Memory: Long-Term Knowledge with `MemoryService`

Currently supported in Python

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

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

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()
```

2. `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

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

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

Full Code

```
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.").
```