+ Code + Markdown

```
[ ]:
# @title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

🧠 Memory Management - Part 2 - Memory

Welcome to Day 3 of the Kaggle 5-day Agents course!

In the previous notebook, you learned how **Sessions** manage conversation threads. Now you'll add **Memory** - a searchable, long-term knowledge store that persists across multiple conversations.

What is Memory ❓

Memory is a service that provides long-term knowledge storage for your agents. The key distinction:

> **Session = Short-term memory** (single conversation)
>
> **Memory = Long-term knowledge** (across multiple conversations)

Think of it in software engineering terms: **Session** is like application state (temporary), while **Memory** is like a database (persistent).

+ Code + Markdown

🤔 Why Memory?

Memory provides capabilities that Sessions alone cannot:

Capability	What It Means	Example
Cross-Conversation Recall	Access information from any past conversation	"What preferences has this user mentioned across all chats?"
Intelligent Extraction	LLM-powered consolidation extracts key facts	Stores "allergic to peanuts" instead of 50 raw messages
Semantic Search	Meaning-based retrieval, not just keyword matching	Query "preferred hue" matches "favorite color is blue"
Persistent Storage	Survives application restarts	Build knowledge that grows over time

Example: Imagine talking to a personal assistant:

- 🟣 **Session**: They remember what you said 10 minutes ago in THIS conversation
- 🧠 **Memory**: They remember your preferences from conversations LAST WEEK

🎯 What you'll learn:

- ✅ Initialize MemoryService and integrate with your agent
- ✅ Transfer session data to memory storage
- ✅ Search and retrieve memories
- ✅ Automate memory storage and retrieval
- ✅ Understand memory consolidation (conceptual overview)

📝 Implementation Note

> This notebook uses `InMemoryMemoryService` for learning - it performs keyword matching and doesn't persist data.
>
> For production applications, use **Vertex AI Memory Bank** (covered in Day 5), which provides LLM-powered consolidation and semantic search with persistent cloud storage.

‼️ Please Read

> ❌ ℹ️ **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

> ℹ️ **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads **"Waiting for the next available notebook"**. The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

> ❌ **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.
> See FAQ on 429 errors for more information.

For help: Ask questions on the Kaggle Discord server.

📖 Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks in the documentation.

Here's how to get started:

1. Verify Your Account (Required)

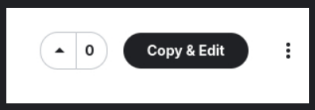
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your Kaggle settings.

2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy and Edit` button in the top-right corner.

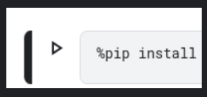


This creates a private copy of the notebook just for you.

3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶️ Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

4. If You Get Stuck

To restart: Select `Factory reset` from the `Run` menu.

For help: Ask questions on the Kaggle Discord server.

⚙️ Section 1: Setup

1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the google-adk library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

1.2: Configure your Gemini API Key

This notebook uses the Gemini API, which requires authentication.

1. Get your API key

If you don't have one already, create an API key in Google AI Studio.

2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select `Add-ons` then `Secrets`.
2. Create a new secret with the label `GOOGLE_API_KEY`.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to `GOOGLE_API_KEY` is selected so that the secret is attached to the notebook.

3. Authenticate in the notebook

Run the cell below to complete authentication.

[1]:
```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Gemini API key setup complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

```
✅ Gemini API key setup complete.
```

1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

[2]:
```
from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.memory import InMemoryMemoryService
from google.adk.tools import load_memory, preload_memory
from google.genai import types

print("✅ ADK components imported successfully.")
```

```
✅ ADK components imported successfully.
```

1.4: Helper functions

This helper function manages a complete conversation session, handling session creation/retrieval, query processing, and response streaming.

[3]:
```
async def run_session(
    runner_instance: Runner, user_queries: list[str] | str, session_id: str = "default"
):
    """Helper function to run queries in a session and display responses."""
    print(f"\n### Session: {session_id}")

    # Create or retrieve session
    try:
        session = await session_service.create_session(
            app_name=APP_NAME, user_id=USER_ID, session_id=session_id
        )
    except:
        session = await session_service.get_session(
            app_name=APP_NAME, user_id=USER_ID, session_id=session_id
        )

    # Convert single query to list
    if isinstance(user_queries, str):
        user_queries = [user_queries]

    # Process each query
    for query in user_queries:
        print(f"\nUser > {query}")
        query_content = types.Content(role="user", parts=[types.Part(text=query)])

        # Stream agent response
        async for event in runner_instance.run_async(
            user_id=USER_ID, session_id=session.id, new_message=query_content
        ):
            if event.is_final_response() and event.content and event.content.parts:
                text = event.content.parts[0].text
                if text and text != "None":
                    print(f"Model: > {text}")


print("✅ Helper functions defined.")
```

```
✅ Helper functions defined.
```

1.5: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

[5]:
```
retry_config = types.HttpRetryOptions(
    attempts=5,  # Maximum retry attempts
    exp_base=7,  # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],  # Retry on these HTTP errors
)
```

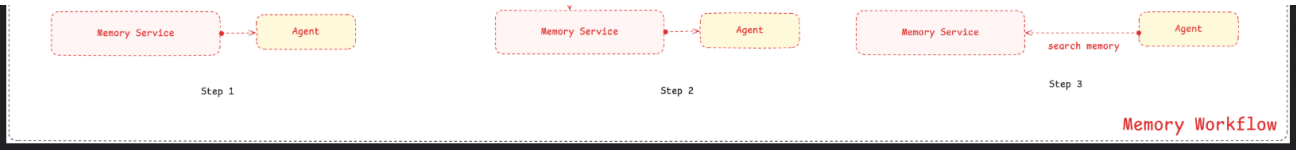
🤓 Section 2: Memory Workflow

From the Introduction section, you now know why we need Memory. In order to integrate Memory into your Agents, there are **three high-level steps.**

Three-step integration process:

1. **Initialize** → Create a `MemoryService` and provide it to your agent via the `Runner`
2. **Ingest** → Transfer session data to memory using `add_session_to_memory()`
3. **Retrieve** → Search stored memories using `search_memory()`

Let's explore each step in the following sections.



🧠 Section 3: Initialize MemoryService

3.1 Initialize Memory

ADK provides multiple `MemoryService` implementations through the `BaseMemoryService` interface:

- **`InMemoryMemoryService`** - Built-in service for prototyping and testing (keyword matching, no persistence)
- **`VertexAiMemoryBankService`** - Managed cloud service with LLM-powered consolidation and semantic search
- **Custom implementations** - You can build your own using databases, though managed services are recommended

For this notebook, we'll use `InMemoryMemoryService` to learn the core mechanics. The same methods work identically with production-ready services like Vertex AI Memory Bank.

[6]:
```
memory_service = (
    InMemoryMemoryService()
)  # ADK's built-in Memory Service for development and testing
```

3.2 Add Memory to Agent

Next, create a simple agent to answer user queries.

[7]:
```
# Define constants used throughout the notebook
APP_NAME = "MemoryDemoApp"
USER_ID = "demo_user"

# Create agent
user_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="MemoryDemoAgent",
    instruction="Answer user questions in simple words.",
)

print("✅ Agent created")
```

```
✅ Agent created
```

Create Runner

Now provide both Session and Memory services to the `Runner`.

Key configuration:

The `Runner` requires both services to enable memory functionality:

- **`session_service`** → Manages conversation threads and events
- **`memory_service`** → Provides long-term knowledge storage

Both services work together: Sessions capture conversations, Memory stores knowledge for retrieval across sessions.

[8]:
```
# Create Session Service
session_service = InMemorySessionService()  # Handles conversations

# Create runner with BOTH services
runner = Runner(
    agent=user_agent,
    app_name="MemoryDemoApp",
    session_service=session_service,
    memory_service=memory_service,  # Memory service is now available!
)

print("✅ Agent and Runner created with memory support!")
```

```
✅ Agent and Runner created with memory support!
```

‼️ Important

💡 **Configuration vs. Usage:** Adding `memory_service` to the `Runner` makes memory *available* to your agent, but doesn't automatically use it. You must explicitly:

1. **Ingest data** using `add_session_to_memory()`
2. **Enable retrieval** by giving your agent memory tools (`load_memory` or `preload_memory`)

Let's learn these steps next!

3.3 MemoryService Implementation Options

This notebook: `InMemoryMemoryService`

- Stores raw conversation events without consolidation
- Keyword-based search (simple word matching)
- In-memory storage (resets on restart)
- Ideal for learning and local development

Production: `VertexAiMemoryBankService` **(You'll learn this on Day 5)**

- LLM-powered extraction of key facts
- Semantic search (meaning-based retrieval)
- Persistent cloud storage
- Integrates external knowledge sources

💡 **API Consistency:** Both implementations use identical methods (`add_session_to_memory()` , `search_memory()`). The workflow you learn here applies to all memory services!

💾 Section 4: Ingest Session Data into Memory

Why should you transfer Session data to Memory?

Now that memory is initialized, you need to populate it with knowledge. When you initialize a MemoryService, it starts completely empty. All your conversations are stored in Sessions, which contain raw events including every message, tool call, and metadata. To make this information available for long-term recall, you explicitly transfer it to memory using `add_session_to_memory()` .

Here's where managed memory services like Vertex AI Memory Bank shine. **During transfer, they perform intelligent consolidation - extracting key facts while discarding conversational noise.** The `InMemoryMemoryService` we're using stores everything without consolidation, which is sufficient for learning the mechanics.

Before we can transfer anything, we need data. Let's have a conversation with our agent to populate the session. This conversation will be stored in the SessionService just like you learned in the previous notebook.

[9]:
```
# User tells agent about their favorite color
await run_session(
    runner,
    "My favorite color is blue-green. Can you write a Haiku about it?",
    "conversation-01",  # Session ID
)
```

```
### Session: conversation-01

User > My favorite color is blue-green. Can you write a Haiku about it?
Model: > A tranquil ocean,
Deep forest in summer light,
My favorite hue.
```

Let's verify the conversation was captured in the session. You should see the session events containing both the user's prompt and the model's response.

[10]:
```
session = await session_service.get_session(
    app_name=APP_NAME, user_id=USER_ID, session_id="conversation-01"
)

# Let's see what's in the session
print("📝 Session contains:")
for event in session.events:
    text = (
        event.content.parts[0].text[:60]
        if event.content and event.content.parts
        else "(empty)"
    )
    print(f"  {event.content.role}: {text}...")
```

```
📝 Session contains:
  user: My favorite color is blue-green. Can you write a Haiku about...
  model: A tranquil ocean,
Deep forest in summer light,
My favorite h...
```

Perfect! The session contains our conversation. Now we're ready to transfer it to memory. Call `add_session_to_memory()` and pass the session object. This ingests the conversation into the memory store, making it available for future searches.

[11]:
```
# This is the key method!
await memory_service.add_session_to_memory(session)

print("✅ Session added to memory!")
```

```
✅ Session added to memory!
```

🔎 Section 5: Enable Memory Retrieval in Your Agent

You've successfully transferred session data to memory, but there's one crucial step remaining. **Agents can't directly access the MemoryService - they need tools to search it.**

This is by design: it gives you control over when and how memory is retrieved.

5.1 Memory Retrieval in ADK

ADK provides two built-in tools for memory retrieval:

`load_memory` (Reactive)

- Agent decides when to search memory
- Only retrieves when the agent thinks it's needed
- More efficient (saves tokens)
- Risk: Agent might forget to search

`preload_memory` (Proactive)

- Automatically searches before every turn
- Memory always available to the agent
- Guaranteed context, but less efficient
- Searches even when not needed

Think of it like studying for an exam: `load_memory` is looking things up only when you need them, while `preload_memory` is reading all your notes before answering each question.

5.2 Add Load Memory Tool to Agent

Let's start by implementing the reactive pattern. We'll recreate the agent from Section 3, this time adding the `load_memory` tool to its toolkit. Since this is a built-in ADK tool, you simply include it in the tools array without any custom implementation.

[12]:
```
# Create agent
user_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="MemoryDemoAgent",
    instruction="Answer user questions in simple words. Use load_memory tool if you need to recall past conversations.",
    tools=[
        load_memory
    ],  # Agent now has access to Memory and can search it whenever it decides to!
)

print("✅ Agent with load_memory tool created.")
```

```
✅ Agent with load_memory tool created.
```

5.3 Update the Runner and Test

Let's now update the Runner to use our new `user_agent` that has the `load_memory` tool. And we'll ask the Agent about the favorite color which we had stored previously in another session.

👉 **Since sessions don't share conversation history, the only way the agent can answer correctly is by using the `load_memory` tool** to retrieve the information from long-term memory that we manually stored.

[13]:
```
# Create a new runner with the updated agent
runner = Runner(
    agent=user_agent,
    app_name=APP_NAME,
    session_service=session_service,
    memory_service=memory_service,
)

await run_session(runner, "What is my favorite color?", "color-test")
```

```
### Session: color-test

User > What is my favorite color?
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
Model: > The user's favorite color is blue-green.
```

5.4 Complete Manual Workflow Test

Let's see the complete workflow in action. We'll have a conversation about a birthday, manually save it to memory, then test retrieval in a new session. This demonstrates the full cycle: **ingest → store → retrieve**.

[14]:
```
await run_session(runner, "My birthday is on March 15th.", "birthday-session-01")
```

```
### Session: birthday-session-01

User > My birthday is on March 15th.
Model: > Thanks for letting me know! March 15th it is.
```

Now manually save this session to memory. This is the crucial step that transfers the conversation from short-term session storage to long-term memory storage.

[15]:
```
# Manually save the session to memory
birthday_session = await session_service.get_session(
    app_name=APP_NAME, user_id=USER_ID, session_id="birthday-session-01"
```

```
)

await memory_service.add_session_to_memory(birthday_session)

print("✅ Birthday session saved to memory!")
```

```
✅ Birthday session saved to memory!
```

Here's the crucial test: we'll start a completely new session with a different session ID and ask the agent to recall the birthday.

[16]:
```
# Test retrieval in a NEW session
await run_session(
    runner, "When is my birthday?", "birthday-session-02"  # Different session ID
)
```

```
### Session: birthday-session-02

User > When is my birthday?
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
Model: > Mar 15th.
```

What happens:

1. Agent receives: "When is my birthday?"
2. Agent recognizes: This requires past conversation context
3. Agent calls: `load_memory("birthday")`
4. Memory returns: Previous conversation containing "March 15th"
5. Agent responds: "Your birthday is on March 15th"

The memory retrieval worked even though this is a completely different session!

🚀 Your Turn: Experiment with Both Patterns

Try swapping `load_memory` with `preload_memory` by changing the tools array to `tools=[preload_memory]`.

What changes:

- `load_memory` (reactive): Agent decides when to search
- `preload_memory` (proactive): Automatically loads memory before every turn

Test it:

1. Ask "What is my favorite color?" in a new session
2. Ask "Tell me a joke" - notice that `preload_memory` still searches memory even though it's unnecessary
3. Which pattern is better for different use cases?

5.5 Manual Memory Search

Beyond agent tools, you can also search memories directly in your code. This is useful for:

- Debugging memory contents
- Building analytics dashboards
- Creating custom memory management UIs

The `search_memory()` method takes a text query and returns a `SearchMemoryResponse` with matching memories.

[17]:
```
# Search for color preferences
search_response = await memory_service.search_memory(
    app_name=APP_NAME, user_id=USER_ID, query="What is the user's favorite color?"
)

print("🔍 Search Results:")
print(f"  Found {len(search_response.memories)} relevant memories")
print()

for memory in search_response.memories:
    if memory.content and memory.content.parts:
        text = memory.content.parts[0].text[:80]
        print(f"  [{memory.author}]: {text}...")
```

```
🔍 Search Results:
  Found 4 relevant memories

  [user]: My favorite color is blue-green. Can you write a Haiku about it?...
  [MemoryDemoAgent]: A tranquil ocean,
Deep forest in summer light,
My favorite hue....
  [user]: My birthday is on March 15th....
  [MemoryDemoAgent]: Thanks for letting me know! March 15th it is....
```

🚀 Your Turn: Test Different Queries

Try these searches to understand how keyword matching works with `InMemoryMemoryService`:

1. **"what color does the user like"**
2. **"haiku"**
3. **"age"**
4. **"preferred hue"**

4. **preferred hue**

Notice which queries return results and which don't. What pattern do you observe?

💡 **Key Insight:** Memory search is grounded in reality - agents can't hallucinate memories that don't exist.

5.6 How Search Works

InMemoryMemoryService (this notebook):

- **Method:** Keyword matching
- **Example:** "favorite color" matches because those exact words exist
- **Limitation:** "preferred hue" won't match

VertexAiMemoryBankService (Day 5):

- **Method:** Semantic search via embeddings
- **Example:** "preferred hue" WILL match "favorite color"
- **Advantage:** Understands meaning, not just keywords

You'll explore semantic search in Day 5!

🤖 Section 6: Automating Memory Storage

So far, we've **manually** called `add_session_to_memory()` to transfer data to long-term storage. Production systems need this to happen **automatically**.

6.1 Callbacks

ADK's callback system lets you hook into key execution moments. Callbacks are **Python functions** you define and attach to agents - ADK automatically calls them at specific stages, acting like checkpoints during the agent's execution flow.

Think of callbacks as event listeners in your agent's lifecycle. When an agent processes a request, it goes through multiple stages: receiving the input, calling the LLM, invoking tools, and generating the response. Callbacks let you insert custom logic at each of these stages without modifying the core agent code.

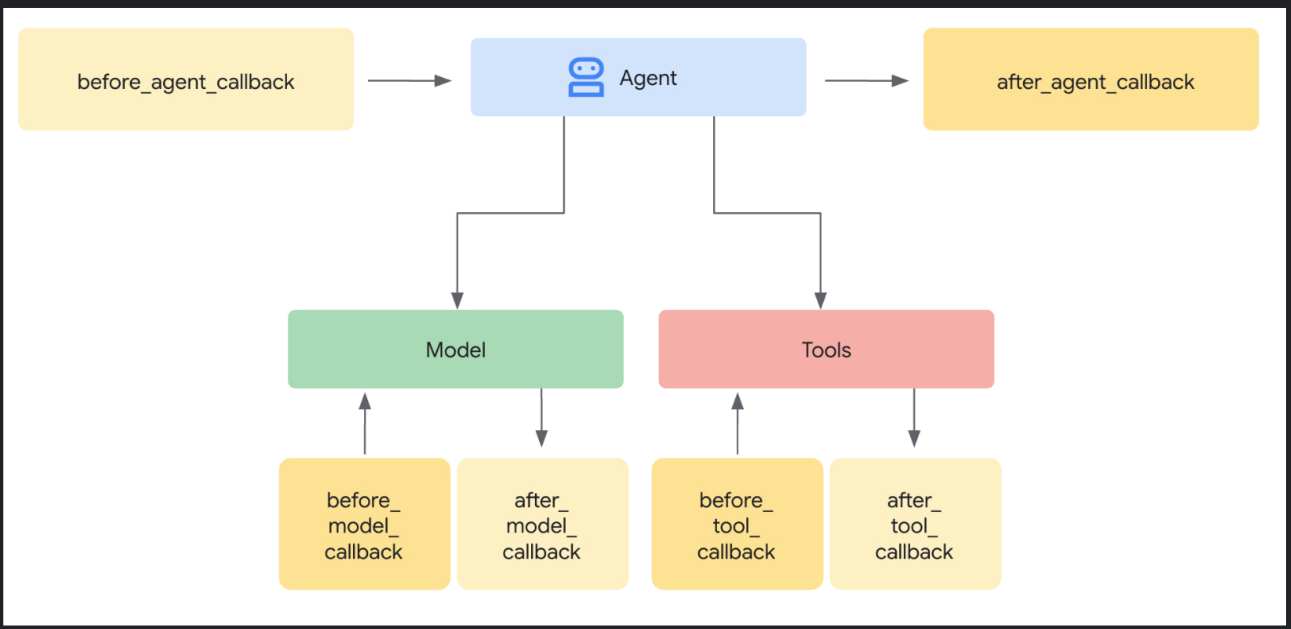
Available callback types:

- `before_agent_callback` → Runs before agent starts processing a request
- `after_agent_callback` → Runs after agent completes its turn
- `before_tool_callback` / `after_tool_callback` → Around tool invocations
- `before_model_callback` / `after_model_callback` → Around LLM calls
- `on_model_error_callback` → When errors occur

Common use cases:

- Logging and observability (track what the agent does)
- Automatic data persistence (like saving to memory)
- Custom validation or filtering
- Performance monitoring

📚 **Learn More:** ADK Callbacks Documentation



6.2 Automatic Memory Storage with Callbacks

For automatic memory storage, we'll use `after_agent_callback`. This function triggers every time the agent finishes a turn, then calls `add_session_to_memory()` to persist the conversation automatically.

But here's the challenge: how does our callback function actually access the memory service and current session? That's where `callback_context` comes in.

When you define a callback function, ADK automatically passes a special parameter called `callback_context` to it. The `callback_context` provides access to the Memory Service and other runtime components.

How we'll use it: In our callback, we'll access the memory service and current session to automatically save conversation data after each turn.

💡 **Important:** You don't create this context - ADK creates it and passes it to your callback automatically when the callback runs.

[18]:
```
async def auto_save_to_memory(callback_context):
    """Automatically save session to memory after each agent turn."""
    await callback_context._invocation_context.memory_service.add_session_to_memory(
        callback_context._invocation_context.session
    )


print("✅ Callback created.")
```

```
✅ Callback created.
```

6.3 Create an Agent: Callback and PreLoad Memory Tool

Now create an agent that combines:

- **Automatic storage:** `after_agent_callback` saves conversations
- **Automatic retrieval:** `preload_memory` loads memories

This creates a fully automated memory system with zero manual intervention.

[19]:
```
# Agent with automatic memory saving
auto_memory_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="AutoMemoryAgent",
    instruction="Answer user questions.",
    tools=[preload_memory],
    after_agent_callback=auto_save_to_memory,  # Saves after each turn!
)

print("✅ Agent created with automatic memory saving!")
```

```
✅ Agent created with automatic memory saving!
```

What happens automatically:

- After every agent response → callback triggers
- Session data → transferred to memory
- No manual `add_session_to_memory()` calls needed

The framework handles everything!

6.4 Create a Runner and Test The Agent

Time to test! Create a Runner with the auto-memory agent, connecting the session and memory services.

[20]:
```
# Create a runner for the auto-save agent
# This connects our automated agent to the session and memory services
auto_runner = Runner(
    agent=auto_memory_agent,  # Use the agent with callback + preload_memory
    app_name=APP_NAME,
    session_service=session_service,  # Same services from Section 3
    memory_service=memory_service,
)

print("✅ Runner created.")
```

```
✅ Runner created.
```

[21]:
```
# Test 1: Tell the agent about a gift (first conversation)
# The callback will automatically save this to memory when the turn completes
await run_session(
    auto_runner,
    "I gifted a new toy to my nephew on his 1st birthday!",
    "auto-save-test",
)

# Test 2: Ask about the gift in a NEW session (second conversation)
# The agent should retrieve the memory using preload_memory and answer correctly
await run_session(
    auto_runner,
    "What did I gift my nephew?",
    "auto-save-test-2",  # Different session ID - proves memory works across sessions!
)
```

```
### Session: auto-save-test

User > I gifted a new toy to my nephew on his 1st birthday!
Model: > Happy birthday to your nephew! That's a lovely gift to give him on his special day.
```

```
### Session: auto-save-test-2

User > What did I gift my nephew?
Model: > You gifted your nephew a new toy!
```

What just happened:

1. **First conversation:** Mentioned gift to nephew
   - Callback automatically saved to memory ✅
2. **Second conversation (new session):** Asked about the gift
   - `preload_memory` automatically retrieved the memory ✅
   - Agent answered correctly ✅

Zero manual memory calls! This is automated memory management in action.

6.5 How often should you save Sessions to Memory?

Options:

Timing	Implementation	Best For
After every turn	`after_agent_callback`	Real-time memory updates
End of conversation	Manual call when session ends	Batch processing, reduce API calls
Periodic intervals	Timer-based background job	Long-running conversations

---

🧩 Section 7: Memory Consolidation

7.1 The Limitation of Raw Storage

What we've stored so far:

- Every user message
- Every agent response
- Every tool call

The problem:

```
Session: 50 messages = 10,000 tokens
Memory:  All 50 messages stored
Search:  Returns all 50 messages → Agent must process 10,000 tokens
```

This doesn't scale. We need **consolidation**.

7.2 What is Memory Consolidation?

Memory Consolidation = Extracting **only important facts** while discarding conversational noise.

Before (Raw Storage):

```
User: "My favorite color is BlueGreen. I also like purple.
       Actually, I prefer BlueGreen most of the time."
Agent: "Great! I'll remember that."
User: "Thanks!"
Agent: "You're welcome!"

→ Stores ALL 4 messages (redundant, verbose)
```

After (Consolidation):

```
Extracted Memory: "User's favorite color: BlueGreen"

→ Stores 1 concise fact
```

Benefits: Less storage, faster retrieval, more accurate answers.

```
Session(id='784', app_name='example_app', user_id='2', state={},
events=[Event(content=Content(
  parts=[
    Part(
      text='My favorite project is Project Alpha.'
    ),
  ],
  role='user'
), grounding_metadata=None, partial=None, turn_complete=None,
error_code=None, error_message=None, interrupted=None,
custom_metadata=None, usage_metadata=None, invocation_id='e-d9dd373a-ee26-
4606-9e22-c435d59debd2', author='user',
actions=EventActions(skip_summarization=None, state_delta={},
artifact_delta={}, transfer_to_agent=None, escalate=None,
requested_auth_configs={}), long_running_tool_ids=None, branch=None,
id='0cf8c777-bfd3-4206-8805-33a2c6622e7f', timestamp=1752594914.926269),
Event(content=Content(
  parts=[
    Part(
      text="""Okay, I understand. Your favorite project is Project Alpha.
"""
    ),
  ],
  role='model'
), grounding_metadata=None, partial=None, turn_complete=None,
error_code=None, error_message=None, interrupted=None,
custom_metadata=None, usage_metadata=GenerateContentResponseUsageMetadata(
  candidates_token_count=13,
  candidates_tokens_details=[
    ModalityTokenCount(
      modality=<MediaModality.TEXT: 'TEXT'>,
      token_count=13
    ),
  ],
  prompt_token_count=29,
  prompt_tokens_details=[
    ModalityTokenCount(
      modality=<MediaModality.TEXT: 'TEXT'>,
      token_count=29
    ),
```

```
memories=[MemoryEntry(content=Content(
  parts=[
    Part(
      text='My favorite project is Project Alpha.'
    ),
  ],
  role='user'
), author='user', timestamp='2025-07-15T15:55:14.926269'), MemoryEntry(content=Content(
  parts=[
    Part(
      text="""Okay, I understand. Your favorite project is Project Alpha.
"""
    ),
  ],
  role='model'
), author='InfoCaptureAgent', timestamp='2025-07-15T15:55:14.926806')]
```

Sample Session data

Sample Memory

7.3 How Consolidation Works (Conceptual)

The pipeline:

```
1. Raw Session Events
   ↓
2. LLM analyzes conversation
   ↓
3. Extracts key facts
   ↓
4. Stores concise memories
   ↓
5. Merges with existing memories (deduplication)
```

Example transformation:

```
Input:  "I'm allergic to peanuts. I can't eat anything with nuts."

Output: Memory {
  allergy: "peanuts, tree nuts"
  severity: "avoid completely"
}
```

Natural language → Structured, actionable data.

7.4 Next Steps for Memory Consolidation

💡 **Key Point:** Managed Memory Services handle consolidation **automatically**.

You use the same API:

- `add_session_to_memory()` ← Same method
- `search_memory()` ← Same method

The difference: What happens behind the scenes.

- **InMemoryMemoryService:** Stores raw events
- **VertexAiMemoryBankService:** Intelligently consolidates before storing

📚 **Learn More:**

- Vertex AI Memory Bank: Memory Consolidation Guide -> You'll explore this in Day 5!

📊 Summary

You've learned the **core mechanics** of Memory in ADK:

1. ✅ **Adding Memory**
   - Initialize `MemoryService` alongside `SessionService`
   - Both services are provided to the `Runner`
2. ✅ **Storing Information**
   - `await memory_service.add_session_to_memory(session)`
   - Transfers session data to long-term storage
   - Can be automated with callbacks
3. ✅ **Searching Memory**
   - `await memory_service.search_memory(app_name, user_id, query)`
   - Returns relevant memories from past conversations
4. ✅ **Retrieving in Agents**
   - **Reactive:** `load_memory` tool (agent decides when to use memory)
   - **Proactive:** `preload_memory` tool (always loads memory into LLM's system instructions)
5. ✅ **Memory Consolidation**
   - Extracts key information from Session data
   - Provided by managed memory services such as Vertex AI Memory Bank

🎉 Congratulations! You've learned Memory Management in ADK!

📚 **Learn More:**

- ADK Memory Documentation
- Vertex AI Memory Bank
- Memory Consolidation Guide

🎯 **Next Steps:**

Ready for Day 4? Learn how to **implement Observability and Evaluate your agents** to ensure they're working as intended in production!

Authors

Authors

Sampath M