# Introduction to Conversational Context: Session, State, and Memory[¶](https://google.github.io/adk-docs/sessions/#introduction-to-conversational-context-session-state-and-memory)

## Why Context Matters[¶](https://google.github.io/adk-docs/sessions/#why-context-matters)

Meaningful, multi-turn conversations require agents to understand context. Just like humans, they need to recall the conversation history: what's been said and done to maintain continuity and avoid repetition. The Agent Development Kit (ADK) provides structured ways to manage this context through Session, State, and Memory.

## Core Concepts[¶](https://google.github.io/adk-docs/sessions/#core-concepts)

Think of different instances of your conversations with the agent as distinct **conversation threads**, potentially drawing upon **long-term knowledge**.

1. **Session**: The Current Conversation Thread
   * Represents a *single, ongoing interaction* between a user and your agent system.
   * Contains the chronological sequence of messages and actions taken by the agent (referred to Events) during *that specific interaction*.
   * A Session can also hold temporary data (State) relevant only *during this conversation*.
2. **State (session.state)**: Data Within the Current Conversation
   * Data stored within a specific Session.
   * Used to manage information relevant *only* to the *current, active* conversation thread (e.g., items in a shopping cart *during this chat*, user preferences mentioned *in this session*).
3. **Memory**: Searchable, Cross-Session Information
   * Represents a store of information that might span *multiple past sessions* or include external data sources.
   * It acts as a knowledge base the agent can *search* to recall information or context beyond the immediate conversation.

## Managing Context: Services[¶](https://google.github.io/adk-docs/sessions/#managing-context-services)

ADK provides services to manage these concepts:

1. **SessionService**: Manages the different conversation threads (Session objects)
   * Handles the lifecycle: creating, retrieving, updating (appending Events, modifying State), and deleting individual Sessions.
2. **MemoryService**: Manages the Long-Term Knowledge Store (Memory)
   * Handles ingesting information (often from completed Sessions) into the long-term store.
   * Provides methods to search this stored knowledge based on queries.

**Implementations**: ADK offers different implementations for both SessionService and MemoryService, allowing you to choose the storage backend that best fits your application's needs. Notably, **in-memory implementations** are provided for both services; these are designed specifically for **local testing and fast development**. It's important to remember that **all data stored using these in-memory options (sessions, state, or long-term knowledge) is lost when your application restarts**. For persistence and scalability beyond local testing, ADK also offers cloud-based and database service options.

**In Summary:**

* **Session & State**: Focus on the **current interaction** – the history and data of the *single, active conversation*. Managed primarily by a SessionService.
* **Memory**: Focuses on the **past and external information** – a *searchable archive* potentially spanning across conversations. Managed by a MemoryService.

## What's Next?[¶](https://google.github.io/adk-docs/sessions/#whats-next)

In the following sections, we'll dive deeper into each of these components:

* **Session**: Understanding its structure and Events.
* **State**: How to effectively read, write, and manage session-specific data.
* **SessionService**: Choosing the right storage backend for your sessions.
* **MemoryService**: Exploring options for storing and retrieving broader context.

Understanding these concepts is fundamental to building agents that can engage in complex, stateful, and context-aware conversations.

# Session: Tracking Individual Conversations[¶](https://google.github.io/adk-docs/sessions/session/#session-tracking-individual-conversations)

Following our Introduction, let's dive into the Session. Think back to the idea of a "conversation thread." Just like you wouldn't start every text message from scratch, agents need context regarding the ongoing interaction. **Session** is the ADK object designed specifically to track and manage these individual conversation threads.

## The Session Object[¶](https://google.github.io/adk-docs/sessions/session/#the-session-object)

When a user starts interacting with your agent, the SessionService creates a Session object (google.adk.sessions.Session). This object acts as the container holding everything related to that *one specific chat thread*. Here are its key properties:

* **Identification (id, appName, userId):** Unique labels for the conversation.
  + id: A unique identifier for *this specific* conversation thread, essential for retrieving it later.
  + appName: Identifies which agent application this conversation belongs to.
  + userId: Links the conversation to a particular user.
* **History (events):** A chronological sequence of all interactions (Event objects – user messages, agent responses, tool actions) that have occurred within this specific thread.
* **Session State (state):** A place to store temporary data relevant *only* to this specific, ongoing conversation. This acts as a scratchpad for the agent during the interaction. We will cover how to use and manage state in detail in the next section.
* **Activity Tracking (lastUpdateTime):** A timestamp indicating the last time an event occurred in this conversation thread.

### Example: Examining Session Properties[¶](https://google.github.io/adk-docs/sessions/session/#example-examining-session-properties)

from google.adk.sessions import InMemorySessionService, Session

# Create a simple session to examine its properties

temp\_service = InMemorySessionService()

example\_session = await temp\_service.create\_session(

app\_name="my\_app",

user\_id="example\_user",

state={"initial\_key": "initial\_value"} # State can be initialized

)

print(f"--- Examining Session Properties ---")

print(f"ID (`id`): {example\_session.id}")

print(f"Application Name (`app\_name`): {example\_session.app\_name}")

print(f"User ID (`user\_id`): {example\_session.user\_id}")

print(f"State (`state`): {example\_session.state}") # Note: Only shows initial state here

print(f"Events (`events`): {example\_session.events}") # Initially empty

print(f"Last Update (`last\_update\_time`): {example\_session.last\_update\_time:.2f}")

print(f"---------------------------------")

# Clean up (optional for this example)

temp\_service = await temp\_service.delete\_session(app\_name=example\_session.app\_name,

user\_id=example\_session.user\_id, session\_id=example\_session.id)

print("The final status of temp\_service - ", temp\_service)

*(Note: The state shown above is only the initial state. State updates happen via events, as discussed in the State section.)*

## Managing Sessions with a SessionService[¶](https://google.github.io/adk-docs/sessions/session/#managing-sessions-with-a-sessionservice)

As seen above, you don't typically create or manage Session objects directly. Instead, you use a **SessionService**. This service acts as the central manager responsible for the entire lifecycle of your conversation sessions.

Its core responsibilities include:

* **Starting New Conversations:** Creating fresh Session objects when a user begins an interaction.
* **Resuming Existing Conversations:** Retrieving a specific Session (using its ID) so the agent can continue where it left off.
* **Saving Progress:** Appending new interactions (Event objects) to a session's history. This is also the mechanism through which session state gets updated (more in the State section).
* **Listing Conversations:** Finding the active session threads for a particular user and application.
* **Cleaning Up:** Deleting Session objects and their associated data when conversations are finished or no longer needed.

## SessionService Implementations[¶](https://google.github.io/adk-docs/sessions/session/#sessionservice-implementations)

ADK provides different SessionService implementations, allowing you to choose the storage backend that best suits your needs:

1. **InMemorySessionService**
   * **How it works:** Stores all session data directly in the application's memory.
   * **Persistence:** None. **All conversation data is lost if the application restarts.**
   * **Requires:** Nothing extra.
   * **Best for:** Quick development, local testing, examples, and scenarios where long-term persistence isn't required.
2. **[Python](https://google.github.io/adk-docs/sessions/session/#python_1)**
3. **[Java](https://google.github.io/adk-docs/sessions/session/#java_1)**

from google.adk.sessions import InMemorySessionService

session\_service = InMemorySessionService()

1. **VertexAiSessionService**
   * **How it works:** Uses Google Cloud's Vertex AI infrastructure via API calls for session management.
   * **Persistence:** Yes. Data is managed reliably and scalably via [Vertex AI Agent Engine](https://google.github.io/adk-docs/deploy/agent-engine/).
   * **Requires:**
     + A Google Cloud project (pip install vertexai)
     + A Google Cloud storage bucket that can be configured by this [step](https://cloud.google.com/vertex-ai/docs/pipelines/configure-project#storage).
     + A Reasoning Engine resource name/ID that can setup following this [tutorial](https://google.github.io/adk-docs/deploy/agent-engine/).
   * **Best for:** Scalable production applications deployed on Google Cloud, especially when integrating with other Vertex AI features.

# Requires: pip install google-adk[vertexai]

# Plus GCP setup and authentication

from google.adk.sessions import VertexAiSessionService

PROJECT\_ID = "your-gcp-project-id"

LOCATION = "us-central1"

# The app\_name used with this service should be the Reasoning Engine ID or name

REASONING\_ENGINE\_APP\_NAME = "projects/your-gcp-project-id/locations/us-central1/reasoningEngines/your-engine-id"

session\_service = VertexAiSessionService(project=PROJECT\_ID, location=LOCATION)

# Use REASONING\_ENGINE\_APP\_NAME when calling service methods, e.g.:

# session\_service = await session\_service.create\_session(app\_name=REASONING\_ENGINE\_APP\_NAME, ...)

1. **DatabaseSessionService  
   python_only**
   * **How it works:** Connects to a relational database (e.g., PostgreSQL, MySQL, SQLite) to store session data persistently in tables.
   * **Persistence:** Yes. Data survives application restarts.
   * **Requires:** A configured database.
   * **Best for:** Applications needing reliable, persistent storage that you manage yourself.

from google.adk.sessions import DatabaseSessionService

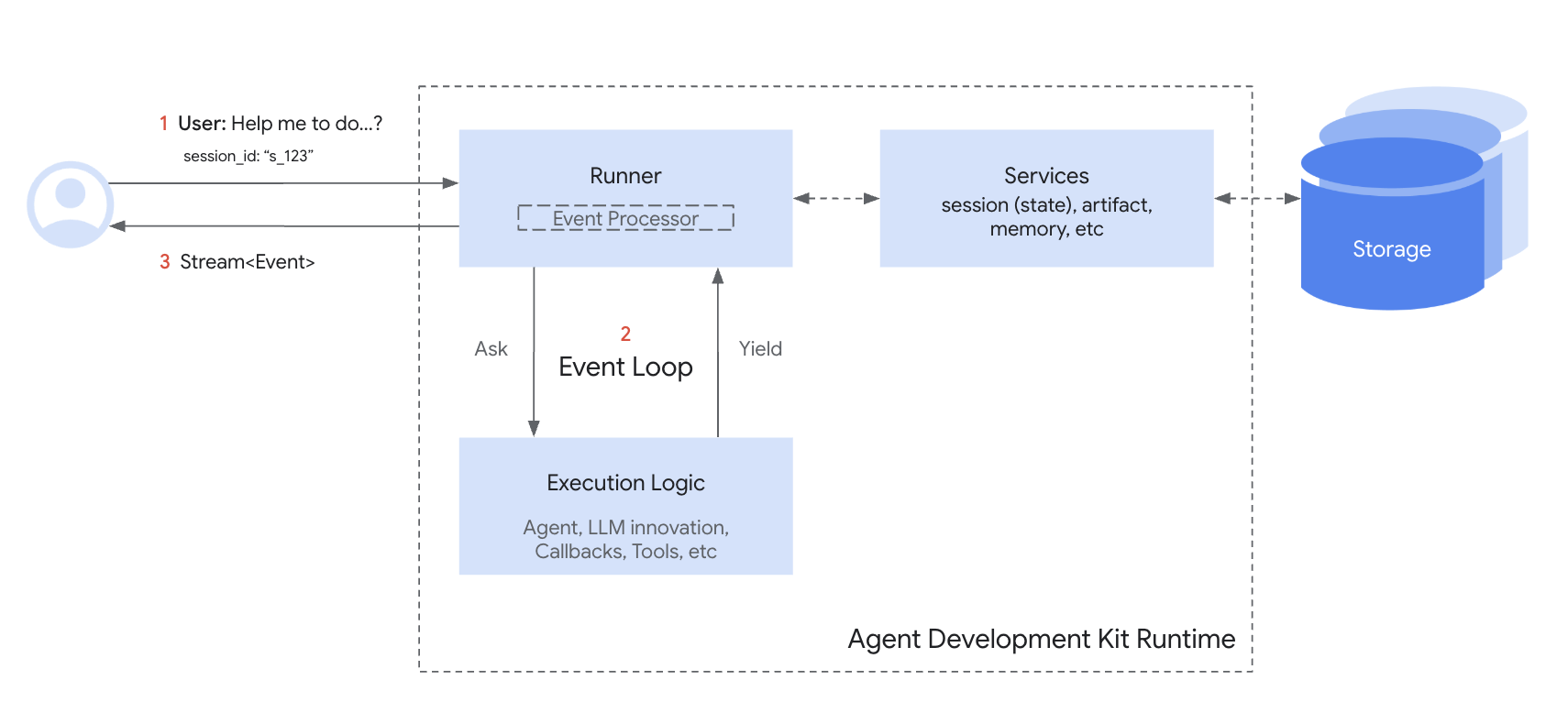
# Example using a local SQLite file:

db\_url = "sqlite:///./my\_agent\_data.db"

session\_service = DatabaseSessionService(db\_url=db\_url)

Choosing the right SessionService is key to defining how your agent's conversation history and temporary data are stored and persist.

## The Session Lifecycle[¶](https://google.github.io/adk-docs/sessions/session/#the-session-lifecycle)



Here’s a simplified flow of how Session and SessionService work together during a conversation turn:

1. **Start or Resume:** Your application's Runner uses the SessionService to either create\_session (for a new chat) or get\_session (to retrieve an existing one).
2. **Context Provided:** The Runner gets the appropriate Session object from the appropriate service method, providing the agent with access to the corresponding Session's state and events.
3. **Agent Processing:** The user prompts the agent with a query. The agent analyzes the query and potentially the session state and events history to determine the response.
4. **Response & State Update:** The agent generates a response (and potentially flags data to be updated in the state). The Runner packages this as an Event.
5. **Save Interaction:** The Runner calls sessionService.append\_event(session, event) with the session and the new event as the arguments. The service adds the Event to the history and updates the session's state in storage based on information within the event. The session's last\_update\_time also get updated.
6. **Ready for Next:** The agent's response goes to the user. The updated Session is now stored by the SessionService, ready for the next turn (which restarts the cycle at step 1, usually with the continuation of the conversation in the current session).
7. **End Conversation:** When the conversation is over, your application calls sessionService.delete\_session(...) to clean up the stored session data if it is no longer required.

This cycle highlights how the SessionService ensures conversational continuity by managing the history and state associated with each Session object.

# State: The Session's Scratchpad[¶](https://google.github.io/adk-docs/sessions/state/#state-the-sessions-scratchpad)

Within each Session (our conversation thread), the **state** attribute acts like the agent's dedicated scratchpad for that specific interaction. While session.events holds the full history, session.state is where the agent stores and updates dynamic details needed *during* the conversation.

## What is session.state?[¶](https://google.github.io/adk-docs/sessions/state/#what-is-sessionstate)

Conceptually, session.state is a collection (dictionary or Map) holding key-value pairs. It's designed for information the agent needs to recall or track to make the current conversation effective:

* **Personalize Interaction:** Remember user preferences mentioned earlier (e.g., 'user\_preference\_theme': 'dark').
* **Track Task Progress:** Keep tabs on steps in a multi-turn process (e.g., 'booking\_step': 'confirm\_payment').
* **Accumulate Information:** Build lists or summaries (e.g., 'shopping\_cart\_items': ['book', 'pen']).
* **Make Informed Decisions:** Store flags or values influencing the next response (e.g., 'user\_is\_authenticated': True).

### Key Characteristics of State[¶](https://google.github.io/adk-docs/sessions/state/#key-characteristics-of-state)

1. **Structure: Serializable Key-Value Pairs**
   * Data is stored as key: value.
   * **Keys:** Always strings (str). Use clear names (e.g., 'departure\_city', 'user:language\_preference').
   * **Values:** Must be **serializable**. This means they can be easily saved and loaded by the SessionService. Stick to basic types in the specific languages (Python/ Java) like strings, numbers, booleans, and simple lists or dictionaries containing *only* these basic types. (See API documentation for precise details).
   * **⚠️ Avoid Complex Objects:** **Do not store non-serializable objects** (custom class instances, functions, connections, etc.) directly in the state. Store simple identifiers if needed, and retrieve the complex object elsewhere.
2. **Mutability: It Changes**
   * The contents of the state are expected to change as the conversation evolves.
3. **Persistence: Depends on SessionService**
   * Whether state survives application restarts depends on your chosen service:
   * InMemorySessionService: **Not Persistent.** State is lost on restart.
   * DatabaseSessionService / VertexAiSessionService: **Persistent.** State is saved reliably.

**Note**

The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., session.state['current\_intent'] = 'book\_flight' in Python, session.state().put("current\_intent", "book\_flight) in Java). Refer to the language-specific API documentation for details.

### Organizing State with Prefixes: Scope Matters[¶](https://google.github.io/adk-docs/sessions/state/#organizing-state-with-prefixes-scope-matters)

Prefixes on state keys define their scope and persistence behavior, especially with persistent services:

* **No Prefix (Session State):**
  + **Scope:** Specific to the *current* session (id).
  + **Persistence:** Only persists if the SessionService is persistent (Database, VertexAI).
  + **Use Cases:** Tracking progress within the current task (e.g., 'current\_booking\_step'), temporary flags for this interaction (e.g., 'needs\_clarification').
  + **Example:** session.state['current\_intent'] = 'book\_flight'
* **user: Prefix (User State):**
  + **Scope:** Tied to the user\_id, shared across *all* sessions for that user (within the same app\_name).
  + **Persistence:** Persistent with Database or VertexAI. (Stored by InMemory but lost on restart).
  + **Use Cases:** User preferences (e.g., 'user:theme'), profile details (e.g., 'user:name').
  + **Example:** session.state['user:preferred\_language'] = 'fr'
* **app: Prefix (App State):**
  + **Scope:** Tied to the app\_name, shared across *all* users and sessions for that application.
  + **Persistence:** Persistent with Database or VertexAI. (Stored by InMemory but lost on restart).
  + **Use Cases:** Global settings (e.g., 'app:api\_endpoint'), shared templates.
  + **Example:** session.state['app:global\_discount\_code'] = 'SAVE10'
* **temp: Prefix (Temporary Session State):**
  + **Scope:** Specific to the *current* session processing turn.
  + **Persistence:** **Never Persistent.** Guaranteed to be discarded, even with persistent services.
  + **Use Cases:** Intermediate results needed only immediately, data you explicitly don't want stored.
  + **Example:** session.state['temp:raw\_api\_response'] = {...}

**How the Agent Sees It:** Your agent code interacts with the *combined* state through the single session.state collection (dict/ Map). The SessionService handles fetching/merging state from the correct underlying storage based on prefixes.

### How State is Updated: Recommended Methods[¶](https://google.github.io/adk-docs/sessions/state/#how-state-is-updated-recommended-methods)

State should **always** be updated as part of adding an Event to the session history using session\_service.append\_event(). This ensures changes are tracked, persistence works correctly, and updates are thread-safe.

**1. The Easy Way: output\_key (for Agent Text Responses)**

This is the simplest method for saving an agent's final text response directly into the state. When defining your LlmAgent, specify the output\_key:

from google.adk.agents import LlmAgent

from google.adk.sessions import InMemorySessionService, Session

from google.adk.runners import Runner

from google.genai.types import Content, Part

# Define agent with output\_key

greeting\_agent = LlmAgent(

name="Greeter",

model="gemini-2.0-flash", # Use a valid model

instruction="Generate a short, friendly greeting.",

output\_key="last\_greeting" # Save response to state['last\_greeting']

)

# --- Setup Runner and Session ---

app\_name, user\_id, session\_id = "state\_app", "user1", "session1"

session\_service = InMemorySessionService()

runner = Runner(

agent=greeting\_agent,

app\_name=app\_name,

session\_service=session\_service

)

session = await session\_service.create\_session(app\_name=app\_name,

user\_id=user\_id,

session\_id=session\_id)

print(f"Initial state: {session.state}")

# --- Run the Agent ---

# Runner handles calling append\_event, which uses the output\_key

# to automatically create the state\_delta.

user\_message = Content(parts=[Part(text="Hello")])

for event in runner.run(user\_id=user\_id,

session\_id=session\_id,

new\_message=user\_message):

if event.is\_final\_response():

print(f"Agent responded.") # Response text is also in event.content

# --- Check Updated State ---

updated\_session = await session\_service.get\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=session\_id)

print(f"State after agent run: {updated\_session.state}")

# Expected output might include: {'last\_greeting': 'Hello there! How can I help you today?'}

Behind the scenes, the Runner uses the output\_key to create the necessary EventActions with a state\_delta and calls append\_event.

**2. The Standard Way: EventActions.state\_delta (for Complex Updates)**

For more complex scenarios (updating multiple keys, non-string values, specific scopes like user: or app:, or updates not tied directly to the agent's final text), you manually construct the state\_delta within EventActions.

from google.adk.sessions import InMemorySessionService, Session

from google.adk.events import Event, EventActions

from google.genai.types import Part, Content

import time

# --- Setup ---

session\_service = InMemorySessionService()

app\_name, user\_id, session\_id = "state\_app\_manual", "user2", "session2"

session = await session\_service.create\_session(

app\_name=app\_name,

user\_id=user\_id,

session\_id=session\_id,

state={"user:login\_count": 0, "task\_status": "idle"}

)

print(f"Initial state: {session.state}")

# --- Define State Changes ---

current\_time = time.time()

state\_changes = {

"task\_status": "active", # Update session state

"user:login\_count": session.state.get("user:login\_count", 0) + 1, # Update user state

"user:last\_login\_ts": current\_time, # Add user state

"temp:validation\_needed": True # Add temporary state (will be discarded)

}

# --- Create Event with Actions ---

actions\_with\_update = EventActions(state\_delta=state\_changes)

# This event might represent an internal system action, not just an agent response

system\_event = Event(

invocation\_id="inv\_login\_update",

author="system", # Or 'agent', 'tool' etc.

actions=actions\_with\_update,

timestamp=current\_time

# content might be None or represent the action taken

)

# --- Append the Event (This updates the state) ---

await session\_service.append\_event(session, system\_event)

print("`append\_event` called with explicit state delta.")

# --- Check Updated State ---

updated\_session = await session\_service.get\_session(app\_name=app\_name,

user\_id=user\_id,

session\_id=session\_id)

print(f"State after event: {updated\_session.state}")

# Expected: {'user:login\_count': 1, 'task\_status': 'active', 'user:last\_login\_ts': <timestamp>}

# Note: 'temp:validation\_needed' is NOT present.

**What append\_event Does:**

* Adds the Event to session.events.
* Reads the state\_delta from the event's actions.
* Applies these changes to the state managed by the SessionService, correctly handling prefixes and persistence based on the service type.
* Updates the session's last\_update\_time.
* Ensures thread-safety for concurrent updates.

### ⚠️ A Warning About Direct State Modification[¶](https://google.github.io/adk-docs/sessions/state/#a-warning-about-direct-state-modification)

Avoid directly modifying the session.state dictionary after retrieving a session (e.g., retrieved\_session.state['key'] = value).

**Why this is strongly discouraged:**

1. **Bypasses Event History:** The change isn't recorded as an Event, losing auditability.
2. **Breaks Persistence:** Changes made this way **will likely NOT be saved** by DatabaseSessionService or VertexAiSessionService. They rely on append\_event to trigger saving.
3. **Not Thread-Safe:** Can lead to race conditions and lost updates.
4. **Ignores Timestamps/Logic:** Doesn't update last\_update\_time or trigger related event logic.

**Recommendation:** Stick to updating state via output\_key or EventActions.state\_delta within the append\_event flow for reliable, trackable, and persistent state management. Use direct access only for *reading* state.

### Best Practices for State Design Recap[¶](https://google.github.io/adk-docs/sessions/state/#best-practices-for-state-design-recap)

* **Minimalism:** Store only essential, dynamic data.
* **Serialization:** Use basic, serializable types.
* **Descriptive Keys & Prefixes:** Use clear names and appropriate prefixes (user:, app:, temp:, or none).
* **Shallow Structures:** Avoid deep nesting where possible.
* **Standard Update Flow:** Rely on append\_event.

# Memory: Long-Term Knowledge with MemoryService[¶](https://google.github.io/adk-docs/sessions/memory/#memory-long-term-knowledge-with-memoryservice)

python_only

We've seen how Session tracks the history (events) and temporary data (state) for a *single, ongoing conversation*. But what if an agent needs to recall information from *past* conversations or access external knowledge bases? This is where the concept of **Long-Term Knowledge** and the **MemoryService** come into play.

Think of it this way:

* **Session / State:** Like your short-term memory during one specific chat.
* **Long-Term Knowledge (MemoryService)**: Like a searchable archive or knowledge library the agent can consult, potentially containing information from many past chats or other sources.

## The MemoryService Role[¶](https://google.github.io/adk-docs/sessions/memory/#the-memoryservice-role)

The BaseMemoryService defines the interface for managing this searchable, long-term knowledge store. Its primary responsibilities are:

1. **Ingesting Information (add\_session\_to\_memory):** Taking the contents of a (usually completed) Session and adding relevant information to the long-term knowledge store.
2. **Searching Information (search\_memory):** Allowing an agent (typically via a Tool) to query the knowledge store and retrieve relevant snippets or context based on a search query.

## MemoryService Implementations[¶](https://google.github.io/adk-docs/sessions/memory/#memoryservice-implementations)

ADK provides different ways to implement this long-term knowledge store:

1. **InMemoryMemoryService**
   * **How it works:** Stores session information in the application's memory and performs basic keyword matching for searches.
   * **Persistence:** None. **All stored knowledge is lost if the application restarts.**
   * **Requires:** Nothing extra.
   * **Best for:** Prototyping, simple testing, scenarios where only basic keyword recall is needed and persistence isn't required.

from google.adk.memory import InMemoryMemoryService

memory\_service = InMemoryMemoryService()

1. **VertexAiRagMemoryService**
   * **How it works:** Leverages Google Cloud's Vertex AI RAG (Retrieval-Augmented Generation) service. It ingests session data into a specified RAG Corpus and uses powerful semantic search capabilities for retrieval.
   * **Persistence:** Yes. The knowledge is stored persistently within the configured Vertex AI RAG Corpus.
   * **Requires:** A Google Cloud project, appropriate permissions, necessary SDKs (pip install google-adk[vertexai]), and a pre-configured Vertex AI RAG Corpus resource name/ID.
   * **Best for:** Production applications needing scalable, persistent, and semantically relevant knowledge retrieval, especially when deployed on Google Cloud.

# Requires: pip install google-adk[vertexai]

# Plus GCP setup, RAG Corpus, and authentication

from google.adk.memory import VertexAiRagMemoryService

# The RAG Corpus name or ID

RAG\_CORPUS\_RESOURCE\_NAME = "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id"

# Optional configuration for retrieval

SIMILARITY\_TOP\_K = 5

VECTOR\_DISTANCE\_THRESHOLD = 0.7

memory\_service = VertexAiRagMemoryService(

rag\_corpus=RAG\_CORPUS\_RESOURCE\_NAME,

similarity\_top\_k=SIMILARITY\_TOP\_K,

vector\_distance\_threshold=VECTOR\_DISTANCE\_THRESHOLD

)

## How Memory Works in Practice[¶](https://google.github.io/adk-docs/sessions/memory/#how-memory-works-in-practice)

The typical workflow involves these steps:

1. **Session Interaction:** A user interacts with an agent via a Session, managed by a SessionService. Events are added, and state might be updated.
2. **Ingestion into Memory:** At some point (often when a session is considered complete or has yielded significant information), your application calls memory\_service.add\_session\_to\_memory(session). This extracts relevant information from the session's events and adds it to the long-term knowledge store (in-memory dictionary or RAG Corpus).
3. **Later Query:** In a *different* (or the same) session, the user might ask a question requiring past context (e.g., "What did we discuss about project X last week?").
4. **Agent Uses Memory Tool:** An agent equipped with a memory-retrieval tool (like the built-in load\_memory tool) recognizes the need for past context. It calls the tool, providing a search query (e.g., "discussion project X last week").
5. **Search Execution:** The tool internally calls memory\_service.search\_memory(app\_name, user\_id, query).
6. **Results Returned:** The MemoryService searches its store (using keyword matching or semantic search) and returns relevant snippets as a SearchMemoryResponse containing a list of MemoryResult objects (each potentially holding events from a relevant past session).
7. **Agent Uses Results:** The tool returns these results to the agent, usually as part of the context or function response. The agent can then use this retrieved information to formulate its final answer to the user.

## Example: Adding and Searching Memory[¶](https://google.github.io/adk-docs/sessions/memory/#example-adding-and-searching-memory)

This example demonstrates the basic flow using the InMemory services for simplicity.

import asyncio

from google.adk.agents import LlmAgent

from google.adk.sessions import InMemorySessionService, Session

from google.adk.memory import InMemoryMemoryService # Import MemoryService

from google.adk.runners import Runner

from google.adk.tools import load\_memory # Tool to query memory

from google.genai.types import Content, Part

# --- Constants ---

APP\_NAME = "memory\_example\_app"

USER\_ID = "mem\_user"

MODEL = "gemini-2.0-flash" # Use a valid model

# --- Agent Definitions ---

# Agent 1: Simple agent to capture information

info\_capture\_agent = LlmAgent(

model=MODEL,

name="InfoCaptureAgent",

instruction="Acknowledge the user's statement.",

# output\_key="captured\_info" # Could optionally save to state too

)

# Agent 2: Agent that can use memory

memory\_recall\_agent = LlmAgent(

model=MODEL,

name="MemoryRecallAgent",

instruction="Answer the user's question. Use the 'load\_memory' tool "

"if the answer might be in past conversations.",

tools=[load\_memory] # Give the agent the tool

)

# --- Services and Runner ---

session\_service = InMemorySessionService()

memory\_service = InMemoryMemoryService() # Use in-memory for demo

runner = Runner(

# Start with the info capture agent

agent=info\_capture\_agent,

app\_name=APP\_NAME,

session\_service=session\_service,

memory\_service=memory\_service # Provide the memory service to the Runner

)

# --- Scenario ---

# Turn 1: Capture some information in a session

print("--- Turn 1: Capturing Information ---")

session1\_id = "session\_info"

session1 = await runner.session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=session1\_id)

user\_input1 = Content(parts=[Part(text="My favorite project is Project Alpha.")], role="user")

# Run the agent

final\_response\_text = "(No final response)"

async for event in runner.run\_async(user\_id=USER\_ID, session\_id=session1\_id, new\_message=user\_input1):

if event.is\_final\_response() and event.content and event.content.parts:

final\_response\_text = event.content.parts[0].text

print(f"Agent 1 Response: {final\_response\_text}")

# Get the completed session

completed\_session1 = await runner.session\_service.get\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=session1\_id)

# Add this session's content to the Memory Service

print("\n--- Adding Session 1 to Memory ---")

memory\_service = await memory\_service.add\_session\_to\_memory(completed\_session1)

print("Session added to memory.")

# Turn 2: In a \*new\* (or same) session, ask a question requiring memory

print("\n--- Turn 2: Recalling Information ---")

session2\_id = "session\_recall" # Can be same or different session ID

session2 = await runner.session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=session2\_id)

# Switch runner to the recall agent

runner.agent = memory\_recall\_agent

user\_input2 = Content(parts=[Part(text="What is my favorite project?")], role="user")

# Run the recall agent

print("Running MemoryRecallAgent...")

final\_response\_text\_2 = "(No final response)"

async for event in runner.run\_async(user\_id=USER\_ID, session\_id=session2\_id, new\_message=user\_input2):

print(f" Event: {event.author} - Type: {'Text' if event.content and event.content.parts and event.content.parts[0].text else ''}"

f"{'FuncCall' if event.get\_function\_calls() else ''}"

f"{'FuncResp' if event.get\_function\_responses() else ''}")

if event.is\_final\_response() and event.content and event.content.parts:

final\_response\_text\_2 = event.content.parts[0].text

print(f"Agent 2 Final Response: {final\_response\_text\_2}")

break # Stop after final response

# Expected Event Sequence for Turn 2:

# 1. User sends "What is my favorite project?"

# 2. Agent (LLM) decides to call `load\_memory` tool with a query like "favorite project".

# 3. Runner executes the `load\_memory` tool, which calls `memory\_service.search\_memory`.

# 4. `InMemoryMemoryService` finds the relevant text ("My favorite project is Project Alpha.") from session1.

# 5. Tool returns this text in a FunctionResponse event.

# 6. Agent (LLM) receives the function response, processes the retrieved text.

# 7. Agent generates the final answer (e.g., "Your favorite project is Project Alpha.").