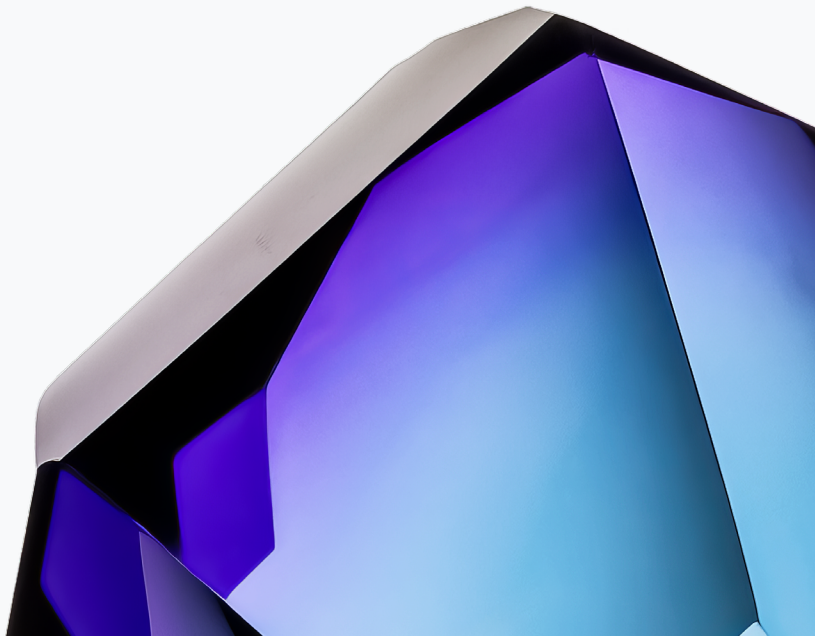
Context Engineering: Sessions, Memory

Authors: Kimberly Milam and Antonio Gulli



Google

Acknowledgements

Content contributors

Kaitlin Ardiff

Shangjie Chen

Yanfei Chen

Derek Egan

Hangfei Lin

Ivan Nardini

Anant Nawalgaria

Kanchana Patlolla

Huang Xia

Jun Yan

Bo Yang

Michael Zimmermann

Curators and editors

Anant Nawalgaria

Kanchana Patlolla

Designer

Michael Lanning

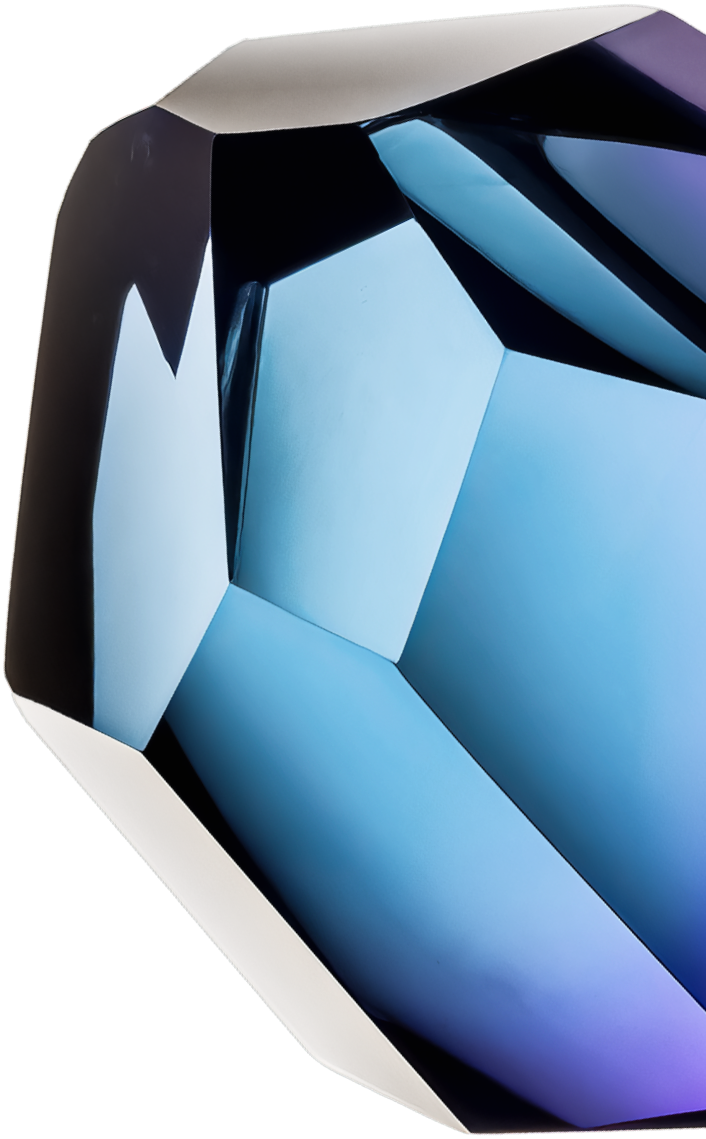


Table of contents

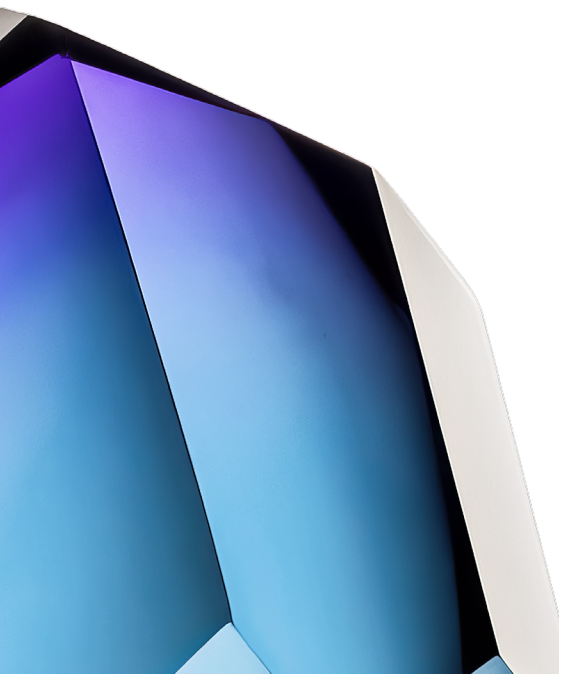


Table of contents

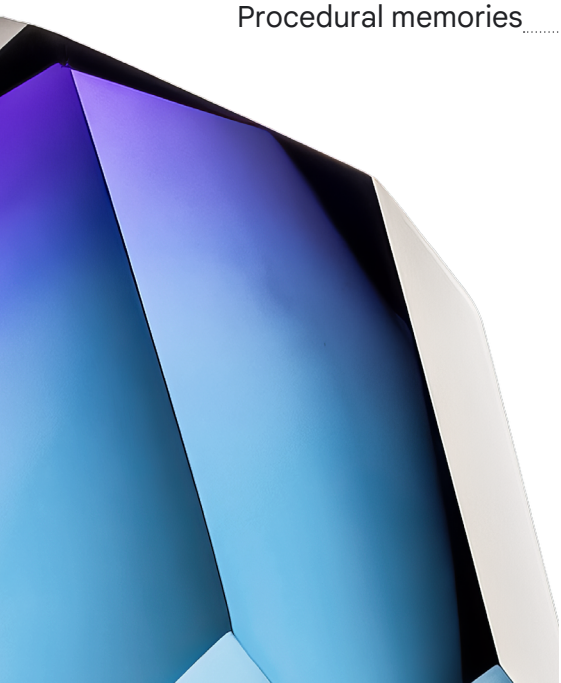
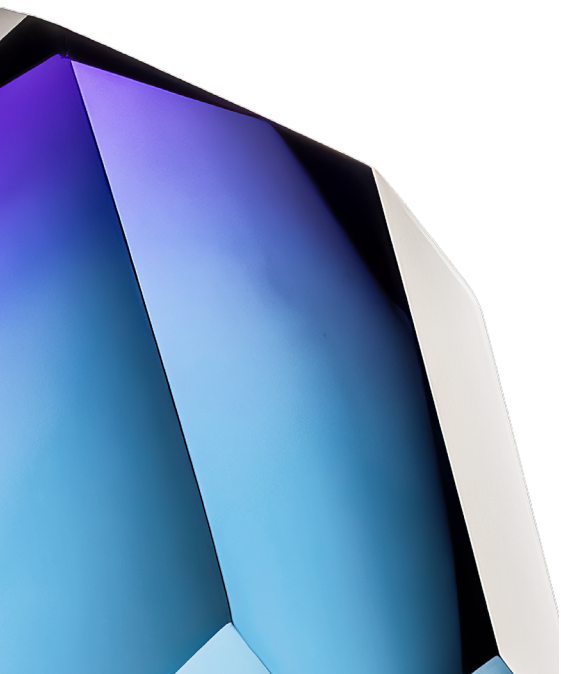


Table of contents



Stateful and personal AI begins with Context Engineering.

Introduction

This whitepaper explores the critical role of Sessions and Memory in building stateful, intelligent LLM agents to empower developers to create more powerful, personalized, and persistent AI experiences. To enable Large Language Models (LLMs) to remember, learn, and personalize interactions, developers must dynamically assemble and manage information within their context window—a process known as Context Engineering.

These core concepts are summarized in the whitepaper below:

- **Context Engineering:** The process of dynamically assembling and managing information within an LLM's context window to enable stateful, intelligent agents.
- **Sessions:** The container for an entire conversation with an agent, holding the chronological history of the dialogue and the agent's working memory.

- **Memory:** The mechanism for long-term persistence, capturing and consolidating key information across multiple sessions to provide a continuous and personalized experience for LLM agents.

Context Engineering

LLMs are inherently stateless. Outside of their training data, their reasoning and awareness are confined to the information provided within the "context window" of a single API call. This presents a fundamental problem, as AI agents must be equipped with operating instructions identifying what actions can be taken, the evidential and factual data to reason over, and the immediate conversational information that defines the current task. To build stateful, intelligent agents that can remember, learn, and personalize interactions, developers must construct this context for every turn of a conversation. This dynamic assembly and management of information for an LLM is known as Context Engineering.

Context Engineering represents an evolution from traditional **Prompt Engineering**. Prompt engineering focuses on crafting optimal, often static, system instructions. Conversely, **Context Engineering** addresses the entire payload, dynamically constructing a state-aware prompt based on the user, conversation history, and external data. It involves strategically selecting, summarizing, and injecting different types of information to maximize relevance while minimizing noise. External systems—such as RAG databases, session stores, and memory managers—manage much of this context. The agent framework must orchestrate these systems to retrieve and assemble context into the final prompt.

Think of Context Engineering as the *mise en* place for an agent—the crucial step where a chef gathers and prepares all their ingredients before cooking. If you only give a chef the recipe (the prompt), they might produce an okay meal with whatever random ingredients they have. However, if you first ensure they have all the right, high-quality ingredients, specialized

tools, and a clear understanding of the presentation style, they can reliably produce an excellent, customized result. The goal of context engineering is to ensure the model has no more and no less than the most relevant information to complete its task.

Context Engineering governs the assembly of a complex payload that can include a variety of components:

- **Context to guide reasoning** defines the agent's fundamental reasoning patterns and available actions, dictating its behavior:
  - **System Instructions:** High-level directives defining the agent's persona, capabilities, and constraints.
  - **Tool Definitions:** Schemas for APIs or functions the agent can use to interact with the outside world.
  - **Few-Shot Examples:** Curated examples that guide the model's reasoning process via in-context learning.
- **Evidential & Factual Data** is the substantive data the agent reasons over, including pre-existing knowledge and dynamically retrieved information for the specific task; it serves as the 'evidence' for the agent's response:
  - **Long-Term Memory:** Persisted knowledge about the user or topic, gathered across multiple sessions.
  - **External Knowledge:** Information retrieved from databases or documents, often using Retrieval-Augmented Generation (RAG)[1].
  - **Tool Outputs:** The data or results returned by a tool.
  - **Sub-Agent Outputs:** The conclusions or results returned by specialized agents that have been delegated a specific sub-task.

    - **Artifacts:** Non-textual data (e.g., files, images) associated with the user or session.
- **Immediate conversational information** grounds the agent in the current interaction, defining the immediate task:
    - **Conversation History:** The turn-by-turn record of the current interaction.
    - **State / Scratchpad:** Temporary, in-progress information or calculations the agent uses for its immediate reasoning process.
    - **User's Prompt:** The immediate query to be addressed.

The dynamic construction of context is critical. Memories, for instance, are not static; they must be selectively retrieved and updated as the user interacts with the agent or new data is ingested. Additionally, effective reasoning often relies on in-context learning[2] (a process where the LLM learns how to perform tasks from demonstrations in the prompt). In-context learning can be more effective when the agent uses few-shot examples that are releva nt to the current task, rather than relying on hardcoded ones. Similarly, external knowledge is retrieved by RAG tools based on the user's immediate query.

One of the most critical challenges in building a context-aware agent is managing an ever-growing conversation history. In theory, models with large context windows can handle extensive transcripts; in practice, as the context grows, cost and latency increase. Additionally, models can suffer from "**context rot**," a phenomenon where their ability to pay attention to critical information diminishes as context grows. Context Engineering directly addresses this by employing strategies to dynamically mutate the history—such as summarization, selective pruning, or other compaction techniques—to preserve vital information while managing the overall token count, ultimately leading to more robust and personalized AI experiences.

This practice manifests as a continuous cycle within the agent's operational loop for each turn of a conversation:

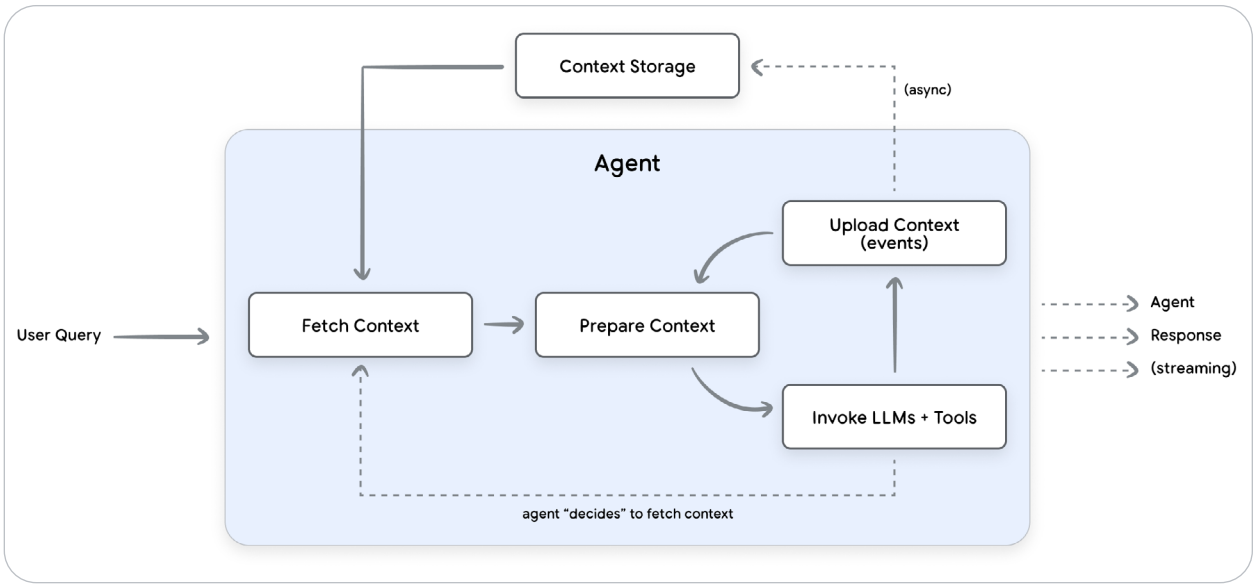


Figure 1. Flow of context management for agents

1. **Fetch Context:** The agent begins by retrieving context—such as user memories, RAG documents, and recent conversation events. For dynamic context retrieval, the agent will use the user query and other metadata to identify what information to retrieve.
2. **Prepare Context:** The agent framework dynamically constructs the full prompt for the LLM call. Although individual API calls may be asynchronous, preparing the context is a blocking, "hot-path" process. The agent cannot proceed until the context is ready.
3. **Invoke LLM and Tools:** The agent iteratively calls the LLM and any necessary tools until a final response for the user is generated. Tool and model output is appended to the context.

4. **Upload Context:** New information gathered during the turn is uploaded to persistent storage. This is often a "background" process, allowing the agent to complete execution while memory consolidation or other post-processing occurs asynchronously.

At the heart of this lifecycle are two fundamental components: **sessions** and **memory**. A **session** manages the turn-by-turn state of a single conversation. **Memory**, in contrast, provides the mechanism for long-term persistence, capturing and consolidating key information across multiple sessions.

You can think of a session as the workbench or desk you're using for a specific project. While you're working, it's covered in all the necessary tools, notes, and reference materials. Everything is immediately accessible but also temporary and specific to the task at hand. Once the project is finished, you don't just shove the entire messy desk into storage. Instead, you begin the process of creating memory, which is like an organized filing cabinet. You review the materials on the desk, discard the rough drafts and redundant notes, and file away only the most critical, finalized documents into labeled folders. This ensures the filing cabinet remains a clean, reliable, and efficient source of truth for all future projects, without being cluttered by the transient chaos of the workbench. This analogy directly mirrors how an effective agent operates: the session serves as the temporary workbench for a single conversation, while the agent's memory is the meticulously organized filing cabinet, allowing it to recall key information during future interactions.

Building on this high-level overview of context engineering, we can now explore two core components: sessions and memory, beginning with sessions.

Sessions

A foundational element of Context Engineering is the session, which encapsulates the immediate dialogue history and working memory for a single, continuous conversation. Each session is a self-contained record that is tied to a specific user. The session allows the agent to maintain context and provide coherent responses within the bounds of a single conversation. A user can have multiple sessions, but each one functions as a distinct, disconnected log of a specific interaction. Every session contains two key components: the chronological history (**events**) and the agent's working memory (**state**).

Events are the building blocks of the conversation. Common types of events include: **user input** (a message from the user (text, audio, image, etc.), **agent response** (the agent's reply to the user), **tool call** (the agent’s decision to use an external tool or API), or **tool output** (the data returned from a tool call, which the agent uses to continue its reasoning).

Beyond the chat history, a Session often includes a **state**—a structured "working memory" or scratchpad. This holds temporary, structured data relevant to the current conversation, like what items are in a shopping cart.

As the conversation progresses, the agent will append additional events to the session. Additionally, it may mutate the state based on logic in the agent.

The structure of the events is analogous to the list of `Content` objects passed to the Gemini API, where each item with a `role` and `parts` represents one turn—or one Event—in the conversation.

Python

```
contents = [
    {
        "role": "user",
        "parts": [ {"text": "What is the capital of France?"} ]
    }, {
        "role": "model",
        "parts": [ {"text": "The capital of France is Paris."} ]
    }
]
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents=contents
)
```

Snippet 1: Example multi-turn call to Gemini

A production agent's execution environment is typically stateless, meaning it retains no information after a request is completed. Consequently, its conversation history must be saved to persistent storage to maintain a continuous user experience. While in-memory storage is suitable for development, production applications should leverage robust databases to reliably store and manage sessions. For example, you can store conversation history in managed solutions like Agent Engine Sessions[3].

Variance across frameworks and models

While the core ideas are similar, different agent frameworks implement sessions, events, and state in distinct ways. Agent frameworks are responsible for maintaining the conversation history and state for LLMs, building LLM requests using this context, and parsing and storing the LLM response.

Agent frameworks act as a universal translator between your code and a LLM. While you, the developer, work with the framework's consistent, internal data structures for each conversational turn, the framework handles the critical task of converting those structures into the precise format the LLM requires. This abstraction is powerful because it decouples your agent's logic from the specific LLM you're using, preventing vendor lock-in.

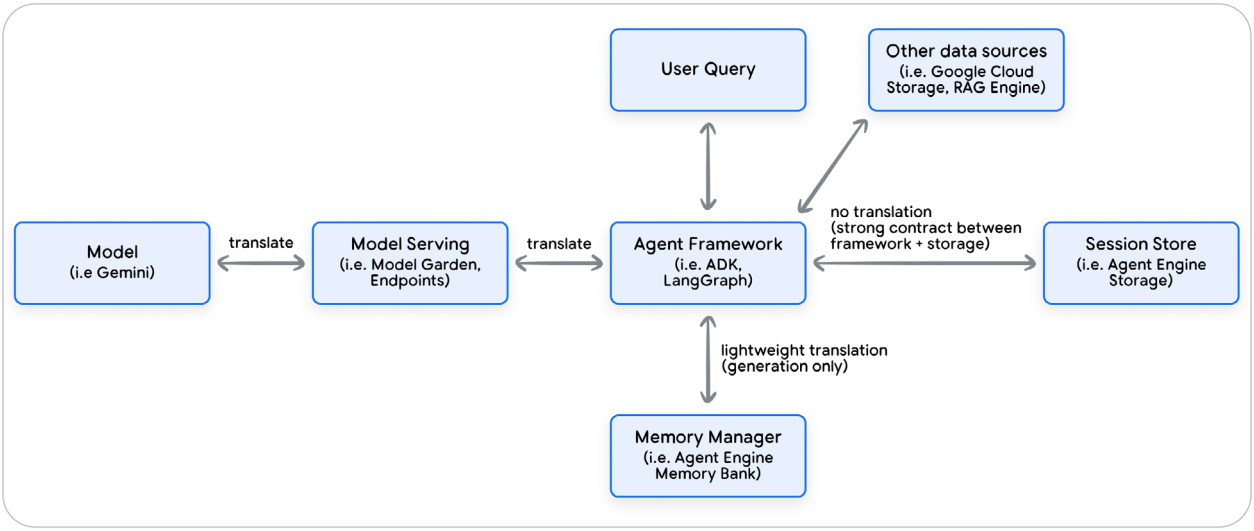


Figure 2: Flow of context management for agents

Ultimately, the goal is to produce a "request" that the LLM can understand. For Google's Gemini models, this is a `List[Content]`. Each Content object is a simple dictionary-like structure containing two keys: `role` which defines who is speaking ("user" or "model") and `parts` which defines the actual content of the message (text, images, tool calls, etc.).

The framework automatically handles mapping the data from its internal object (e.g., an ADK `Event`) to the corresponding role and parts in the `Content` object before making the API call. In essence, the framework provides a stable, internal API for the developer, while managing the complex and varied external APIs of the different LLMs behind the scenes.

ADK uses an explicit `Session` object that contains a list of `Event` objects and a separate state object. The Session is like a filing cabinet, with one folder for the conversation history (events) and another for working memory (state).

LangGraph doesn't have a formal "session" object. Instead, the state is the session. This all-encompassing state object holds the conversation history (as a list of `Message` objects) and all other working data. Unlike the append-only log of a traditional session, LangGraph's state is mutable. It can be transformed, and strategies like history compaction can alter the record. This is useful for managing long conversations and token limits.

Sessions for multi-agent systems

In a multi-agent system, multiple agents collaborate. Each agent focuses on a smaller, specialized task. For these agents to work together effectively, they must share information. As shown in the diagram below, the system's architecture defines the communication patterns they use to share information. A central component of this architecture is how the system handles session history—the persistent log of all interactions.

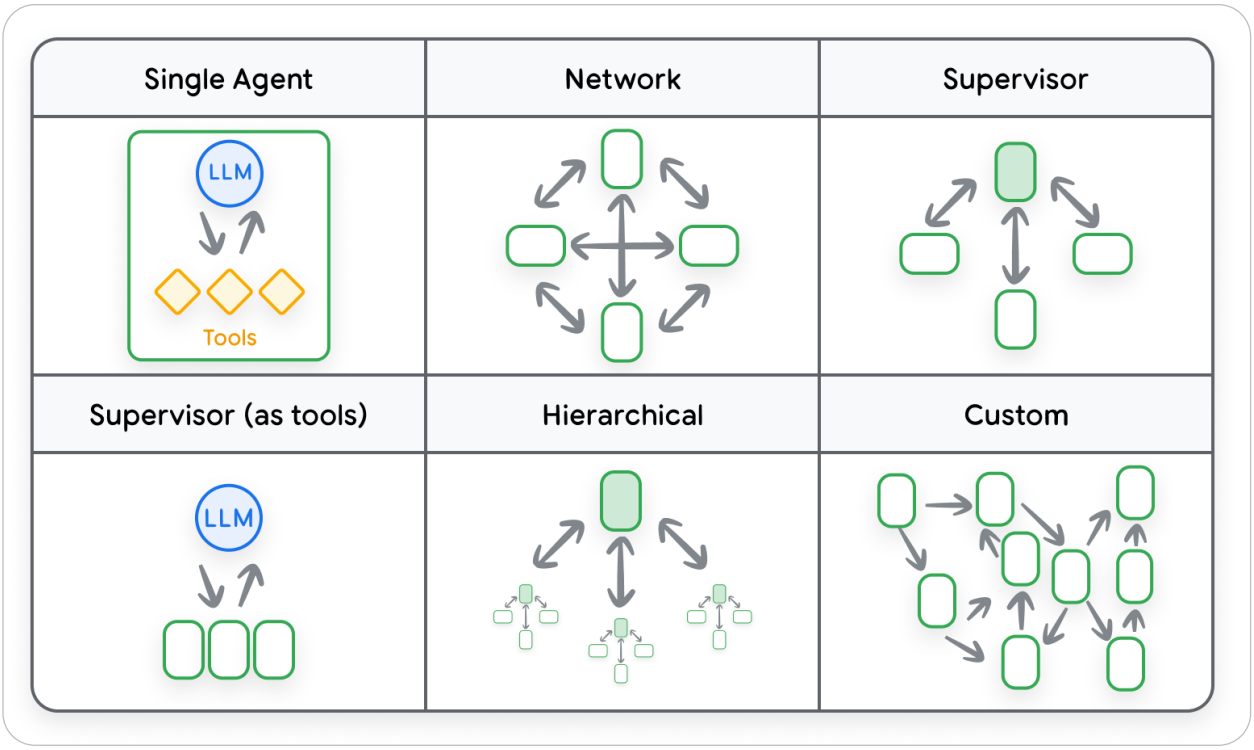


Figure 3: Different multi-agent architectural patterns[30]

Before exploring the architectural patterns for managing this history, it's crucial to distinguish it from the context sent to an LLM. Think of the session history as the permanent, unabridged transcript of the entire conversation. The context, on the other hand, is the carefully crafted information payload sent to the LLM for a single turn. An agent might construct this context by selecting only a relevant excerpt from the history or by adding special formatting, like a guiding preamble, to steer the model's response. This section focuses on what information is passed across agents, not necessarily what context is sent to the LLM.

Agent frameworks handle session history for multi-agent systems using one of two primary approaches: a shared, unified history where all agents contribute to a single log, or separate, individual histories where each agent maintains its own perspective[4]. The choice between these two patterns depends on the nature of the task and the desired collaboration style between the agents.

For the **shared, unified history** model, all agents in the system read from and write all events to the same, single conversation history. Every agent's message, tool call, and observation is appended to one central log in chronological order. This approach is best for tightly coupled, collaborative tasks requiring a single source of truth, such as a multi-step problem-solving process where one agent's output is the direct input for the next. Even with a shared history, a sub-agent might process the log before passing it to the LLM. For instance, it could filter for a subset of relevant events or add labels to identify which agent generated each event.

If you use ADK's LLM-driven delegation to handoff to sub-agents, all of the intermediary events of the sub-agent would be written to the same session as the root agent[5]:

Python

```python
from google.adk.agents import LlmAgent


# The sub-agent has access to Session and writes events to it.
sub_agent_1 = LlmAgent(...)


# Optionally, the sub-agent can save the final response text (or structured
output) to the specified state key.
sub_agent_2 = LlmAgent(
  ...,
  output_key="..."
)
```

Continues next page...

```
# Parent agent.
root_agent = LlmAgent(
  ...,
  sub_agents=[sub_agent_1, sub_agent_2]
)
```

Snippet 2: A2A communication across multiple agent frameworks

In the **separate, individual histories model**, each agent maintains its own private conversation history and functions like a black box to other agents. All internal processes—such as intermediary thoughts, tool use, and reasoning steps—are kept within the agent's private log and are not visible to others. Communication occurs only through explicit messages, where an agent shares its final output, not its process.

This interaction is typically implemented by either implementing Agent-as-a-tool or using the Agent-to-Agent (A2A) Protocol. With Agent-as a-Tool, one agent invokes another as if it were a standard tool, passing inputs and receiving a final, self-contained output[6]. With the Agent-to-Agent (A2A) Protocol, agents use a structured protocol for direct messaging[7].

We'll explore the A2A protocol in more detail in the next session.

Interoperability across multiple agent frameworks

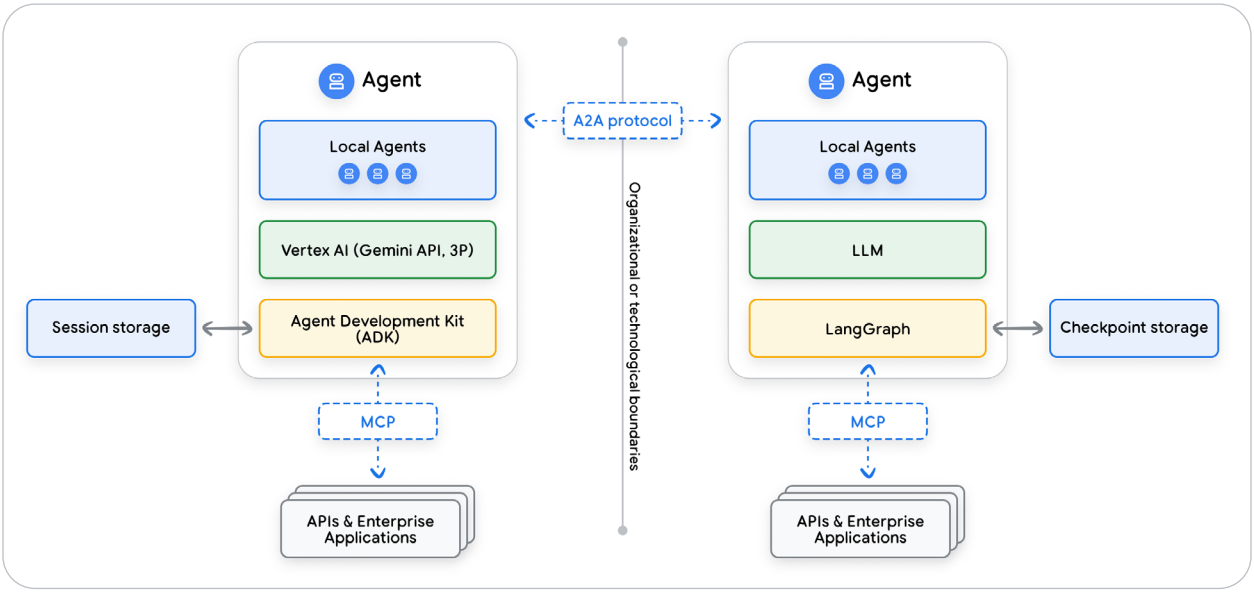


Figure 4: A2A communication across multiple agents that use different frameworks

A framework's use of an internal data representation introduces a critical architectural trade-off for multi-agent system: the very abstraction that decouples an agent from an LLM also isolates it from agents using other agent frameworks. This isolation is solidified at the persistence layer. The storage model for a `Session` typically couples the database schema directly to the framework's internal objects, creating a rigid, relatively non-portable conversation record. Therefore, an agent built with LangGraph cannot natively interpret the distinct `Session` and `Event` objects persisted by an ADK-based agent, making seamless task handoffs impossible.

One emerging architectural pattern architectural pattern for coordinating collaboration between these isolated agents is Agent-to-Agent (A2A) communication[8]. While this pattern enables agents to exchange messages, it fails to address the core problem of sharing rich, contextual state. Each agent's conversation history is encoded in its framework's internal schema. As a result, any A2A message containing session events requires a translation layer to be useful.

A more robust architectural pattern for interoperability involves abstracting shared knowledge into a framework-agnostic data layer, such as Memory. Unlike a `Session` store, which preserves raw, framework-specific objects like `Events` and `Messsages`, a memory layer is designed to hold **processed**, canonical information. Key information—like summaries, extracted entities, and facts—is extracted from the conversation and is typically stored as strings or dictionaries. The memory layer's data structures are not coupled to any single framework's internal data representation, which allows it to serve as a universal, common data layer. This pattern allows heterogeneous agents to achieve true collaborative intelligence by sharing a common cognitive resource without requiring custom translators.

Production Considerations for Sessions

When moving an agent to a production environment, its session management system must evolve from a simple log to a robust, enterprise-grade service. The key considerations fall into three critical areas: **security and privacy, data integrity, and performance**. A managed session store, like Agent Engine Sessions, is specifically designed to address these production requirements.

Security and Privacy

Protecting the sensitive information contained within a session is a non-negotiable requirement. **Strict Isolation** is the most critical security principle. A session is owned by a single user, and the system must enforce strict isolation to ensure one user can never access another user's session data (i.e. via ACLs). Every request to the session store must be authenticated and authorized against the session's owner.

A best practice for handling Personally Identifiable Information (PII) is to redact it before the session data is ever written to storage. This is a fundamental security measure that drastically reduces the risk and "blast radius" of a potential data breach. By ensuring sensitive data is never persisted using tools like Model Armor[9], you simplify compliance with privacy regulations like GDPR and CCPA and build user trust.

Data Integrity and Lifecycle Management

A production system requires clear rules for how session data is stored and maintained over time. Sessions should not live forever. You can implement a Time-to-Live (TTL) policy to automatically delete inactive sessions to manage storage costs and reducing data management overhead. This requires a clear data retention policy that defines how long sessions should be kept before being archived or permanently deleted.

Additionally, the system must guarantee that operations are appended to the session history in a **deterministic order**. Maintaining the correct chronological sequence of events is fundamental to the integrity of the conversation log.

Performance and Scalability

Session data is on the "hot path" of every user interaction, making its performance a primary concern. Reading and writing the session history must be extremely fast to ensure a responsive user experience. Agent runtimes are typically stateless, so the entire session history is retrieved from a central database at the start of every turn, incurring network transfer latency.

To mitigate latency, it is crucial to reduce the size of the data transferred. A key optimization is to filter or compact the session history before sending it to the agent. For example, you can remove old, irrelevant function call outputs that are no longer needed for the current state of the conversation. The following section details several strategies for compacting history to effectively manage long-context conversations.

Managing long context conversation: tradeoffs and optimizations

In a simplistic architecture, a session is an immutable log of the conversation between the user and agent. However, as the conversation scales, the conversation's token usage increases. Modern LLMs can handle long contexts, but limitations exist, especially for latency-sensitive applications[10]:

1. **Context Window Limits:** Every LLM has a maximum amount of text (context window) it can process at once. If the conversation history exceeds this limit, the API call will fail.
2. **API Costs ($):** Most LLM providers charge based on the number of tokens you send and receive. Shorter histories mean fewer tokens and lower costs per turn.

3. **Latency (Speed):** Sending more text to the model takes longer to process, resulting in a slower response time for the user. Compaction keeps the agent feeling quick and responsive.
4. **Quality:** As the number of tokens increases, performance can get worse due to additional noise in the context and autoregressive errors.

Managing a long conversation with an agent can be compared to a savvy traveler packing a suitcase for a long trip. The suitcase represents the agent's limited context window, and the clothes and items are the pieces of information from the conversation. If you simply try to stuff everything in, the suitcase becomes too heavy and disorganized, making it difficult to find what you need quickly—like how an overloaded context window increases processing costs and slows down response times. On the other hand, if you pack too little, you risk leaving behind essential items like a passport or a warm coat, compromising the entire trip—like how an agent could lose critical context, leading to irrelevant or incorrect answers. Both the traveler and the agent operate under a similar constraint: success hinges not on how much you can carry, but on carrying only what you need.

Compaction strategies shrink long conversation histories, condensing dialogue to fit within the model's context window, reducing API costs and latency. As a conversation gets longer, the history sent to the model with each turn can become too large. Compaction strategies solve this by intelligently trimming the history while trying to preserve the most important context.

So, how do you know **what** content to throw out of a Session without losing valuable information? Strategies range from simple truncation to sophisticated compaction:

- **Keep the last N turns:** This is the simplest strategy. The agent only keeps the most recent N turns of the conversation (a "sliding window") and discards everything older.

- **Token-Based Truncation:** Before sending the history to the model, the agent counts the tokens in the messages, starting with the most recent and working backward. It includes as many messages as possible without exceeding a predefined token limit (e.g., 4000 tokens). Everything older is simply cut off.
- **Recursive Summarization:** Older parts of the conversation are replaced by an AI-generated summary. As the conversation grows, the agent periodically uses another LLM call to summarize the oldest messages. This summary is then used as a condensed form of the history, often prefixed to the more recent, verbatim messages.

For example, you can **keep the last N turns** with ADK by using a built-in plug-in for your ADK app to limit the context sent to the model. This does not modify the historical events stored in your session storage:

Python

```python
from google.adk.apps import App
from google.adk.plugins.context_filter_plugin import ContextFilterPlugin


app = App(
    name='hello_world_app',
    root_agent=agent,
    plugins=[
       # Keep the last 10 turns and the most recent user query.
        ContextFilterPlugin(num_invocations_to_keep=10),
    ],
)
```

Snippet 3: Session truncation to only use the last N turns with ADK

Given that sophisticated compaction strategies aim to reduce cost and latency, it is critical to perform expensive operations (like recursive summarization) asynchronously in the background and persist the results. “In the background” ensures the client is not kept waiting, and “persistence” ensures that expensive computations are not excessively repeated. Frequently, the agent's memory manager is responsible for both generating and persisting these recursive summaries. The agent must also keep a record of which events are included in the compacted summary; this prevents the original, more verbose events from being needlessly sent to the LLM.

Additionally, the agent must decide **when** compaction is necessary. The trigger mechanism generally falls into a few distinct categories:

- **Count-Based Triggers** (i.e. token size or turn count threshold): The conversation is compacted once the conversation exceeds a certain predefined threshold. This approach is often “good enough" for managing context length.
- **Time-Based Triggers:** Compaction is triggered not by the size of the conversation, but by a lack of activity. If a user stops interacting for a set period (e.g., 15 or 30 minutes), the system can run a compaction job in the background.
- **Event-Based Triggers** (i.e. Semantic/Task Completion): The agent decides to trigger compaction when it detects that a specific task, sub-goal, or topic of conversation has concluded.

For example, you can use ADK’s `EventsCompactionConfig` to trigger LLM-based summarization after a configured number of turns:

Python

```python
from google.adk.apps import App
from google.adk.apps.app import EventsCompactionConfig


app = App(
    name='hello_world_app',
    root_agent=agent,
    events_compaction_config=EventsCompactionConfig(
        compaction_interval=5,
        overlap_size=1,
    ),
)
```

Snippet 4: Session compaction using summarization with ADK

Memory generation is the broad capability of extracting persistent knowledge from a verbose and noisy data source. In this section, we covered a primary example of extracting information from conversation history: session compaction. Compaction distills the verbatim transcript of an entire conversation, extracting key facts and summaries while discarding conversational filler.

Building on compaction, the next section will explore memory generation and management more broadly. We will discuss the various ways memories can be created, stored, and retrieved to build an agent's long-term knowledge.

Memory

Memory and Sessions share a deeply symbiotic relationship: sessions are the primary data source for generating memories, and memories are a key strategy for managing the size of a session. A memory is a snapshot of extracted, meaningful information from a conversation or data source. It's a condensed representation that preserves important context, making it useful for future interactions. Generally, memories are persisted across sessions to provide a continuous and personalized experience.

As a specialized, decoupled service, a "*memory manager*" provides the foundation for multi-agent interoperability. Memory managers frequently use framework-agnostic data structures, like simple strings and dictionaries. This allows agents built on different frameworks to connect to a single memory store, enabling the creation of a shared knowledge base that any connected agent can utilize.

Note: some frameworks may also refer to Sessions or verbatim conversation as "short-term memory." For this whitepaper, memories are defined as extracted information, not the raw dialogue of turn-by-turn conversation.

Storing and retrieving memories is crucial for building sophisticated and intelligent agents. A robust memory system transforms a basic chatbot into a truly intelligent agent by unlocking several key capabilities:

- **Personalization:** The most common use case is to remember user preferences, facts, and past interactions to tailor future responses. For example, remembering a user's favorite sports team or their preferred seat on an airplane creates a more helpful and personal experience.

- **Context Window Management:** As conversations become longer, the full history can exceed an LLM's context window. Memory systems can compact this history by creating summaries or extracting key facts, preserving context without sending thousands of tokens in every turn. This reduces both cost and latency.
- **Data Mining and Insight:** By analyzing stored memories across many users (in an aggregated, privacy-preserving way), you can extract insights from the noise. For example, a retail chatbot might identify that many users are asking about the return policy for a specific product, flagging a potential issue.
- **Agent Self-Improvement and Adaptation:** The agent learns from previous runs by creating procedural memories about its own performance—recording which strategies, tools, or reasoning paths led to successful outcomes. This enables the agent to build a playbook of effective solutions, allowing it to adapt and improve its problem-solving over time.

Creating, storing, and utilizing memory in an AI system is a collaborative process. Each component in the stack—from the end-user to the developer's code—has a distinct role to play.

1. **The User:** Provides the raw source data for memories. In some systems, users may provide memories directly (i.e. via a form).
2. **The Agent (Developer Logic):** Configures how to decide what and when to remember, orchestrating calls to the memory manager. In simple architectures, the developer can implement the logic such that memory is *always* retrieved and *always* triggered-to-be-generated. In more advanced architectures, the developer may implement memory-as-a-tool, where the agent (via LLM) decides when memory should be retrieved or generated.

3. **The Agent Framework (e.g., ADK, LangGraph):** Provides the structure and tools for memory interaction. The framework acts as the plumbing. It defines how the developer's logic can access conversation history and interact with the memory manager, but it doesn't manage the long-term storage itself. It also defines how to stuff retrieved memories into the context window.
4. **The Session Storage (i.e. Agent Engine Sessions, Spanner, Redis):** Stores the turn-by-turn conversation of the Session. The raw dialogue will be ingested into the memory manager in order to generate memories.
5. **The Memory Manager (e.g. Agent Engine Memory Bank, Mem0, Zep):** Handles the storage, retrieval, and compaction of memories. The mechanisms to store and retrieve memories depend on what provider is used. This is the specialized service or component that takes the potential memory identified by the agent and handles its entire lifecycle.
    - **Extraction** distills the key information from the source data.
    - **Consolidation** curates memories to merge duplicative entities.
    - **Storage** persists the memory to persistent databases.
    - **Retrieval** fetches relevant memories to provide context for new interactions

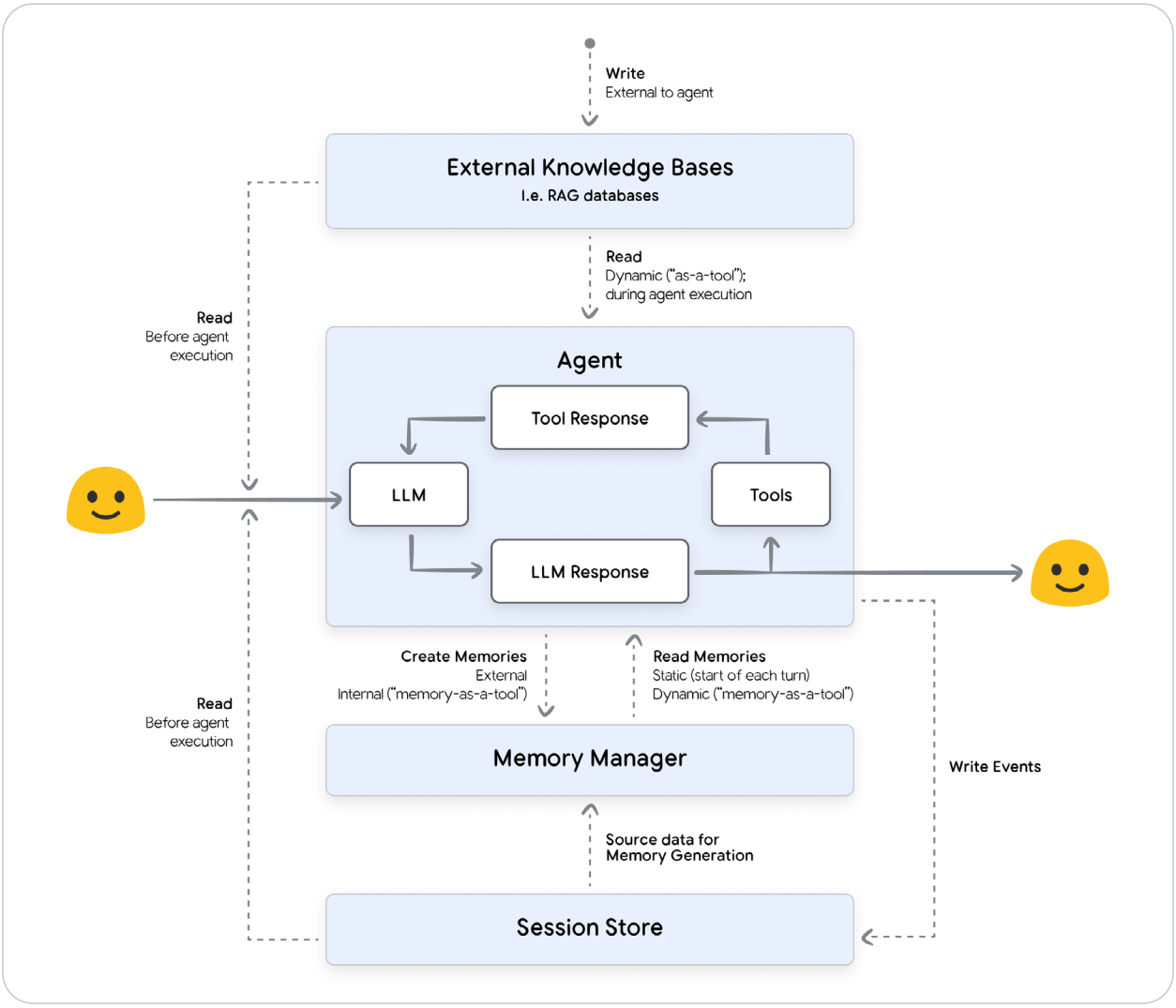


Figure 5: The flow of information between sessions, memory, and external knowledge

The division of responsibilities ensures that the developer can focus on the agent's unique logic without having to build the complex underlying infrastructure for memory persistence and management. It is important to recognize that a memory manager is an active system, not just a passive vector database. While it uses similarity search for retrieval, its core value

lies in its ability to intelligently extract, consolidate, and curate memories over time. Managed memory services, like Agent Engine Memory Bank, handle the entire lifecycle of memory generation and storage, freeing you to focus on your agent's core logic.

This retrieval capability is also why memory is frequently compared to another key architectural pattern: Retrieval-Augmented Generation (RAG). However, they are built on different architectural principles, as RAG handles static, external data while Memory curates dynamic, user-specific context. They fulfill two distinct and complementary roles: RAG makes an agent an expert on facts, while memory makes it an expert on the user. The following chart breaks down their high-level differences:

	RAG Engines	Memory Managers
Primary Goal	To inject **external, factual** knowledge into the context	To create a personalized and stateful experience. The agent remembers facts, adapts to the user over time, and maintains long-running context.
Data source	A static, pre-indexed external knowledge base (e.g., PDFs, wikis, documents, APIs).	The dialogue between the user and agent.
Isolation Level	**Generally Shared**. The knowledge base is typically a global, read-only resource accessible by all users to ensure consistent, factual answers.	**Highly Isolated:** Memory is almost always scoped per-user to prevent data leaks.
Information type	Static, factual, and authoritative. Often contains domain-specific data, product details, or technical documentation.	Dynamic and (generally) user-specific. Memories are derived from conversation, so there's an inherent level of uncertainty.
Write patterns	Batch processing Triggered via an offline, administrative action.	Event-based processing Triggered at some cadence (i.e. every turn or at the end of a session) or Memory-as-a-tool (agent decides to generate memories).
Read patterns	RAG data is almost always retrieved "**as-a-tool**". It's retrieved when the agent decides that the user's query requires external information.	There are two common read patterns: • **Memory-as-a-tool:** Retrieved when the user's query requires additional information about the user (or some other identity). • **Static retrieval:** Memory is always retrieved at the start of each turn.
Data Format	A natural-language "chunk".	A natural language snippet or a structured profile.
Data preparation	**Chunking and Indexing:** Source documents are broken into smalvler Chunks, which are then converted to embeddings and stored for fast lookup.	**Extraction and consolidation:** Extract key details from the conversation, ensuring content is not duplicative or contradictory.

Table 1: Comparison of RAG engines and memory managers

A helpful way to understand the difference is to think of RAG as the agent's research librarian and a memory manager as its personal assistant.

The research librarian (**RAG**) works in a vast public library filled with encyclopedias, textbooks, and official documents. When the agent needs an established fact—like a product's technical specifications or a historical date—it consults the librarian. The librarian retrieves information from this static, shared, and authoritative knowledge base to provide consistent, factual answers. The librarian is an expert on the world's facts, but they don't know anything personal about the user asking the question.

In contrast, the personal assistant (**memory**) follows the agent and carries a private notebook, recording the details of every interaction with a specific user. This notebook is dynamic and highly isolated, containing personal preferences, past conversations, and evolving goals. When the agent needs to recall a user's favorite sports team or the context of last week's project discussion, it turns to the assistant. The assistant's expertise is not in global facts, but in the user themselves.

Ultimately, a truly intelligent agent needs both. RAG provides it with expert knowledge of the world, while memory provides it with an expert understanding of the user it's serving.

The next section deconstructs the concept of memory by examining its core components: the types of information it stores, the patterns for its organization, the mechanisms for its storage and creation, the strategic definition of its scope, and its handling of multimodal versus textual data.

Types of memory

An agent's memory can be categorized by how the information is stored and how it was captured. These different types of memory work together to create a rich, contextual understanding of a user and their needs. Across all types of memories, the rule stands that memories are descriptive, not predictive.

A “memory” is an atomic piece of context that is returned by the memory manager and used by the agent as context. While the exact schema can vary, a single memory generally consists of two main components: **content** and **metadata**.

Content is the substance of the memory that was extracted from the source data (i.e. the raw dialogue of the session). Crucially, the content is designed to be framework-agnostic, using simple data structures that any agent can easily ingest. The content can either be structured or unstructured data. **Structured memories** include information typically stored in universal formats like a dictionary or JSON. Its schema is typically defined by the developer, not a specific framework. For example, `{“seat_preference”: “Window”}`. **Unstructured memories** are natural language descriptions that capture the essence of a longer interaction, event, or topic. For example, “The user prefers a window seat.”

Metadata provides context about the memory, typically stored as a simple string. This can include a unique identifier for the memory, identifiers for the “owner” of the memory, and labels describing the content or data source of the memory.

Types of information

Beyond their basic structure, memories can be classified by the fundamental type of knowledge they represent. This distinction, crucial for understanding how an agent uses memories, separates memory into two primary functional categories derived from cognitive science[11]: **declarative memories** (*"knowing what"*) and **procedural memories** (*"knowing how"*).

Declarative memory is the agent's knowledge of facts, figures, and events. It's all the information that the agent can explicitly state or "declare." If the memory is an answer to a "what" question, it's declarative. This category encompasses both general world knowledge (Semantic) and specific user facts (Entity/Episodic).

Procedural memory is the agent's knowledge of skills and workflows. It guides the agent's actions by demonstrating implicitly how to perform a task correctly. If the memory helps answer a "how" question—like the correct sequence of tool calls to book a trip—it's procedural.

Organization patterns

Once a memory is created, the next question is how to organize it. Memory managers typically employ one or more of the following patterns to organize memories: Collections[12], **Structured User Profile**, or **"Rolling Summary"**. The patterns define how individual memories relate to each other and to the user.

The collections[13] pattern organizes content into multiple self-contained, natural language memories for a single user. Each memory is a distinct event, summary, or observation, although there may be multiple memories in the collection for a single high-level topic. Collections allow for storing and searching through a larger, less structured pool of information related to specific goals or topics.

The **structured user profile** pattern organizes memories as a set of core facts about a user, like a contact card that is continuously updated with new, stable information. It's designed for quick lookups of essential, factual information like names, preferences, and account details.

Unlike a structured user profile, the **"rolling" summary** pattern consolidates all information into a single, evolving memory that represents a natural-language summary of the entire user-agent relationship. Instead of creating new, individual memories, the manager continuously updates this one master document. This pattern is frequently used to compact long Sessions, preserving vital information while managing the overall token count.

Storage architectures

Additionally, the storage architecture is a critical decision that determines how quickly and intelligently an agent can retrieve memories. The choice of architecture defines whether the agent excels at finding conceptually similar ideas, understanding structured relationships, or both.

Memories are generally stored in **vector databases** and/or **knowledge graphs**. Vector databases help find memories that are conceptually similar to the query. Knowledge graphs store memories as a network of entities and their relationships.

Vector databases are the most common approach, enabling retrieval based on semantic similarity rather than exact keywords. Memories are converted into embedding vectors, and the database finds the closest conceptual matches to a user's query. This excels at retrieving unstructured, natural language memories where context and meaning are key (i.e. "atomic facts"[14]).

Knowledge graphs are used to store memories as a network of entities (nodes) and their relationships (edges). Retrieval involves traversing this graph to find direct and indirect connections, allowing the agent to reason about how different facts are linked. It is ideal for structured, relational queries and understanding complex connections within the data (i.e. "knowledge triples"[15]).

You can also combine both methods into a **hybrid approach** by enriching a knowledge graph's structured entities with vector embeddings. This enables the system to perform both relational and semantic searches simultaneously. This provides the structured reasoning of a graph and the nuanced, conceptual search of a vector database, offering the best of both worlds.

Creation mechanisms

We can also classify memories by how they were created, including how the information was derived. **Explicit memories** are created when the user gives a direct command to the agent to remember something (e.g., "Remember my anniversary is October 26th"). On the other hand, **implicit memories** are created when the agent infers and extracts information from the conversation without a direct command (e.g., "My anniversary is next week. Can you help me find a gift for my partner?")

Memories can also be distinguished by whether the memory extraction logic is located internally or externally to the agent framework. **Internal memory** refers to memory management that is built directly into the agent framework. It's convenient for getting started but often lacks advanced features. Internal memory can use external storage, but the mechanism for generating memories is internal to the agent.

External Memory involves using a separate, specialized service dedicated to memory management (e.g., Agent Engine Memory Bank, Mem0, Zep). The agent framework makes API calls to this external service to store, retrieve, and process memories. This approach provides more sophisticated features like semantic search, entity extraction, and automatic summarization, offloading the complex task of memory management to a purpose-built tool.

Memory scope

You also need to consider *who* or *what* a memory describes. This has implications on what entity (i.e. a **user**, **session**, or **application**) you use to aggregate and retrieve memories.

User-Level scope is the most common implementation, designed to create a continuous, personalized experience for each individual; for example, *"the User prefers the middle seat."* Memories are tied to a specific user ID and persist across all their sessions, allowing the agent to build a long-term understanding of their preferences and history.

Session-Level scope is designed for the compaction of long conversations; for example, *"the User is shopping for tickets between New York and Paris between November 7, 2025 and November 14, 2025. They prefer direct flights and the middle seat"*. It creates a persistent record of insights extracted from a single session, allowing an agent to replace the verbose,

token-heavy transcript with a concise set of key facts. Crucially, this memory is distinct from the raw session log; it contains only the processed insights from the dialogue, not the dialogue itself, and its context is isolated to that specific session.

Application-level scope (or global context), are memories accessible by all users of an application; for example, *"The codename XYZ refers to the project...."* This scope is used to provide shared context, broadcast system-wide information, or establish a baseline of common knowledge. A common use case for application-level memories is *procedural memories*, which provide "how-to" instructions for the agent; the memories are generally intended to help with the agent's reasoning for all users. It is critical that these memories are sanitized of all sensitive content to prevent data leaks between users.

Multimodal memory

"Multimodal memory" is a crucial concept that describes how an agent handles non-textual information, like images, videos, and audio. The key is to distinguish between the data the memory is *derived from* (its **source**) and the data the memory is stored as (its **content**).

Memory from a multimodal source is the most common implementation. The agent can process various data types—text, images, audio—but the memory it creates is a **textual insight** derived from that source. For example, an agent can process a user's voice memo to create memories. It doesn't store the audio file itself; instead, it transcribes the audio and creates a textual memory like, "User expressed frustration about the recent shipping delay."

Memory with Multimodal Content is a more advanced approach where the memory itself contains non-textual media. The agent doesn't just describe the content; it stores the content directly. For example, a user can upload an image and say "Remember this design for our logo." The agent creates a memory that directly contains the image file, linked to the user's request.

Most contemporary memory managers focus on handling multimodal sources while producing textual content. This is because generating and retrieving unstructured binary data like images or audio for a specific memory requires specialized models, algorithms, and infrastructure. It is far simpler to convert all inputs into a common, searchable format: text.

For example, you can generate memories from multimodal input[16] using Agent Engine Memory Bank. The output memories will be textual insights extracted from the content:

Python

```
from google.genai import types


client = vertexai.Client(project=..., location=...)
response = client.agent_engines.memories.generate(
       name=agent_engine_name,
       direct_contents_source={
           "events": [
             {
                "content": types.Content(
                   role="user",
                   parts=[
                     types.Part.from_text(
                            "This is context about the multimodal input."
                     ),
```

Continues next page...

```
                    types.Part.from_bytes(
                           data=CONTENT_AS_BYTES,
                           mime_type=MIME_TYPE
                    ),
                    types.Part.from_uri(
                           file_uri="file/path/to/content",
                           mime_type=MIME_TYPE
                    )
        ])}]},
        scope={"user_id": user_id}
)
```

Snippet 5: Example memory generation API call for Agent Engine Memory Bank

The next section examines the mechanics of memory generation, detailing the two core stages: the extraction of new information from source data, and the subsequent consolidation of that information with the existing memory corpus.

Memory Generation: Extraction and Consolidation

Memory generation autonomously transforms raw conversational data into structured, meaningful insights, functioning. Think of it as an **LLM-driven ETL (Extract, Transform, Load) pipeline** designed to extract and condense memories. Memory generation's ETL pipeline distinguishes memory managers from RAG engines and traditional databases.

Rather than requiring developers to manually specify database operations, a memory manager uses an LLM to intelligently decide when to add, update, or merge memories. This automation is a memory manager's core strength; it abstracts away the complexity of managing the database contents, chaining together LLM calls, and deploying background services for data processing.

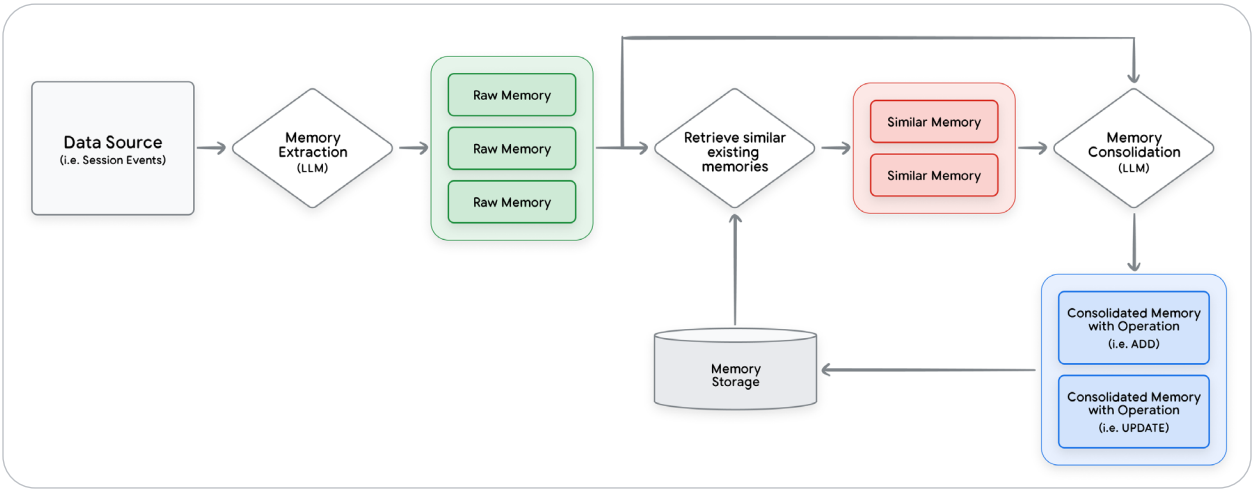


Figure 6: High-level algorithm of memory generation which extracts memories from new data sources and consolidates them with existing memories

While the specific algorithms vary by platform (e.g., Agent Engine Memory Bank, Mem0, Zep), the high-level process of memory generation generally follows these four stages:

1. **Ingestion:** The process begins when the client provides a source of raw data, typically a conversation history, to the memory manager.
2. **Extraction & Filtering:** The memory manager uses an LLM to extract meaningful content from the source data. The key is that this LLM doesn't extract everything; it only captures information that fits a predefined **topic definition**. If the ingested data contains no information that matches these topics, no memory is created.
3. **Consolidation:** This is the most sophisticated stage, where the memory manager handles conflict resolution and deduplication. It performs a "self-editing" process, using an LLM to compare the newly extracted information with existing memories. To ensure the user's knowledge base remains coherent, accurate, and evolves over time based on new information, the manager can decide to:
    - **Merge** the new insight into an existing memory.

- **Delete** an existing memory if it's now invalidated.
- **Create** an entirely new memory if the topic is novel.

4. **Storage:** Finally, the new or updated memory is persisted to a durable storage layer (such as a vector database or knowledge graph) so it can be retrieved in future interactions.

A managed memory manager, like Agent Engine Memory Bank, fully automates this pipeline. They provide a single, coherent system for turning conversational noise into structured knowledge, allowing developers to focus on agent logic rather than building and maintaining the underlying data infrastructure themselves. For example, triggering memory generation with Memory Bank only requires a simple API call[17]:

Python

```
from google.cloud import vertexai

client = vertexai.Client(project=..., location=...)

client.agent_engines.memories.generate(
       name="projects/.../locations/...reasoningEngines/...",
       scope={"user_id": "123"},
       direct_contents_source={
              "events": [...]
       },
       config={
              # Run memory generation in the background.
              "wait_for_completion": False
       }
)
```

Snippet 6: Generate memories with Agent Engine Memory Bank

The process of memory generation can be compared to the work of a diligent gardener tending to a garden. Extraction is like receiving new seeds and saplings (new information from a conversation). The gardener doesn't just throw them randomly onto the plot. Instead, they perform Consolidation by pulling out weeds (deleting redundant or conflicting data), pruning back overgrown branches to improve the health of existing plants (refining and summarizing existing memories), and then carefully planting the new saplings in the optimal location. This constant, thoughtful curation ensures the garden remains healthy, organized, and continues to flourish over time, rather than becoming an overgrown, unusable mess. This asynchronous process happens in the background, ensuring the garden is always ready for the next visit.

Now, let’s dive into the two key steps of memory generation: extraction and consolidation.

Deep-dive: Memory Extraction

The goal of memory extraction is to answer the fundamental question: **"What information in this conversation is meaningful enough to become a memory?"** This is not simple summarization; it is a targeted, intelligent filtering process designed to separate the signal (important facts, preferences, goals) from the noise (pleasantries, filler text).

"Meaningful" is not a universal concept; it is defined entirely by the agent's purpose and use case. What a customer support agent needs to remember (e.g., order numbers, technical issues) is fundamentally different from what a personal wellness coach needs to remember (e.g., long-term goals, emotional states). Customizing what information is preserved is therefore the key to creating a truly effective agent.

The memory manager's LLM decides what to extract by following a carefully constructed set of programmatic guardrails and instructions, usually embedded in a complex system prompt. This prompt defines what "meaningful" means by providing the LLM with a set of topic definitions. With schema and template-based extraction, the LLM is given a predefined JSON schema or a template using LLM features like structured output[18]; the LLM is instructed to construct the JSON using corresponding information in the conversation. Alternatively, with natural language topic definitions, the LLM is guided by a simple natural language description of the topic.

With few-shot prompting, the LLM is "shown" what information to extract using examples. The prompt includes several examples of input text and the ideal, high-fidelity memory that should be extracted. The LLM learns the desired extraction pattern from the examples, making it highly effective for custom or nuanced topics that are difficult to describe with a schema or a simple definition.

Most memory managers work out-of-the-box by looking for common topics, such as user preferences, key facts, or goals. Many platforms also allow developers to define their own custom topics, tailoring the extraction process to their specific domain. For example, you can customize what information Agent Engine Memory Bank considers to be meaningful to be persisted by providing your own topic definitions and few-shot examples[19]:

Python

```
from google.genai.types import Content, Part


# See https://cloud.google.com/agent-builder/agent-engine/memory-bank/set-up for
more information.
memory_bank_config = {
  "customization_configs": [{
    "memory_topics": [
        { "managed_memory_topic": {"managed_topic_enum": "USER_PERSONAL_INFO" }},
```

Continues next page...

```
        {
            "custom_memory_topic": {
             "label": "business_feedback",
             "description": """Specific user feedback about their experience at the coffee
shop. This includes opinions on drinks, food, pastries, ambiance, staff friendliness,
service speed, cleanliness, and any suggestions for improvement."""
            }
         }
       ],
    "generate_memories_examples": {
      "conversationSource": {
        "events": [
          {
            "content": Content(
                    role="model",
                    parts=[Part(text="Welcome back to The Daily Grind! We'd love to hear
your feedback on your visit.")])
             },{
            "content": Content(
                    role="user",
                    parts=[Part(text= "Hey. The drip coffee was a bit lukewarm today, which
was a bummer. Also, the music was way too loud, I could barely hear my friend.")])
           }]
       },
       "generatedMemories": [
         {"fact": "The user reported that the drip coffee was lukewarm."},
         {"fact": "The user felt the music in the shop was too loud."}
       ]
  }
  }]
}


agent_engine = client.agent_engines.create(
       config={
              "context_spec": {"memory_bank_config": memory_bank_config }
       }
)
```

Snippet 7: Customizing what information Agent Engine Memory Bank considers meaningful to persist

Although memory extraction itself is not "summarization," the algorithm may incorporate summarization to distill information. To enhance efficiency, many memory managers incorporate a rolling summary of the conversation directly into the [memory extraction]

prompt[20]. This condensed history provides the necessary context to extract key information from the most recent interactions. It eliminates the need to repeatedly process the full, verbose dialogue with each turn to maintain context.

Once information has been extracted from the data source, the existing corpus of memories must be updated to reflect the new information via consolidation.

Deep-dive: Memory Consolidation

After memories are extracted from the verbose conversation, **consolidation** should integrate the new information into a coherent, accurate, and evolving knowledge base. It is arguably the most sophisticated stage in the memory lifecycle, transforming a simple collection of facts into a curated understanding of the user. Without consolidation, an agent's memory would quickly become a noisy, contradictory, and unreliable log of every piece of information ever captured. This "self-curation" is typically managed by an LLM and is what elevates a memory manager beyond a simple database.

Consolidation addresses fundamental problems arising from conversational data, including:

- **Information Duplication:** A user might mention the same fact in multiple ways across different conversations (e.g., "I need a flight to NYC" and later "I'm planning a trip to New York"). A simple extraction process would create two redundant memories.
- **Conflicting Information:** A user's state changes over time. Without consolidation, the agent's memory would contain contradictory facts.
- **Information Evolution:** A simple fact can become more nuanced. An initial memory that "the user is interested in marketing" might evolve into "the user is leading a marketing project focused on Q4 customer acquisition."

- **Memory Relevance Decay:** Not all memories remain useful forever. An agent must engage in **forgetting**—proactively **pruning** old, stale, or low-confidence memories to keep the knowledge base relevant and efficient. Forgetting can happen by instructing the LLM to defer to newer information during consolidation or through automatic deletion via a time-to-live (TTL).

The consolidation process is an LLM-driven workflow that compares newly extracted insights against the user's existing memories. First, the workflow tries to retrieve existing memories that are similar to the newly extracted memories. These existing memories are candidates for consolidation. If the existing memory is contradicted by the new information, it may be deleted. If it is augmented, it may be updated.

Second, an LLM is presented with both the *existing memories* and the *new information*. Its core task is to analyze them together and identify what operations should be performed. The primary operations include:

- **UPDATE:** Modify an existing memory with new or corrected information.
- **CREATE:** If the new insight is entirely novel and unrelated to existing memories, create a new one.
- **DELETE / INVALIDATE:** If the new information makes an old memory completely irrelevant or incorrect, delete or invalidate it.

Finally, the memory manager translates the LLM's decision into a transaction that updates the memory store.

Memory Provenance

The classic machine learning axiom of *"garbage in, garbage out"* is even more critical for LLMs, where the outcome is often *"garbage in, confident garbage out."* For an agent to make reliable decisions and for a memory manager to effectively consolidate memories, they must be able to critically evaluate the quality of its own memories. This trustworthiness is derived directly from a memory's **provenance**—a detailed record of its origin and history.

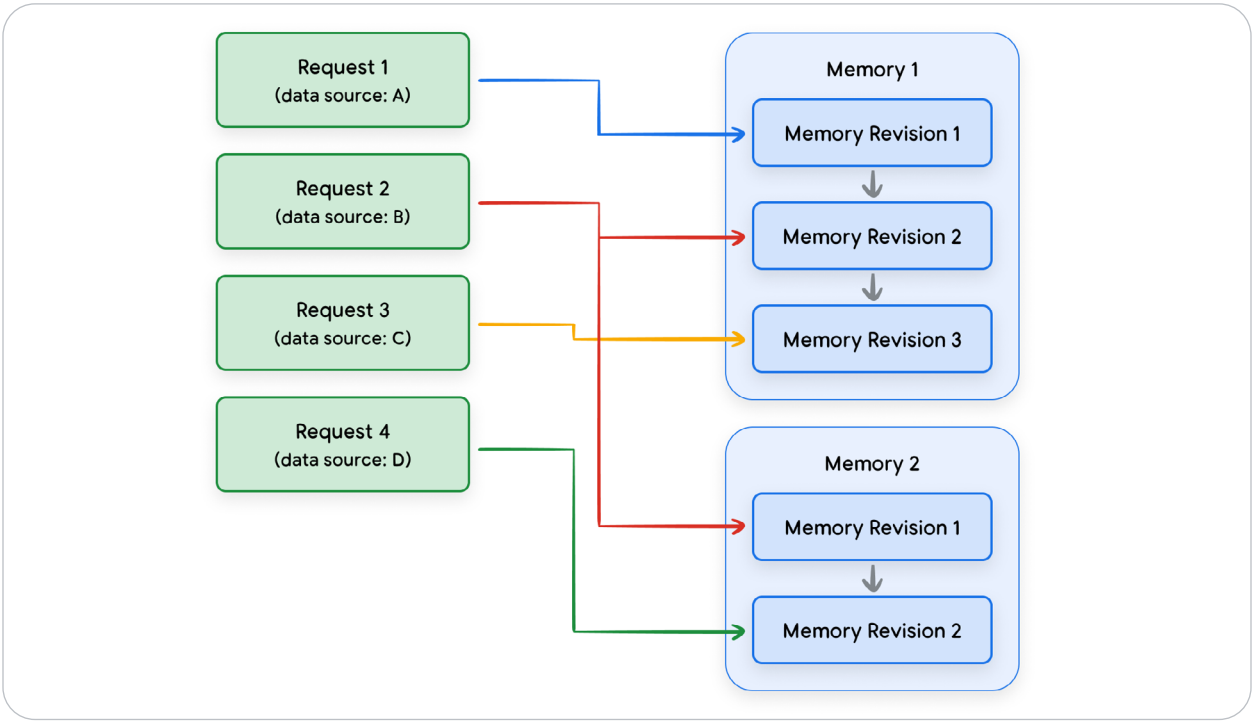


Figure 7: The flow of information between data sources and memories. A single memory can be derived from multiple data sources, and a single data source may contribute to multiple memories.

The process of **memory consolidation**—merging information from multiple sources into a single, evolving memory—creates the need to track its lineage. As shown in the diagram above, a single memory might be a blend of multiple data sources, and a single source might be segmented into multiple memories.

To assess trustworthiness, the agent must track key details for each source, such as its origin (source type) and age (“freshness”). These details are critical for two reasons: they dictate the weight each source has during memory consolidation, and they inform how much the agent should rely on that memory during inference.

The source type is one of the most important factors in determining trust. Data sources fall into three main categories:

- **Bootstrapped Data:** Information pre-loaded from internal systems, such as a CRM. This high-trust data can be used to initialize a user's memories to address the cold-start problem, which is the challenge of providing a personalized experience to a user the agent has never interacted with before.
- **User Input:** This includes data provided explicitly (e.g., via a form, which is high-trust) or information extracted implicitly from a conversation (which is generally less trustworthy).
- **Tool Output:** Data returned from an external tool call. Generating memories from Tool Output is generally discouraged because these memories tend to be brittle and stale, making this source type better suited for short-term caching.

Accounting for memory lineage during memory management

This dynamic, multi-source approach to memory creates two primary operational challenges when managing memories: **conflict resolution** and **deleting derived data**.

Memory consolidation inevitably leads to conflicts where one data source conflicts with another. A memory’s provenance allows the memory manager to establish a hierarchy of trust for its information sources. When memories from different sources contradict each

other, the agent must use this hierarchy in a conflict resolution strategy. Common strategies include prioritizing the most trusted source, favoring the most recent information, or looking for corroboration across multiple data points.

Another challenge to managing memories occurs when deleting memories. A memory can be derived from multiple data sources. When a user revokes access to one data source, data derived from that source should also be removed. Deleting every memory "touched" by that source can be overly aggressive. A more precise, though computationally expensive, approach is to regenerate the affected memories from scratch using only the remaining, valid sources.

Beyond static provenance details, confidence in a memory must evolve. Confidence increases through corroboration, such as when multiple trusted sources provide consistent information. However, an efficient memory system must also actively curate its existing knowledge through memory pruning—a process that identifies and "forgets" memories that are no longer useful. This pruning can be triggered by several factors.

- **Time-based Decay:** The importance of a memory can decrease over time. A memory about a meeting from two years ago is likely less relevant than one from last week.
- **Low Confidence:** A memory that was created from a weak inference and was never corroborated by other sources may be pruned.
- **Irrelevance:** As an agent gains a more sophisticated understanding of a user, it might determine that some older, trivial memories are no longer relevant to the user's current goals.

By combining a reactive consolidation pipeline with proactive pruning, the memory manager ensures that the agent's knowledge base is not just a growing log of everything ever said. Instead, it’s a curated, accurate, and relevant understanding of the user.

Accounting for memory lineage during inference

In addition to accounting for a memory's lineage while curating the corpus's contents, a memory's trustworthiness should also be considered at inference time. An agent's confidence in a memory should not be static; it must evolve based on new information and the passage of time. Confidence increases through corroboration, such as when multiple trusted sources provide consistent information. Conversely, confidence decreases (or decays) over time as older memories become stale, and it also drops when contradictory information is introduced. Eventually, the system can "forget" by archiving or deleting low-confidence memories. This dynamic confidence score is critical during inference time. Rather than being shown to the user, memories and, if available, their confidence scores are injected into the prompt, enabling the LLM to assess information reliability and make more nuanced decisions.

This entire trust framework serves the agent's internal reasoning process. Memories and their confidence scores are not typically shown to the user directly. Instead, they are injected into the system prompt, allowing the LLM to weigh the evidence, consider the reliability of its information, and ultimately make more nuanced and trustworthy decisions.

Triggering memory generation

Although memory managers automate memory extraction and consolidation once generation is triggered, the agent must still decide when memory generation should be attempted. This is a critical architectural choice, balancing data freshness against computational cost and latency. This decision is typically managed by the agent's logic, which can employ several triggering strategies. Memory generation can be initiated based on various events:

- **Session Completion:** Triggering generation at the end of a multi-turn session.

- **Turn Cadence:** Running the process after a specific number of turns (e.g., every 5 turns).
- **Real-Time:** Generating memories after every single turn.
- **Explicit Command:** Activating the process upon a direct user command (e.g., "Remember this"

The choice of trigger involves a direct tradeoff between cost and fidelity. **Frequent generation** (e.g., real-time) ensures memories are highly detailed and fresh, capturing every nuance of the conversation. However, this incurs the highest LLM and database costs and can introduce latency if not handled properly. **Infrequent generation** (e.g., at session completion) is far more cost-effective but risks creating lower-fidelity memories, as the LLM must summarize a much larger block of conversation at once. You also want to be careful that the memory manager is not processing the same events multiple times, as that introduces unnecessary cost.

Memory-as-a-Tool

A more sophisticated approach is to allow the agent to decide for itself when to create a memory. In this pattern, memory generation is exposed as a tool (i.e. `create_memory`); the tool definition should define what types of information should be considered meaningful. The agent can then analyze the conversation and autonomously decide to call this tool when it identifies information that is meaningful to persist. This shifts the responsibility for identifying "meaningful information" from the external memory manager to the agent (and thus you as the developer) itself.

For example, you can do this using ADK by packaging your memory generation code into a Tool[21] that the agent decides to invoke when it deems the conversation meaningful to persist. You can send the Session to Memory Bank, and Memory Bank will extract and consolidate memories from the conversation history:

Python

```python
from google.adk.agents import LlmAgent
from google.adk.memory import VertexAiMemoryBankService
from google.adk.runners import Runner
from google.adk.tools import ToolContext

def generate_memories(tool_context: ToolContext):
   """Triggers memory generation to remember the session."""
   # Option 1: Extract memories from the complete conversation history using the
   # ADK memory service.
   tool_context._invocation_context.memory_service.add_session_to_memory(
       session)


   # Option 2: Extract memories from the last conversation turn.
   client.agent_engines.memories.generate(
       name="projects/.../locations/...reasoningEngines/...",
       direct_contents_source={
           "events": [
              {"content": tool_context._invocation_context.user_content}
           ]
       },
       scope={
           "user_id": tool_context._invocation_context.user_id,
           "app_name": tool_context._invocation_context.app_name
       },
       # Generate memories in the background
       config={"wait_for_completion": False}
   )
   return {"status": "success"}

agent = LlmAgent(
    ...,
    tools=[generate_memories]
)

runner = Runner(
    agent=agent,
    app_name=APP_NAME,
    session_service=session_service,
    memory_service=VertexAiMemoryBankService(
      agent_engine_id=AGENT_ENGINE_ID,
      project=PROJECT,
      location=LOCATION
    )
)
```

Snippet 8: ADK agent using a custom tool to trigger memory generation. Memory Bank will extract and consolidate the memories.

Another approach is to leverage internal memory, where the agent actively decides what to remember from a conversation. In this workflow, the agent is responsible for extracting key information. Optionally, these extracted memories are then sent to [Agent Engine Memory Bank] to be consolidated with the user's existing memories[22]:

Python

```
def extract_memories(query: str, tool_context: ToolContext):
   """Triggers memory generation to remember information.

   Args:
     query: Meaningful information that should be persisted about the user.
   """
   client.agent_engines.memories.generate(
       name="projects/.../locations/...reasoningEngines/...",
       # The meaningful information is already extracted from the conversation, so we
       # just want to consolidate it with existing memories for the same user.
       direct_memories_source={
           "direct_memories": [{"fact": query}]
       },
       scope={
           "user_id": tool_context._invocation_context.user_id,
           "app_name": tool_context._invocation_context.app_name
       },
       config={"wait_for_completion": False}
   )
   return {"status": "success"}

agent = LlmAgent(
    ...,
    tools=[extract_memories]
)
```

Snippet 9: ADK agent using a custom tool to extract memories from the conversation and trigger consolidation with Agent Engine Memory Bank. Unlike Snippet 8, the agent is responsible for extracting memories, not Memory Bank.

Background vs. Blocking Operations

Memory generation is an expensive operation requiring LLM calls and database writes. For agents in production, memory generation should almost always be handled **asynchronously as a background process**[23].

After an agent sends its response to the user, the memory generation pipeline can run in parallel without blocking the user experience. This decoupling is essential for keeping the agent feeling fast and responsive. A blocking (or synchronous) approach, where the user has to wait for the memory to be written before receiving a response, would create an unacceptably slow and frustrating user experience. This necessitates that memory generation occurs in a service that is architecturally separate from the agent's core runtime.

Memory Retrieval

With a mechanism for memory generation in place, your focus can shift to the critical task of retrieval. An intelligent retrieval strategy is essential for an agent's performance, encompassing decisions about which memories should be retrieved and when to retrieve them.

The strategy for retrieving a memory depends heavily on how memories are organized. For a **structured user profile**, retrieval is typically a straightforward lookup for the full profile or a specific attribute. For a **collection of memories**, however, retrieval is a far more complex search problem. The goal is to discover the most pertinent, conceptually related information from a large pool of unstructured or semi-structured data. The strategies discussed in this section are designed to solve this complex retrieval challenge for memory collections.

Memory retrieval searches for the most pertinent memories for the current conversation. An effective retrieval strategy is crucial; providing irrelevant memories can confuse the model and degrade its response, while finding the perfect piece of context can lead to a remarkably intelligent interaction. The core challenge is balancing memory 'usefulness' within a strict latency budget.

Advanced memory systems go beyond a simple search and score potential memories across multiple dimensions to find the best fit.

- **Relevance (Semantic Similarity):** How conceptually related is this memory to the current conversation?
- **Recency (Time-based):** How recently was this memory created?
- **Importance (Significance):** How critical is this memory overall? Unlike relevance, the “importance” of a memory may be defined at generation-time.

Relying solely on vector-based relevance is a common pitfall. Similarity scores can surface memories that are conceptually similar but old or trivial. The most effective strategy is a blended approach that combines the scores from all three dimensions.

For applications where accuracy is paramount, retrieval can be refined using approaches like query rewriting, reranking, or specialized retrievers. However, these techniques are computationally expensive and add significant latency, making them unsuitable for most real-time applications. For scenarios where these complex algorithms are necessary and the memories do not quickly become stale, a caching layer can be an effective mitigation. Caching allows the expensive results of a retrieval query to be temporarily stored, bypassing the high latency cost for subsequent identical requests.

With **query rewriting**, an LLM can be used to improve the search query itself. This can involve **rewriting** a user's ambiguous input into a more precise query, or **expanding** a single query into multiple related ones to capture different facets of a topic. While this significantly improves the quality of the initial search results, it adds the latency of an extra LLM call at the start of the process.

With reranking, an initial retrieval fetches a broad set of candidate memories (e.g., the top 50 results) using similarity search. Then, an LLM can re-evaluate and re-rank this smaller set to produce a more accurate final list[24].

Finally, you can train a **specialized retriever** using fine-tuning. However, this requires access to labeled data and can significantly increase costs.

Ultimately, the best approach to retrieval starts with better memory generation. Ensuring the memory corpus is high-quality and free of irrelevant information is the most effective way to guarantee that any set of retrieved memories will be helpful.

Timing for retrieval

The final architectural decision for retrieval is *when* to retrieve memories. One approach is **proactive retrieval**, where memories are automatically loaded at the start of every turn. This ensures context is always available but introduces unnecessary latency for turns that don't require memory access. Since memories remain static throughout a single turn, they can be efficiently cached to mitigate this performance cost.

For example, you can implement proactive retrieval in ADK using the built-in `PreloadMemoryTool` or a custom callback[25]:

Python

```
# Option 1: Use the built-in PreloadMemoryTool which retrieves memories with
similarity search every turn.
agent = LlmAgent(
    ...,
    tools=[adk.tools.preload_memory_tool.PreloadMemoryTool()]
)


# Option 2: Use a custom callback to have more control over how memories
are retrieved.
def retrieve_memories_callback(callback_context, llm_request):
    user_id = callback_context._invocation_context.user_id
    app_name = callback_context._invocation_context.app_name


    response = client.agent_engines.memories.retrieve(
        name="projects/.../locations/...reasoningEngines/...",
        scope={
            "user_id": user_id,
            "app_name": app_name
        }
    )
    memories = [f"* {memory.memory.fact}" for memory in list(response)]
    if not memories:
        # No memories to add to System Instructions.
        return
    # Append formatted memories to the System Instructions
    llm_request.config.system_instruction += "\nHere is information that you have
about the user:\n"
    llm_request.config.system_instruction += "\n".join(memories)
    agent = LlmAgent(
       ...,
       before_model_callback=retrieve_memories_callback,
    )
```

Snippet 10: Retrieve memories at the start of every turn with ADK using a built-in tool or custom callback

Alternatively, you can use **reactive retrieval ("Memory-as-a-Tool")** where the agent is given a tool to query its memory, deciding for itself when to retrieve context. This is more efficient and robust but requires an additional LLM call, increasing latency and cost; however, memory is retrieved only when necessary, so the latency cost is incurred less frequently. Additionally, the agent may not know if relevant information exists to be retrieved. However, this can be mitigated by making the agent aware of the types of memories available (e.g., in the tool's description if you're using a custom tool), allowing for a more informed decision on when to query.

Python

```
# Option 1: Use the built-in LoadMemory.
agent = LlmAgent(
    ...,
    tools=[adk.tools.load_memory_tool.LoadMemoryTool()],
)


# Option 2: Use a Custom tool where you can describe what type of information
# might be available.
def load_memory(query: str, tool_context: ToolContext):
    """Retrieves memories for the user.


    The following types of information may be stored for the user:
    * User preferences, like the user's favorite foods.
    ...
    """
    # Retrieve memories using similarity search.
    response = tool_context.search_memory(query)
    return response.memories


agent = LlmAgent(
    ...,
    tools=[load_memory],
)
```

Snippet 11: Configure your ADK agent to decide when memories should be retrieved using a built-in or custom tool

Inference with Memories

Once relevant memories have been retrieved, the final step is to strategically place them into the model's context window. This is a critical process; the placement of memories can significantly influence the LLM's reasoning, affect operational costs, and ultimately determine the quality of the final answer.

Memories are primarily presented by appending them to system instructions or injecting them into conversation history. In practice, a hybrid strategy is often the most effective. Use the **system prompt** for stable, global memories (like a user profile) that should always be present. Otherwise, use **dialogue injection** or **memory-as-a-tool** for transient, episodic memories that are only relevant to the immediate context of the conversation. This balances the need for persistent context with the flexibility of in-the-moment information retrieval.

Memories in the System Instructions

A simple option to use memories for inference is to append memories to the system instructions. This method keeps the conversation history clean by appending retrieved memories directly to the system prompt alongside a preamble, framing them as foundational context for the entire interaction. For example, you can use Jinja to dynamically add memories to your system instructions:

Python

```
from jinja2 import Template

template = Template("""
{{ system_instructions }}}

<MEMORIES>
Here is some information about the user:
{% for retrieved_memory in data %}* {{ retrieved_memory.memory.fact }}
{% endfor %}</MEMORIES>
""")

prompt = template.render(
    system_instructions=system_instructions,
    data=retrieved_memories
)
```

Snippet 12: Build your system instruction using retrieved memories

Including memories in the system instructions gives memories high authority, cleanly separates context from dialogue, and is ideal for stable, "global" information like a user profile. However, there is a risk of **over-influence**, where the agent might try to relate every topic back to the memories in its core instructions, even when inappropriate.

This architectural pattern introduces several constraints. First, it requires the agent framework to support dynamic construction of the system prompt before each LLM call; this functionality isn't always readily supported. Additionally, the pattern is incompatible with *"Memory-as-a-Tool"* given that the system prompt must be finalized before the LLM can decide to call a memory retrieval tool. Finally, it poorly handles non-textual memories. Most LLMs only accept a text for the system instructions, making it challenging to embed multimodal content like images or audio directly into the prompt.

Memories in the Conversation History

In this approach, retrieved memories are injected directly into the turn-by-turn dialogue. Memories can either be placed before the full conversation history or right before the latest user query.

However, this method can be noisy, increasing token costs and potentially confusing the model if the retrieved memories are irrelevant. Its primary risk is **dialogue injection**, where the model might mistakenly treat a memory as something that was actually said in the conversation. You also need to be more careful about the perspective of the memories that you're injecting into the conversation; for example, if you're using the "user" role and user-level memories, memories should be written in first-person point of view.

A special case of injecting memories into the conversation history is retrieving memories via tool calls. The memories will be included directly in the conversation as part of the tool output.

Python

```
def load_memory(query: str, tool_context: ToolContext):
    """Loads memories into the conversation history..."""
    response = tool_context.search_memory(query)
    return response.memories


agent = LlmAgent(
    ...,
    tools=[load_memory],
)
```

Snippet 13: Retrieve memories as a tool, which directly inserts memories into the conversation

Procedural memories

This whitepaper has focused primarily on declarative memories, a concentration that mirrors the current commercial memory landscape. Most memory management platforms are also architected for this declarative approach, excelling at extracting, storing, and retrieving the "what"—facts, history, and user data.

However, these systems are not designed to manage procedural memories, the mechanism for improving an agent's workflows and reasoning. Storing the "how" is not an information retrieval problem; it is a reasoning augmentation problem. Managing this "knowing how" requires a completely separate and specialized algorithmic lifecycle, albeit with a similar high-level structure[26]:

1. **Extraction:** Procedural extraction requires specialized prompts designed to distill a reusable *strategy* or "playbook" from a successful interaction, rather than just capturing a fact or meaningful information.
2. **Consolidation:** While declarative consolidation merges related facts (the "what"), procedural consolidation curates the workflow itself (the "how"). This is an active logic management process focused on integrating new successful methods with existing "best practices," patching flawed steps in a known plan, and pruning outdated or ineffective procedures.
3. **Retrieval:** The goal is not to retrieve data to answer a question, but to retrieve a plan that guides the agent on how to execute a complex task. Therefore, procedural memories may have a different data schema than declarative memories.

This capacity for an agent to 'self-evolve' its logic naturally invites a comparison to a common adaptation method: fine-tuning—often via Reinforcement Learning from Human Feedback (RLHF)[27]. While both processes aim to improve agent behavior, their mechanisms

and applications are fundamentally different. Fine-tuning is a relatively slow, offline training process that alters model weights. Procedural memory provides a fast, online adaptation by dynamically injecting the correct "playbook" into the prompt, guiding the agent via in-context learning without requiring any fine-tuning.

Testing and Evaluation

Now that you have a memory-enabled agent, you should validate the behavior of your memory-enabled agent via comprehensive quality and evaluation tests. Evaluating an agent's memory is a multi-layered process. Evaluation requires verifying that the agent is remembering the right things (quality), that it can find those memories when needed (retrieval), and that using those memories actually helps it accomplish its goals (task success). While academia focuses on reproducible benchmarks, industry evaluation is centered on how memory directly impacts the performance and usability of a production agent.

Memory generation quality metrics evaluate the content of the memories themselves, answering the question: **"Is the agent remembering the right things?"** This is typically measured by comparing the agent's generated memories against a manually created "golden set" of ideal memories.

- **Precision:** Of all the memories the agent created, what percentage are accurate and relevant? High precision guards against an "over-eager" memory system that pollutes the knowledge base with irrelevant noise.
- **Recall:** Of all the relevant facts it should have remembered from the source, what percentage did it capture? High recall ensures the agent doesn't miss critical information.
- **F1-Score:** The harmonic mean of precision and recall, providing a single, balanced measure of quality.

Memory retrieval performance metrics evaluate the agent's ability to find the right memory at the right time.

- **Recall@K:** When a memory is needed, is the correct one found within the top 'K' retrieved results? This is the primary measure of a retrieval system's accuracy.
- **Latency:** Retrieval is on the "hot path" of an agent's response. The entire retrieval process must execute within a strict latency budget (e.g., under 200ms) to avoid degrading the user experience.

End-to-End task success metrics are the ultimate test, answering the question: "Does memory actually help the agent perform its job better?" This is measured by evaluating the agent's performance on downstream tasks using its memory, often with an LLM "judge" comparing the agent's final output to a golden answer. The judge determines if the agent's answer was accurate, effectively measuring how well the memory system contributed to the final outcome.

Evaluation is not a one-time event; it’s an engine for continuous improvement. The metrics above provide the data needed to identify weaknesses and systematically enhance the memory system over time. This iterative process involves establishing a baseline, analyzing failures, tuning the system (e.g., refining prompts, adjusting retrieval algorithms), and re-evaluating to measure the impact of the changes.

While the metrics above focus on quality, production-readiness also depends on performance. For each evaluation area, it is critical to measure the latency of underlying algorithms and their ability to scale under load. Retrieving memories “on the hot-path” may have a strict, sub-second latency budget. Generation and consolidation, while often asynchronous, must have enough throughput to keep up with user demand. Ultimately, a successful memory system must be intelligent, efficient, and robust for real-world use.

Production considerations for Memory

In addition to performance, transitioning a memory-enabled agent from prototype to production demands a focus on enterprise-grade architectural concerns. This move introduces critical requirements for scalability, resilience, and security. A production-grade system must be designed not only for intelligence but also for enterprise-level robustness.

To ensure the user experience is never blocked by the computationally expensive process of memory generation, a robust architecture must decouple memory processing from the main application logic. While this is an event-driven pattern, it is typically implemented via direct, non-blocking API calls to a dedicated memory service rather than a self-managed message queue. The flow looks like this:

1. **Agent pushes data:** After a relevant event (e.g., a session ends), the agent application makes a non-blocking API call to the memory manager, "pushing" the raw source data (like the conversation transcript) to be processed.
2. **Memory manager processes in the background:** The memory manager service immediately acknowledges the request and places the generation task into its own internal, managed queue. It is then solely responsible for the asynchronous heavy lifting: making the necessary LLM calls to extract, consolidate, and format memories. The manager may delay processing the events until a certain period of inactivity elapses.
3. **Memories are persisted:** The service writes the final memories—which may be new entries or updates to existing ones—to a dedicated, durable database. For managed memory managers, the storage is built-in.
4. **Agent retrieves memories:** The main agent application can then query this memory store directly when it needs to retrieve context for a new user interaction.

This service-based, non-blocking approach ensures that failures or latency in the memory pipeline do not directly impact the user-facing application, making the system far more resilient. It also informs the choice between **online** (real-time) generation, which is ideal for conversational freshness, and **offline** (batch) processing, which is useful for populating the system from historical data.

As an application grows, the memory system must handle high-frequency events without failure. Given **concurrent** requests, the system must prevent deadlocks or race conditions when multiple events try to modify the same memory. You can mitigate race conditions using transactional database operations or optimistic locking; however, this can introduce **queuing** or **throttling** when multiple requests are trying to modify the same memories. A robust message queue is essential to buffer high volumes of events and prevent the memory generation service from being overwhelmed.

The memory service must also be resilient to transient errors (**failure handling**). If an LLM call fails, the system should use a retry mechanism with exponential backoff and route persistent failures to a dead-letter queue for analysis.

For global applications, the memory manager must use a database with built-in **multi-region replication** to ensure low latency and high availability. Client-side replication is not feasible because consolidation requires a single, transactionally consistent view of the data to prevent conflicts. Therefore, the memory system must handle replication internally, presenting a single, logical datastore to the developer while ensuring the underlying knowledge base is globally consistent.

Managed memory systems, like Agent Engine Memory Bank, should help you address these production considerations, so that you can focus on the core agent logic.

Privacy and security risks

Memories are derived from and include user data, so they require stringent privacy and security controls. A useful analogy is to think of the system's memory as a secure corporate archive managed by a professional archivist, whose job is to preserve valuable knowledge while protecting the company.

The cardinal rule for this archive is data isolation. Just as an archivist would never mix confidential files from different departments, memory must be strictly isolated at the user or tenant level. An agent serving one user must never have access to the memories of another, enforced using restrictive Access Control Lists (ACLs). Furthermore, users must have programmatic control over their data, with clear options to opt-out of memory generation or request the deletion of all their files from the archive.

Before filing any document, the archivist performs critical security steps. First, they meticulously go through each page to redact sensitive personal information (PII), ensuring knowledge is saved without creating a liability. Second, the archivist is trained to spot and discard forgeries or intentionally misleading documents—a safeguard against memory poisoning[28]. In the same way, the system must validate and sanitize information before committing it to long-term memory to prevent a malicious user from corrupting the agent's persistent knowledge through prompt injection. The system must include safeguards like Model Armor to validate and sanitize information before committing it to long-term memory[29].

Additionally, there is an exfiltration risk if multiple users share the same set of memories, like with procedural memories (which teach an agent how to do something). For example, if a procedural memory from one user is used as an example for another—like sharing a memo company-wide—the archivist must first perform rigorous anonymization to prevent sensitive information from leaking across user boundaries.

Conclusion

This whitepaper has explored the discipline of **Context Engineering**, focusing on its two central components: **Sessions** and **Memory**. The journey from a simple conversational turn to a piece of persistent, actionable intelligence is governed by this practice, which involves dynamically assembling all necessary information—including conversation history, memories, and external knowledge—into the LLM's context window. This entire process relies on the interplay between two distinct but interconnected systems: the immediate Session and the long-term Memory.

The **Session** governs the "now," acting as a low-latency, chronological container for a single conversation. Its primary challenge is performance and security, requiring **low-latency access** and **strict isolation**. To prevent context window overflow and latency, you must use **extraction** techniques like token-based truncation or recursive summarization to **compact** content *within* the Session's history or a single request payload. Furthermore, security is paramount, mandating **PII redaction** before session data is persisted.

Memory is the engine of **long-term personalization** and the core mechanism for persistence across multiple sessions. It moves beyond RAG (which makes an agent an expert on *facts*) to make the agent an expert on the *user*. Memory is an active, LLM-driven ETL pipeline—responsible for **extraction, consolidation, and retrieval**—that distills the most important information from conversation history. With **extraction**, the system distills the most critical information into key memory points. Following this, **consolidation** curates and integrates this new information with the existing corpus, resolving conflicts, and deleting redundant data to ensure a coherent knowledge base. To maintain a snappy user experience, memory generation must run as an **asynchronous background process** after the agent has responded. By tracking **provenance** and employing safeguards against risks like memory poisoning, developers can build trusted, adaptive assistants that truly learn and grow with the user.

Endnotes

1. https://cloud.google.com/use-cases/retrieval-augmented-generation?hl=en
2. https://arxiv.org/abs/2301.00234
3. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/sessions/overview
4. https://langchain-ai.github.io/langgraph/concepts/multi_agent/#message-passing-between-agents
5. https://google.github.io/adk-docs/agents/multi-agents/
6. https://google.github.io/adk-docs/agents/multi-agents/#c-explicit-invocation-agenttool
7. https://agent2agent.info/docs/concepts/message/
8. https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
9. https://cloud.google.com/security-command-center/docs/model-armor-overview
10. https://ai.google.dev/gemini-api/docs/long-context#long-context-limitations
11. https://huggingface.co/blog/Kseniase/memory
12. https://langchain-ai.github.io/langgraph/concepts/memory/#semantic-memory
13. https://langchain-ai.github.io/langgraph/concepts/memory/#semantic-memory
14. https://arxiv.org/pdf/2412.15266
15. https://arxiv.org/pdf/2412.15266
16. https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/inference#sample-requests-text-gen-multimodal-prompt
17. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/memory-bank/generate-memories
18. https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/control-generated-output
19. https://cloud.google.com/agent-builder/agent-engine/memory-bank/set-up#memory-bank-config
20. https://arxiv.org/html/2504.19413v1
21. https://google.github.io/adk-docs/tools/#how-agents-use-tools

22. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/memory-bank/generate-memories#consolidate-pre-extracted-memories

23. https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/memory-bank/generate-memories#background-memory-generation

24. https://arxiv.org/pdf/2503.08026

25. https://google.github.io/adk-docs/callbacks/

26. https://arxiv.org/html/2508.06433v2

27. https://cloud.google.com/blog/products/ai-machine-learning/rlhf-on-google-cloud

28. https://arxiv.org/pdf/2503.03704

29. https://cloud.google.com/security-command-center/docs/model-armor-overview

30. https://cloud.google.com/architecture/choose-design-pattern-agentic-ai-system