**Chapter 8\_ Memory Management**

Chapter 8: Memory Management

Effective memory management is crucial for intelligent agents to retain information. Agents require different types of memory, much like humans, to operate efficiently. This chapter delves into memory management, specifically addressing the immediate (short-term) and persistent (long-term) memory requirements of agents.

In agent systems, memory refers to an agent's ability to retain and utilize information from past interactions, observations, and learning experiences. This capability allows agents to make informed decisions, maintain conversational context, and improve over time. Agent memory is generally categorized into two main types:

* **Short-Term Memory (Contextual Memory):** Similar to working memory, this holds information currently being processed or recently accessed. For agents using large language models (LLMs), short-term memory primarily exists within the context window. This window contains recent messages, agent replies, tool usage results, and agent reflections from the current interaction, all of which inform the LLM's subsequent responses and actions. The context window has a limited capacity, restricting the amount of recent information an agent can directly access. Efficient short-term memory management involves keeping the most relevant information within this limited space, possibly through techniques like summarizing older conversation segments or emphasizing key details. The advent of models with 'long context' windows simply expands the size of this short-term memory, allowing more information to be held within a single interaction. However, this context is still ephemeral and is lost once the session concludes, and it can be costly and inefficient to process every time. Consequently, agents require separate memory types to achieve true persistence, recall information from past interactions, and build a lasting knowledge base.
* **Long-Term Memory (Persistent Memory):** This acts as a repository for information agents need to retain across various interactions, tasks, or extended periods, akin to long-term knowledge bases. Data is typically stored outside the agent's immediate processing environment, often in databases, knowledge graphs, or vector databases. In vector databases, information is converted into numerical vectors and stored, enabling agents to retrieve data based on semantic similarity rather than exact keyword matches, a process known as semantic search. When an agent needs information from long-term memory, it queries the external storage, retrieves relevant data, and integrates it into the short-term context for immediate use, thus combining prior knowledge with the current interaction.

**Practical Applications & Use Cases**

Memory management is vital for agents to track information and perform intelligently over time. This is essential for agents to surpass basic question-answering capabilities. Applications include:

* **Chatbots and Conversational AI:** Maintaining conversation flow relies on short-term memory. Chatbots require remembering prior user inputs to provide coherent responses. Long-term memory enables chatbots to recall user preferences, past issues, or prior discussions, offering personalized and continuous interactions.
* **Task-Oriented Agents:** Agents managing multi-step tasks need short-term memory to track previous steps, current progress, and overall goals. This information might reside in the task's context or temporary storage. Long-term memory is crucial for accessing specific user-related data not in the immediate context.
* **Personalized Experiences:** Agents offering tailored interactions utilize long-term memory to store and retrieve user preferences, past behaviors, and personal information. This allows agents to adapt their responses and suggestions.
* **Learning and Improvement:** Agents can refine their performance by learning from past interactions. Successful strategies, mistakes, and new information are stored in long-term memory, facilitating future adaptations. Reinforcement learning agents store learned strategies or knowledge in this way.
* **Information Retrieval (RAG):** Agents designed for answering questions access a knowledge base, their long-term memory, often implemented within Retrieval Augmented Generation (RAG). The agent retrieves relevant documents or data to inform its responses.
* **Autonomous Systems:** Robots or self-driving cars require memory for maps, routes, object locations, and learned behaviors. This involves short-term memory for immediate surroundings and long-term memory for general environmental knowledge.

Memory enables agents to maintain history, learn, personalize interactions, and manage complex, time-dependent problems.

**Hands-On Code: Memory Management in Google Agent Developer Kit (ADK)**

The Google Agent Developer Kit (ADK) offers a structured method for managing context and memory, including components for practical application. A solid grasp of ADK's Session, State, and Memory is vital for building agents that need to retain information.

Just as in human interactions, agents require the ability to recall previous exchanges to conduct coherent and natural conversations. ADK simplifies context management through three core concepts and their associated services.

Every interaction with an agent can be considered a unique conversation thread. Agents might need to access data from earlier interactions. ADK structures this as follows:

* **Session:** An individual chat thread that logs messages and actions (Events) for that specific interaction, also storing temporary data (State) relevant to that conversation.
* **State (session.state):** Data stored within a Session, containing information relevant only to the current, active chat thread.
* **Memory:** A searchable repository of information sourced from various past chats or external sources, serving as a resource for data retrieval beyond the immediate conversation.

ADK provides dedicated services for managing critical components essential for building complex, stateful, and context-aware agents. The SessionService manages chat threads (Session objects) by handling their initiation, recording, and termination, while the MemoryService oversees the storage and retrieval of long-term knowledge (Memory).

Both the SessionService and MemoryService offer various configuration options, allowing users to choose storage methods based on application needs. In-memory options are available for testing purposes, though data will not persist across restarts. For persistent storage and scalability, ADK also supports database and cloud-based services.

**Session: Keeping Track of Each Chat**

A Session object in ADK is designed to track and manage individual chat threads. Upon initiation of a conversation with an agent, the SessionService generates a Session object, represented as `google.adk.sessions.Session`. This object encapsulates all data relevant to a specific conversation thread, including unique identifiers (id, app\_name, user\_id), a chronological record of events as Event objects, a storage area for session-specific temporary data known as state, and a timestamp indicating the last update (last\_update\_time). Developers typically interact with Session objects indirectly through the SessionService. The SessionService is responsible for managing the lifecycle of conversation sessions, which includes initiating new sessions, resuming previous sessions, recording session activity (including state updates), identifying active sessions, and managing the removal of session data. The ADK provides several SessionService implementations with varying storage mechanisms for session history and temporary data, such as the InMemorySessionService, which is suitable for testing but does not provide data persistence across application restarts.

|  |
| --- |
| # Example: Using InMemorySessionService  # This is suitable for local development and testing where data  # persistence across application restarts is not required.  from google.adk.sessions import InMemorySessionService  session\_service = InMemorySessionService() |

Then there's DatabaseSessionService if you want reliable saving to a database you manage.

|  |
| --- |
| # Example: Using DatabaseSessionService  # This is suitable for production or development requiring persistent storage.  # You need to configure a database URL (e.g., for SQLite, PostgreSQL, etc.).  # Requires: pip install google-adk[sqlalchemy] and a database driver (e.g., psycopg2 for PostgreSQL)  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) |

Besides, there's VertexAiSessionService which uses Vertex AI infrastructure for scalable production on Google Cloud.

|  |
| --- |
| # Example: Using VertexAiSessionService  # This is suitable for scalable production on Google Cloud Platform, leveraging  # Vertex AI infrastructure for session management.  # Requires: pip install google-adk[vertexai] and GCP setup/authentication  from google.adk.sessions import VertexAiSessionService  PROJECT\_ID = "your-gcp-project-id" # Replace with your GCP project ID  LOCATION = "us-central1" # Replace with your desired GCP location  # The app\_name used with this service should correspond to the Reasoning Engine ID or name  REASONING\_ENGINE\_APP\_NAME = "projects/your-gcp-project-id/locations/us-central1/reasoningEngines/your-engine-id" # Replace with your Reasoning Engine resource name  session\_service = VertexAiSessionService(project=PROJECT\_ID, location=LOCATION)  # When using this service, pass REASONING\_ENGINE\_APP\_NAME to service methods:  # session\_service.create\_session(app\_name=REASONING\_ENGINE\_APP\_NAME, ...)  # session\_service.get\_session(app\_name=REASONING\_ENGINE\_APP\_NAME, ...)  # session\_service.append\_event(session, event, app\_name=REASONING\_ENGINE\_APP\_NAME)  # session\_service.delete\_session(app\_name=REASONING\_ENGINE\_APP\_NAME, ...) |

Choosing an appropriate SessionService is crucial as it determines how the agent's interaction history and temporary data are stored and their persistence.

Each message exchange involves a cyclical process: A message is received, the Runner retrieves or establishes a Session using the SessionService, the agent processes the message using the Session's context (state and historical interactions), the agent generates a response and may update the state, the Runner encapsulates this as an Event, and the session\_service.append\_event method records the new event and updates the state in storage. The Session then awaits the next message. Ideally, the delete\_session method is employed to terminate the session when the interaction concludes. This process illustrates how the SessionService maintains continuity by managing the Session-specific history and temporary data.

**State: The Session's Scratchpad**

In the ADK, each Session, representing a chat thread, includes a state component akin to an agent's temporary working memory for the duration of that specific conversation. While session.events logs the entire chat history, session.state stores and updates dynamic data points relevant to the active chat.

Fundamentally, session.state operates as a dictionary, storing data as key-value pairs. Its core function is to enable the agent to retain and manage details essential for coherent dialogue, such as user preferences, task progress, incremental data collection, or conditional flags influencing subsequent agent actions.

The state’s structure comprises string keys paired with values of serializable Python types, including strings, numbers, booleans, lists, and dictionaries containing these basic types. State is dynamic, evolving throughout the conversation. The permanence of these changes depends on the configured SessionService.

State organization can be achieved using key prefixes to define data scope and persistence. Keys without prefixes are session-specific.

* The user: prefix associates data with a user ID across all sessions.
* The app: prefix designates data shared among all users of the application.
* The temp: prefix indicates data valid only for the current processing turn and is not persistently stored.

The agent accesses all state data through a single session.state dictionary. The SessionService handles data retrieval, merging, and persistence. State should be updated upon adding an Event to the session history via session\_service.append\_event(). This ensures accurate tracking, proper saving in persistent services, and safe handling of state changes.

1. **The Simple Way: Using output\_key (for Agent Text Replies):** This is the easiest method if you just want to save your agent's final text response directly into the state. When you set up your LlmAgent, just tell it the output\_key you want to use. The Runner sees this and automatically creates the necessary actions to save the response to the state when it appends the event. Let's look at a code example demonstrating state update via output\_key.

|  |
| --- |
| # Import necessary classes from the Google Agent Developer Kit (ADK)  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 an LlmAgent with an output\_key.  greeting\_agent = LlmAgent(  name="Greeter",  model="gemini-2.0-flash",  instruction="Generate a short, friendly greeting.",  output\_key="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 = 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 ---  user\_message = Content(parts=[Part(text="Hello")])  print("\n--- Running the agent ---")  for event in runner.run(  user\_id=user\_id,  session\_id=session\_id,  new\_message=user\_message  ):  if event.is\_final\_response():  print("Agent responded.")  # --- Check Updated State ---  # Correctly check the state \*after\* the runner has finished processing all events.  updated\_session = session\_service.get\_session(app\_name, user\_id, session\_id)  print(f"\nState after agent run: {updated\_session.state}") |

Behind the scenes, the Runner sees your output\_key and automatically creates the necessary actions with a state\_delta when it calls append\_event.

1. **The Standard Way: Using EventActions.state\_delta (for More Complicated Updates):** For times when you need to do more complex things – like updating several keys at once, saving things that aren't just text, targeting specific scopes like user: or app:, or making updates that aren't tied to the agent's final text reply – you'll manually build a dictionary of your state changes (the state\_delta) and include it within the EventActions of the Event you're appending. Let's look at one example:

|  |
| --- |
| import time  from google.adk.tools.tool\_context import ToolContext  from google.adk.sessions import InMemorySessionService  # --- Define the Recommended Tool-Based Approach ---  def log\_user\_login(tool\_context: ToolContext) -> dict:  """  Updates the session state upon a user login event.  This tool encapsulates all state changes related to a user login.  Args:  tool\_context: Automatically provided by ADK, gives access to session state.  Returns:  A dictionary confirming the action was successful.  """  # Access the state directly through the provided context.  state = tool\_context.state    # Get current values or defaults, then update the state.  # This is much cleaner and co-locates the logic.  login\_count = state.get("user:login\_count", 0) + 1  state["user:login\_count"] = login\_count  state["task\_status"] = "active"  state["user:last\_login\_ts"] = time.time()  state["temp:validation\_needed"] = True    print("State updated from within the `log\_user\_login` tool.")    return {  "status": "success",  "message": f"User login tracked. Total logins: {login\_count}."  }  # --- Demonstration of Usage ---  # In a real application, an LLM Agent would decide to call this tool.  # Here, we simulate a direct call for demonstration purposes.  # 1. Setup  session\_service = InMemorySessionService()  app\_name, user\_id, session\_id = "state\_app\_tool", "user3", "session3"  session = 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}")  # 2. Simulate a tool call (in a real app, the ADK Runner does this)  # We create a ToolContext manually just for this standalone example.  from google.adk.tools.tool\_context import InvocationContext  mock\_context = ToolContext(  invocation\_context=InvocationContext(  app\_name=app\_name, user\_id=user\_id, session\_id=session\_id,  session=session, session\_service=session\_service  )  )  # 3. Execute the tool  log\_user\_login(mock\_context)  # 4. Check the updated state  updated\_session = session\_service.get\_session(app\_name, user\_id, session\_id)  print(f"State after tool execution: {updated\_session.state}")  # Expected output will show the same state change as the  # "Before" case,  # but the code organization is significantly cleaner  # and more robust. |

This code demonstrates a tool-based approach for managing user session state in an application. It defines a function *log\_user\_login*, which acts as a tool. This tool is responsible for updating the session state when a user logs in.

The function takes a ToolContext object, provided by the ADK, to access and modify the session's state dictionary. Inside the tool, it increments a *user:login\_count*, sets the t*ask\_status* to "active", records the *user:last\_login\_ts (timestamp)*, and adds a temporary flag temp:validation\_needed.

The demonstration part of the code simulates how this tool would be used. It sets up an in-memory session service and creates an initial session with some predefined state. A ToolContext is then manually created to mimic the environment in which the ADK Runner would execute the tool. The log\_user\_login function is called with this mock context. Finally, the code retrieves the session again to show that the state has been updated by the tool's execution. The goal is to show how encapsulating state changes within tools makes the code cleaner and more organized compared to directly manipulating state outside of tools.

Note that direct modification of the `session.state` dictionary after retrieving a session is strongly discouraged as it bypasses the standard event processing mechanism. Such direct changes will not be recorded in the session's event history, may not be persisted by the selected `SessionService`, could lead to concurrency issues, and will not update essential metadata such as timestamps. The recommended methods for updating the session state are using the `output\_key` parameter on an `LlmAgent` (specifically for the agent's final text responses) or including state changes within `EventActions.state\_delta` when appending an event via `session\_service.append\_event()`. The `session.state` should primarily be used for reading existing data.

To recap, when designing your state, keep it simple, use basic data types, give your keys clear names and use prefixes correctly, avoid deep nesting, and always update state using the append\_event process.

**Memory: Long-Term Knowledge with MemoryService**

In agent systems, the Session component maintains a record of the current chat history (events) and temporary data (state) specific to a single conversation. However, for agents to retain information across multiple interactions or access external data, long-term knowledge management is necessary. This is facilitated by the MemoryService.

|  |
| --- |
| # Example: Using InMemoryMemoryService  # This is suitable for local development and testing where data  # persistence across application restarts is not required.  # Memory content is lost when the app stops.  from google.adk.memory import InMemoryMemoryService  memory\_service = InMemoryMemoryService() |

Session and State can be conceptualized as short-term memory for a single chat session, whereas the Long-Term Knowledge managed by the MemoryService functions as a persistent and searchable repository. This repository may contain information from multiple past interactions or external sources. The MemoryService, as defined by the BaseMemoryService interface, establishes a standard for managing this searchable, long-term knowledge. Its primary functions include adding information, which involves extracting content from a session and storing it using the add\_session\_to\_memory method, and retrieving information, which allows an agent to query the store and receive relevant data using the search\_memory method.

The ADK offers several implementations for creating this long-term knowledge store. The InMemoryMemoryService provides a temporary storage solution suitable for testing purposes, but data is not preserved across application restarts. For production environments, the VertexAiRagMemoryService is typically utilized. This service leverages Google Cloud's Retrieval Augmented Generation (RAG) service, enabling scalable, persistent, and semantic search capabilities (Also, refer to the chapter 14 on RAG).

|  |
| --- |
| # Example: Using VertexAiRagMemoryService  # This is suitable for scalable production on GCP, leveraging  # Vertex AI RAG (Retrieval Augmented Generation) for persistent,  # searchable memory.  # Requires: pip install google-adk[vertexai], GCP  # setup/authentication, and a Vertex AI RAG Corpus.  from google.adk.memory import VertexAiRagMemoryService  # The resource name of your Vertex AI RAG Corpus  RAG\_CORPUS\_RESOURCE\_NAME = "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id" # Replace with your Corpus resource name  # Optional configuration for retrieval behavior  SIMILARITY\_TOP\_K = 5 # Number of top results to retrieve  VECTOR\_DISTANCE\_THRESHOLD = 0.7 # Threshold for vector similarity  memory\_service = VertexAiRagMemoryService(  rag\_corpus=RAG\_CORPUS\_RESOURCE\_NAME,  similarity\_top\_k=SIMILARITY\_TOP\_K,  vector\_distance\_threshold=VECTOR\_DISTANCE\_THRESHOLD  )  # When using this service, methods like add\_session\_to\_memory  # and search\_memory will interact with the specified Vertex AI  # RAG Corpus. |

**Hands-on code: Memory Management in LangChain and LangGraph**

In LangChain and LangGraph, Memory is a critical component for creating intelligent and natural-feeling conversational applications. It allows an AI agent to remember information from past interactions, learn from feedback, and adapt to user preferences. LangChain's memory feature provides the foundation for this by referencing a stored history to enrich current prompts and then recording the latest exchange for future use. As agents handle more complex tasks, this capability becomes essential for both efficiency and user satisfaction.

**Short-Term Memory:** This is thread-scoped, meaning it tracks the ongoing conversation within a single session or thread. It provides immediate context, but a full history can challenge an LLM's context window, potentially leading to errors or poor performance. LangGraph manages short-term memory as part of the agent's state, which is persisted via a checkpointer, allowing a thread to be resumed at any time.

**Long-Term Memory:** This stores user-specific or application-level data across sessions and is shared between conversational threads. It is saved in custom "namespaces" and can be recalled at any time in any thread. LangGraph provides stores to save and recall long-term memories, enabling agents to retain knowledge indefinitely.

LangChain provides several tools for managing conversation history, ranging from manual control to automated integration within chains.

**ChatMessageHistory: Manual Memory Management.** For direct and simple control over a conversation's history outside of a formal chain, the ChatMessageHistory class is ideal. It allows for the manual tracking of dialogue exchanges.

|  |
| --- |
| from langchain.memory import ChatMessageHistory  # Initialize the history object  history = ChatMessageHistory()  # Add user and AI messages  history.add\_user\_message("I'm heading to New York next week.")  history.add\_ai\_message("Great! It's a fantastic city.")  # Access the list of messages  print(history.messages) |

**ConversationBufferMemory: Automated Memory for Chains**. For integrating memory directly into chains, ConversationBufferMemory is a common choice. It holds a buffer of the conversation and makes it available to your prompt. Its behavior can be customized with two key parameters:

* memory\_key: A string that specifies the variable name in your prompt that will hold the chat history. It defaults to "history".
* return\_messages: A boolean that dictates the format of the history.
* If False (the default), it returns a single formatted string, which is ideal for standard LLMs.
* If True, it returns a list of message objects, which is the recommended format for Chat Models.

|  |
| --- |
| from langchain.memory import ConversationBufferMemory  # Initialize memory  memory = ConversationBufferMemory()  # Save a conversation turn  memory.save\_context({"input": "What's the weather like?"}, {"output": "It's sunny today."})  # Load the memory as a string  print(memory.load\_memory\_variables({})) |

Integrating this memory into an LLMChain allows the model to access the conversation's history and provide contextually relevant responses

|  |
| --- |
| from langchain\_openai import OpenAI  from langchain.chains import LLMChain  from langchain.prompts import PromptTemplate  from langchain.memory import ConversationBufferMemory  # 1. Define LLM and Prompt  llm = OpenAI(temperature=0)  template = """You are a helpful travel agent.  Previous conversation:  {history}  New question: {question}  Response:"""  prompt = PromptTemplate.from\_template(template)  # 2. Configure Memory  # The memory\_key "history" matches the variable in the prompt  memory = ConversationBufferMemory(memory\_key="history")  # 3. Build the Chain  conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)  # 4. Run the Conversation  response = conversation.predict(question="I want to book a flight.")  print(response)  response = conversation.predict(question="My name is Sam, by the way.")  print(response)  response = conversation.predict(question="What was my name again?")  print(response) |

For improved effectiveness with chat models, it is recommended to use a structured list of message objects by setting `return\_messages=True`.

|  |
| --- |
| from langchain\_openai import ChatOpenAI  from langchain.chains import LLMChain  from langchain.memory import ConversationBufferMemory  from langchain\_core.prompts import (  ChatPromptTemplate,  MessagesPlaceholder,  SystemMessagePromptTemplate,  HumanMessagePromptTemplate,  )  # 1. Define Chat Model and Prompt  llm = ChatOpenAI()  prompt = ChatPromptTemplate(  messages=[  SystemMessagePromptTemplate.from\_template("You are a friendly assistant."),  MessagesPlaceholder(variable\_name="chat\_history"),  HumanMessagePromptTemplate.from\_template("{question}")  ]  )  # 2. Configure Memory  # return\_messages=True is essential for chat models  memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)  # 3. Build the Chain  conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)  # 4. Run the Conversation  response = conversation.predict(question="Hi, I'm Jane.")  print(response)  response = conversation.predict(question="Do you remember my name?")  print(response) |

**Types of Long-Term Memory**: Long-term memory allows systems to retain information across different conversations, providing a deeper level of context and personalization. It can be broken down into three types analogous to human memory:

* **Semantic Memory: Remembering Facts:** This involves retaining specific facts and concepts, such as user preferences or domain knowledge. It is used to ground an agent's responses, leading to more personalized and relevant interactions. This information can be managed as a continuously updated user "profile" (a JSON document) or as a "collection" of individual factual documents.
* **Episodic Memory: Remembering Experiences:** This involves recalling past events or actions. For AI agents, episodic memory is often used to remember how to accomplish a task. In practice, it's frequently implemented through few-shot example prompting, where an agent learns from past successful interaction sequences to perform tasks correctly.
* **Procedural Memory: Remembering Rules:** This is the memory of how to perform tasks—the agent's core instructions and behaviors, often contained in its system prompt. It's common for agents to modify their own prompts to adapt and improve. An effective technique is "Reflection," where an agent is prompted with its current instructions and recent interactions, then asked to refine its own instructions.

Below is pseudo-code demonstrating how an agent might use reflection to update its procedural memory stored in a LangGraph BaseStore

|  |
| --- |
| # Node that updates the agent's instructions  def update\_instructions(state: State, store: BaseStore):  namespace = ("instructions",)  # Get the current instructions from the store  current\_instructions = store.search(namespace)[0]    # Create a prompt to ask the LLM to reflect on the conversation  # and generate new, improved instructions  prompt = prompt\_template.format(  instructions=current\_instructions.value["instructions"],  conversation=state["messages"]  )    # Get the new instructions from the LLM  output = llm.invoke(prompt)  new\_instructions = output['new\_instructions']    # Save the updated instructions back to the store  store.put(("agent\_instructions",), "agent\_a", {"instructions": new\_instructions})  # Node that uses the instructions to generate a response  def call\_model(state: State, store: BaseStore):  namespace = ("agent\_instructions", )  # Retrieve the latest instructions from the store  instructions = store.get(namespace, key="agent\_a")[0]    # Use the retrieved instructions to format the prompt  prompt = prompt\_template.format(instructions=instructions.value["instructions"])  # ... application logic continues |

LangGraph stores long-term memories as JSON documents in a store. Each memory is organized under a custom namespace (like a folder) and a distinct key (like a filename). This hierarchical structure allows for easy organization and retrieval of information. The following code demonstrates how to use InMemoryStore to put, get, and search for memories.

|  |
| --- |
| from langgraph.store.memory import InMemoryStore  # A placeholder for a real embedding function  def embed(texts: list[str]) -> list[list[float]]:  # In a real application, use a proper embedding model  return [[1.0, 2.0] for \_ in texts]  # Initialize an in-memory store. For production, use a database-backed store.  store = InMemoryStore(index={"embed": embed, "dims": 2})  # Define a namespace for a specific user and application context  user\_id = "my-user"  application\_context = "chitchat"  namespace = (user\_id, application\_context)  # 1. Put a memory into the store  store.put(  namespace,  "a-memory", # The key for this memory  {  "rules": [  "User likes short, direct language",  "User only speaks English & python",  ],  "my-key": "my-value",  },  )  # 2. Get the memory by its namespace and key  item = store.get(namespace, "a-memory")  print("Retrieved Item:", item)  # 3. Search for memories within the namespace, filtering by content  # and sorting by vector similarity to the query.  items = store.search(  namespace,  filter={"my-key": "my-value"},  query="language preferences"  )  print("Search Results:", items) |

**Vertex Memory Bank**

Memory Bank, a managed service in the Vertex AI Agent Engine, provides agents with persistent, long-term memory. The service uses Gemini models to asynchronously analyze conversation histories to extract key facts and user preferences.

This information is stored persistently, organized by a defined scope like user ID, and intelligently updated to consolidate new data and resolve contradictions. Upon starting a new session, the agent retrieves relevant memories through either a full data recall or a similarity search using embeddings. This process allows an agent to maintain continuity across sessions and personalize responses based on recalled information.

The agent's runner interacts with the VertexAiMemoryBankService, which is initialized first. This service handles the automatic storage of memories generated during the agent's conversations. Each memory is tagged with a unique USER\_ID and APP\_NAME, ensuring accurate retrieval in the future.

|  |
| --- |
| from google.adk.memory import VertexAiMemoryBankService  agent\_engine\_id = agent\_engine.api\_resource.name.split("/")[-1]  memory\_service = VertexAiMemoryBankService(  project="PROJECT\_ID",  location="LOCATION",  agent\_engine\_id=agent\_engine\_id  )  session = await session\_service.get\_session(  app\_name=app\_name,  user\_id="USER\_ID",  session\_id=session.id  )  await memory\_service.add\_session\_to\_memory(session) |

Memory Bank offers seamless integration with the Google ADK, providing an immediate out-of-the-box experience. For users of other agent frameworks, such as LangGraph and CrewAI, Memory Bank also offers support through direct API calls. Online code examples demonstrating these integrations are readily available for interested readers.

**At a Glance**

**What**: Agentic systems need to remember information from past interactions to perform complex tasks and provide coherent experiences. Without a memory mechanism, agents are stateless, unable to maintain conversational context, learn from experience, or personalize responses for users. This fundamentally limits them to simple, one-shot interactions, failing to handle multi-step processes or evolving user needs. The core problem is how to effectively manage both the immediate, temporary information of a single conversation and the vast, persistent knowledge gathered over time.

**Why:** The standardized solution is to implement a dual-component memory system that distinguishes between short-term and long-term storage. Short-term, contextual memory holds recent interaction data within the LLM's context window to maintain conversational flow. For information that must persist, long-term memory solutions use external databases, often vector stores, for efficient, semantic retrieval. Agentic frameworks like the Google ADK provide specific components to manage this, such as Session for the conversation thread and State for its temporary data. A dedicated MemoryService is used to interface with the long-term knowledge base, allowing the agent to retrieve and incorporate relevant past information into its current context.

**Rule of thumb:** Use this pattern when an agent needs to do more than answer a single question. It is essential for agents that must maintain context throughout a conversation, track progress in multi-step tasks, or personalize interactions by recalling user preferences and history. Implement memory management whenever the agent is expected to learn or adapt based on past successes, failures, or newly acquired information.

**Visual summary**

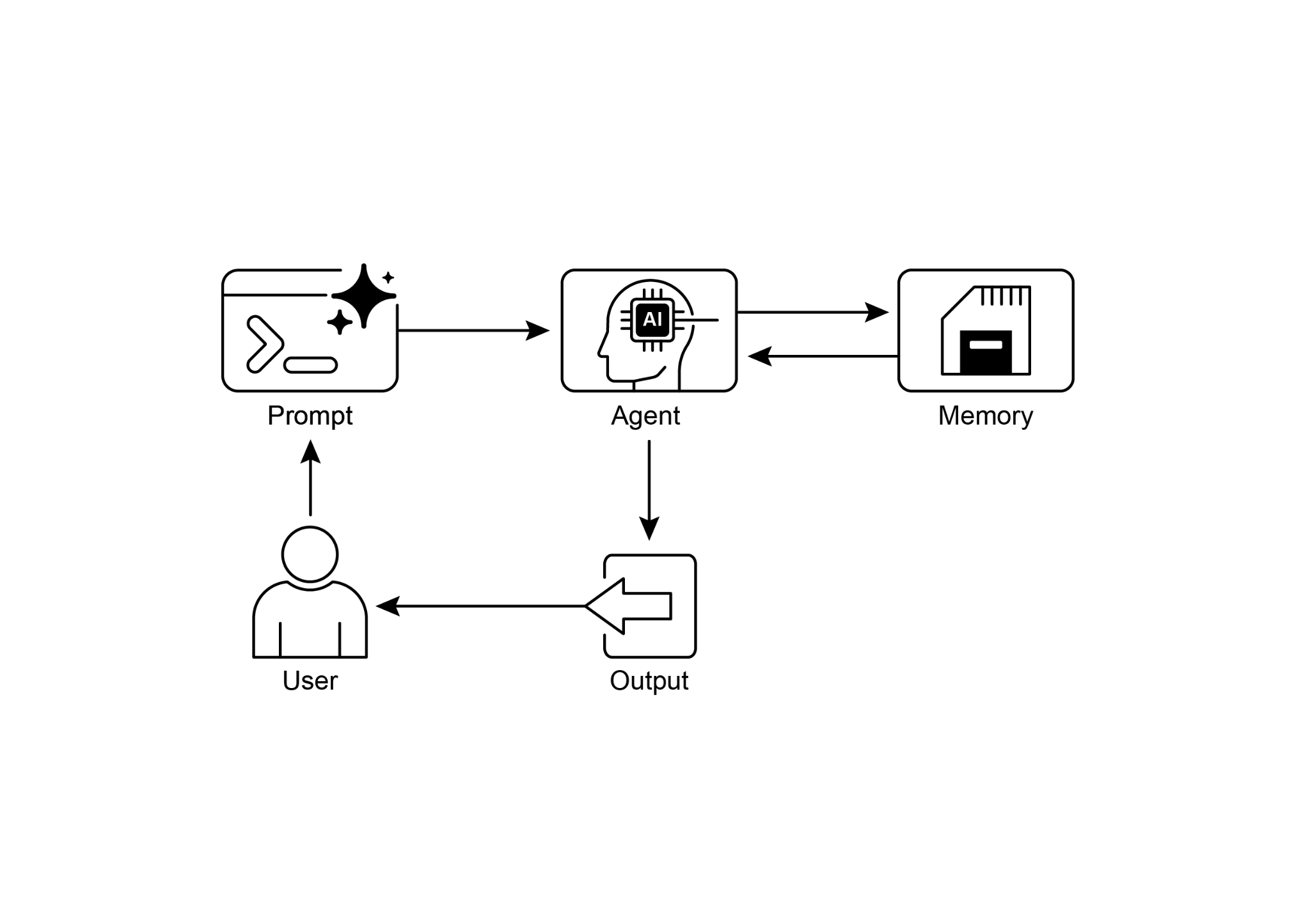


Fig.1: Memory management design pattern

**Key Takeaways**

To quickly recap the main points about memory management:

* Memory is super important for agents to keep track of things, learn, and personalize interactions.
* Conversational AI relies on both short-term memory for immediate context within a single chat and long-term memory for persistent knowledge across multiple sessions.
* Short-term memory (the immediate stuff) is temporary, often limited by the LLM's context window or how the framework passes context.
* Long-term memory (the stuff that sticks around) saves info across different chats using outside storage like vector databases and is accessed by searching.
* Frameworks like ADK have specific parts like Session (the chat thread), State (temporary chat data), and MemoryService (the searchable long-term knowledge) to manage memory.
* ADK's SessionService handles the whole life of a chat session, including its history (events) and temporary data (state).
* ADK's session.state is a dictionary for temporary chat data. Prefixes (user:, app:, temp:) tell you where the data belongs and if it sticks around.
* In ADK, you should update state by using EventActions.state\_delta or output\_key when adding events, not by changing the state dictionary directly.
* ADK's MemoryService is for putting info into long-term storage and letting agents search it, often using tools.
* LangChain offers practical tools like ConversationBufferMemory to automatically inject the history of a single conversation into a prompt, enabling an agent to recall immediate context.
* LangGraph enables advanced, long-term memory by using a store to save and retrieve semantic facts, episodic experiences, or even updatable procedural rules across different user sessions.
* Memory Bank is a managed service that provides agents with persistent, long-term memory by automatically extracting, storing, and recalling user-specific information to enable personalized, continuous conversations across frameworks like Google's ADK, LangGraph, and CrewAI.

**Conclusion**

This chapter dove into the really important job of memory management for agent systems, showing the difference between the short-lived context and the knowledge that sticks around for a long time. We talked about how these types of memory are set up and where you see them used in building smarter agents that can remember things. We took a detailed look at how Google ADK gives you specific pieces like Session, State, and MemoryService to handle this. Now that we've covered how agents can remember things, both short-term and long-term, we can move on to how they can learn and adapt. The next pattern ​​"Learning and Adaptation" is about an agent changing how it thinks, acts, or what it knows, all based on new experiences or data.

**References**

1. ADK Memory, <https://google.github.io/adk-docs/sessions/memory/>
2. LangGraph Memory, <https://langchain-ai.github.io/langgraph/concepts/memory/>
3. Vertex AI Agent Engine Memory Bank, <https://cloud.google.com/blog/products/ai-machine-learning/vertex-ai-memory-bank-in-public-preview>

**第8章\_内存管理**

第8章：内存管理

有效的内存管理对于智能体保留信息至关重要。与人类一样，智能体也需要不同类型的内存才能高效运行。本章深入探讨内存管理，特别关注智能体的即时（短期）和持久（长期）内存需求。

在智能体系统中，记忆指的是智能体保留并利用过去交互、观察和学习经验中信息的能力。这种能力使智能体能够做出明智的决策、保持对话上下文，并随着时间的推移不断改进。智能体记忆通常分为两种主要类型：

* **短期记忆（上下文记忆）：**与工作记忆类似，它存储当前正在处理或最近访问过的信息。对于使用大语言模型（LLM）的智能体而言，短期记忆主要存在于上下文窗口中。该窗口包含当前交互中的近期消息、智能体回复、工具使用结果以及智能体反思，所有这些都为大语言模型的后续响应和行动提供依据。上下文窗口的容量有限，限制了智能体可直接访问的近期信息量。高效的短期记忆管理包括在这个有限空间内保留最相关的信息，可能通过总结旧对话片段或强调关键细节等技术来实现。具有“长上下文”窗口的模型的出现只是扩大了这种短期记忆的容量，允许在单次交互中保留更多信息。然而，这种上下文仍然是短暂的，会话结束后就会丢失，并且每次处理都可能成本高昂且效率低下。因此，智能体需要不同类型的记忆来实现真正的持久性，回忆过去交互中的信息，并建立持久的知识库。
* **长期记忆（持久记忆）：**它充当信息的存储库，供智能体在各种交互、任务或较长时间段内保留信息，类似于长期知识库。数据通常存储在智能体的直接处理环境之外，常见于数据库、知识图谱或向量数据库中。在向量数据库中，信息被转换为数值向量并存储，使智能体能够基于语义相似性而非精确的关键词匹配来检索数据，这一过程被称为语义搜索。当智能体需要从长期记忆中获取信息时，它会查询外部存储，检索相关数据，并将其整合到短期上下文中以供即时使用，从而将先验知识与当前交互相结合。

**实际应用与用例**

内存管理对于智能体跟踪信息并随时间推移进行智能操作至关重要。这对于智能体超越基本问答能力是必不可少的。应用包括：

* **聊天机器人和对话式AI：**保持对话流畅依赖于短期记忆。聊天机器人需要记住用户之前的输入，以便提供连贯的回应。长期记忆使聊天机器人能够回忆起用户偏好、过去的问题或之前的讨论，从而提供个性化和持续的交互。
* **面向任务的智能体：**管理多步骤任务的智能体需要短期记忆来跟踪之前的步骤、当前进度和总体目标。这些信息可能存储在任务的上下文或临时存储中。长期记忆对于访问不在直接上下文中的特定用户相关数据至关重要。
* **个性化体验：**提供定制化交互的智能体利用长期记忆来存储和检索用户偏好、过往行为和个人信息。这使得智能体能够调整其回应和建议。
* **学习与改进：**智能体可以通过从过去的交互中学习来提升其性能。成功的策略、错误和新信息都被存储在长期记忆中，以便于未来的调整。强化学习智能体以这种方式存储所学的策略或知识。
* **信息检索（RAG）：**旨在回答问题的智能体访问知识库，即它们的长期记忆，通常在检索增强生成（RAG）中实现。智能体检索相关文档或数据以提供其回答的依据。
* **自主系统：**机器人或自动驾驶汽车需要内存来存储地图、路线、物置和学习到的行为。这涉及到用于即时环境的短期记忆和用于一般环境知识的长期记忆。

记忆使智能体能够保存历史记录、学习、个性化交互并处理复杂的、与时间相关的问题。

**实践代码：谷歌智能体开发套件（ADK）中的内存管理**

谷歌智能体开发套件（ADK）提供了一种结构化的方法来管理上下文和内存，包括用于实际应用的组件。扎实掌握ADK的会话、状态和内存对于构建需要保留信息的智能体至关重要。

就像在人际交流中一样，智能体需要具备回忆先前交流内容的能力，才能进行连贯自然的对话。ADK通过三个核心概念及其相关服务简化了上下文管理。

与座席的每次交互都可视为一个独特的对话线程。座席可能需要访问早期交互的数据。ADK按以下方式对其进行结构化处理：

* **会话：**一个单独的聊天线程，用于记录特定交互的消息和操作（事件），还存储与该对话相关的临时数据（状态）。
* **状态（会话状态）：**存储在会话中的数据，仅包含与当前活动聊天线程相关的信息。
* **记忆：**一个可搜索的信息库，其信息来源于过往的各种聊天记录或外部资源，作为超越即时对话的数据检索资源。

ADK为管理构建复杂、有状态且具备上下文感知能力的智能体所必需的关键组件提供专门服务。会话服务（SessionService）通过处理聊天线程（会话对象）的启动、记录和终止来管理它们，而记忆服务（MemoryService）则负责长期知识（记忆）的存储和检索。

会话服务和内存服务均提供多种配置选项，允许用户根据应用需求选择存储方式。为测试目的提供了内存选项，不过数据在重启后不会持久化。对于持久存储和可扩展性，ADK还支持数据库和基于云的服务。

**会话：跟踪每次聊天**

ADK中的会话对象旨在跟踪和管理各个聊天线程。在与代理发起对话时，会话服务会生成一个会话对象，以 `google.adk.sessions.Session` 表示。该对象封装了与特定对话线程相关的所有数据，包括唯一标识符（id、应用名称、用户id）、按时间顺序记录的事件（以事件对象表示）、用于存储会话特定临时数据的区域（称为状态），以及指示最后更新时间的时间戳（last\_update\_time）。开发人员通常通过会话服务间接与会话对象进行交互。会话服务负责管理对话会话的生命周期，包括启动新会话、恢复以前的会话、记录会话活动（包括状态更新）、识别活跃会话，以及管理会话数据的删除。ADK提供了几种会话服务实现，它们对会话历史记录和临时数据采用不同的存储机制，例如内存会话服务，它适用于测试，但不提供跨应用重启的数据持久性。

|  |
| --- |
| #示例：使用InMemorySessionService  # 这适用于本地开发和测试，其中数据  # 应用程序重启时的持久性不是必需的。  from google.adk.sessions import InMemorySessionService  session\_service = InMemorySessionService() |

如果你希望可靠地保存到你管理的数据库中，那么还有 DatabaseSessionService。

|  |
| --- |
| #示例：使用DatabaseSessionService  # 这适用于需要持久存储的生产或开发环境。  # 你需要配置一个数据库 URL（例如，用于 SQLite、PostgreSQL 等）。  # 要求：pip install google-adk[sqlalchemy] 并安装数据库驱动（例如，PostgreSQL 使用 psycopg2）  from google.adk.sessions import DatabaseSessionService  # 使用本地SQLite文件的示例：  db\_url = "sqlite:///./my\_agent\_data.db"  session\_service = DatabaseSessionService(db\_url=db\_url) |

此外，还有VertexAiSessionService，它利用Vertex AI基础设施在谷歌云平台上实现可扩展的生产。

|  |
| --- |
| #示例：使用Vertex AI会话服务  #这适用于在谷歌云平台上进行可扩展生产，利用  # Vertex AI会话管理基础设施。  # 要求：pip install google-adk[vertexai] 并进行 GCP 设置/身份验证  from google.adk.sessions import VertexAiSessionService  PROJECT\_ID = "your-gcp-project-id" # 替换为你的 GCP 项目 ID  LOCATION = "us-central1" # 替换为您所需的GCP区域  # 此服务使用的应用名称应与推理引擎 ID 或名称相对应  REASONING\_ENGINE\_APP\_NAME = "projects/your-gcp-project-id/locations/us-central1/reasoningEngines/your-engine-id" # 替换为您的推理引擎资源名称  session\_service = VertexAiSessionService(project=PROJECT\_ID, location=LOCATION)  # 使用此服务时，请将 REASONING\_ENGINE\_APP\_NAME 传递给服务方法：  # session\_service.create\_session(app\_name=REASONING\_ENGINE\_APP\_NAME,...)  # session\_service.get\_session(app\_name=REASONING\_ENGINE\_APP\_NAME,...)  # session\_service.append\_event(session, event, app\_name=REASONING\_ENGINE\_APP\_NAME)  # session\_service.delete\_session(app\_name=REASONING\_ENGINE\_APP\_NAME,...) |

选择合适的会话服务至关重要，因为它决定了代理的交互历史和临时数据的存储方式及其持久性。

每次消息交换都涉及一个循环过程：接收消息，运行器使用会话服务检索或建立会话，代理使用会话的上下文（状态和历史交互）处理消息，代理生成响应并可能更新状态，运行器将此封装为事件，会话服务的 append\_event 方法记录新事件并更新存储中的状态。然后会话等待下一条消息。理想情况下，当交互结束时，使用 delete\_session 方法终止会话。此过程说明了会话服务如何通过管理特定于会话的历史记录和临时数据来保持连续性。

**状态：会话的便笺本**

在ADK中，每个会话（代表一个聊天线程）都包含一个状态组件，类似于代理在特定对话期间的临时工作内存。会话事件记录整个聊天历史，而会话状态则存储和更新与当前活跃聊天相关的动态数据点。

从根本上说，session.state作为一个字典运行，以键值对的形式存储数据。其核心功能是使智能体能够保留和管理连贯对话所必需的细节，如用户偏好、任务进度、增量数据采集或影响后续智能体行动的条件标志。

状态的结构由字符串键和可序列化的Python类型的值组成，这些类型包括字符串、数字、布尔值、列表以及包含这些基本类型的字典。状态是动态的，会在整个对话过程中不断演变。这些变化的持久性取决于所配置的会话服务。

可以使用键前缀来实现状态组织，以定义数据范围和持久性。无前缀的键是特定于会话的。

* 用户：前缀将数据与所有会话中的用户ID关联起来。
* 应用程序：前缀指定在应用程序所有用户之间共享的数据。
* “temp:”前缀表示数据仅在当前处理回合有效，不会被持久存储。

代理通过单个session.state字典访问所有状态数据。SessionService处理数据检索、合并和持久化。通过session\_service.append\_event()将事件添加到会话历史记录时，应更新状态。这确保了准确跟踪、在持久服务中正确保存以及安全处理状态更改。

1. **简单方法：使用 output\_key（用于代理文本回复）：**如果您只想将代理的最终文本响应直接保存到状态中，这是最简单的方法。在设置 LlmAgent 时，只需告诉它您要使用的 output\_key。Runner 会看到这个设置，并在追加事件时自动创建必要的操作，将响应保存到状态中。让我们看一个通过 output\_key 演示状态更新的代码示例。

|  |
| --- |
| # 从 Google 代理开发套件 (ADK) 导入必要的类  from google.adk.agents import LlmAgent  from google.adk.sessions import InMemorySessionService, Session  from google.adk.runners import Runner  从google.genai.types导入Content, Part  # 使用输出键定义一个 LlmAgent。  greeting\_agent = LlmAgent(  name="迎宾员",  model="gemini-2.0-flash",  instruction="生成简短友好的问候语",  output\_key="last\_greeting"  )  # --- 设置运行器和会话 ---  app\_name、user\_id、session\_id = "state\_app"、"user1"、"session1"  session\_service = InMemorySessionService()  runner = Runner(  agent=greeting\_agent,  app\_name=应用名称,  session\_service=会话服务  )  session = session\_service.create\_session(  app\_name=应用名称,  user\_id=用户ID,  session\_id=会话ID  )  print(f"初始状态: {session.state}")  # --- 运行代理 ---  user\_message = Content(parts=[Part(text="你好")])  print("\n--- 运行代理 ---")  for event in runner.run(  user\_id=用户ID,  session\_id=session\_id,  new\_message=user\_message  ):  if event.is\_final\_response():  print("代理已响应。")  # --- 检查更新状态 ---  # 在运行器完成处理所有事件 \*之后\* 正确检查状态。  updated\_session = session\_service.get\_session(app\_name, user\_id, session\_id)  print(f"\n代理运行后的状态: {updated\_session.state}") |

在幕后，Runner会识别你的output\_key，并在调用append\_event时自动创建带有state\_delta的必要操作。

1. **标准方法：使用 EventActions.state\_delta（适用于更复杂的更新）：**当你需要执行更复杂的操作时，例如同时更新多个键、保存不只是文本的内容、针对特定范围（如 user: 或 app:），或者进行与代理的最终文本回复无关的更新，你需要手动构建一个包含状态更改的字典（即 state\_delta），并将其包含在你正在追加的事件的 EventActions 中。让我们来看一个示例：

|  |
| --- |
| 导入时间  from google.adk.tools.tool\_context import ToolContext  from google.adk.sessions import InMemorySessionService  # --- 定义推荐的基于工具的方法 ---  def log\_user\_login(tool\_context: ToolContext) -> dict:  """  在用户登录事件发生时更新会话状态。  此工具封装了与用户登录相关的所有状态更改。  参数：  tool\_context：由ADK自动提供，可访问会话状态。  返回值：  一个确认操作成功的字典。  """  # 通过提供的上下文直接访问状态。  state = tool\_context.state    # 获取当前值或默认值，然后更新状态。  # 这样更简洁，且将逻辑集中在一起。  login\_count = state.get("user:login\_count", 0) + 1  state["user:login\_count"] = 登录次数  state["task\_status"] = "active"  state["user:last\_login\_ts"] = time.time()  state["temp:validation\_needed"] = True    print("状态已在 `log\_user\_login` 工具内部更新。")    返回 {  "状态": "成功",  "message": f"用户登录已记录。总登录次数：{login\_count}。"  }  # --- 使用演示 ---  # 在实际应用中，大语言模型智能体将决定调用此工具。  # 在这里，我们模拟一个直接调用以作演示。  # 1. 设置  session\_service = InMemorySessionService()  app\_name、user\_id、session\_id = "state\_app\_tool"、"user3"、"session3"  session = session\_service.create\_session(  app\_name=应用名称,  user\_id=用户ID,  session\_id=session\_id,  state={"user:login\_count": 0, "task\_status": "idle"}  )  print(f"初始状态: {session.state}")  # 2. 模拟工具调用（在实际应用中，ADK运行器会执行此操作）  # 我们仅为这个独立示例手动创建一个工具上下文。  从google.adk.tools.tool\_context导入InvocationContext  mock\_context = ToolContext(  invocation\_context=InvocationContext(  app\_name=应用名称, user\_id=用户ID, session\_id=会话ID,  session=会话, session\_service=会话服务  )  )  # 3. 执行工具  log\_user\_login(mock\_context)  # 4. 检查更新后的状态  updated\_session = session\_service.get\_session(app\_name, user\_id, session\_id)  print(f"工具执行后的状态: {updated\_session.state}")  #预期输出将显示与相同的状态变化  # “之前”的情况，  # 但代码结构明显更简洁  # 且更稳健。 |

此代码展示了一种基于工具的方法，用于管理应用程序中的用户会话状态。它定义了一个函数*log\_user\_login*，该函数作为一个工具。此工具负责在用户登录时更新会话状态。

该函数接收一个由ADK提供的ToolContext对象，以访问和修改会话的状态字典。在工具内部，它会增加*user:login\_count*，将*task\_status*设置为"active"，记录*user:last\_login\_ts（时间戳）*，并添加一个临时标志temp:validation\_needed。

代码的演示部分模拟了该工具的使用方式。它设置了一个内存中的会话服务，并创建了一个具有预定义状态的初始会话。然后手动创建一个工具上下文，以模拟ADK运行器执行该工具的环境。使用这个模拟上下文调用log\_user\_login函数。最后，代码再次检索会话，以展示工具的执行已更新了状态。目的是展示将状态更改封装在工具内部如何使代码比直接在工具外部操作状态更简洁、更有条理。

请注意，强烈不建议在获取会话后直接修改 `session.state` 字典，因为这会绕过标准的事件处理机制。此类直接更改不会记录在会话的事件历史中，所选的 `SessionService` 可能不会持久化这些更改，可能会导致并发问题，并且不会更新诸如时间戳之类的关键元数据。更新会话状态的推荐方法是在 `LlmAgent` 上使用 `output\_key` 参数（专门用于处理代理的最终文本响应），或者在通过 `session\_service.append\_event()` 追加事件时在 `EventActions.state\_delta` 中包含状态更改。`session.state` 主要应用于读取存量数据。

总结一下，在设计状态时，要保持简单，使用基本数据类型，为键取清晰的名称并正确使用前缀，避免深度嵌套，始终使用 append\_event 过程更新状态。

**记忆：借助MemoryService实现的长期知识**

在智能体系统中，会话组件维护着当前聊天历史（事件）和特定于单个对话的临时数据（状态）的记录。然而，为了让智能体在多次交互中保留信息或访问外部数据，长期知识管理是必要的。这由内存服务来实现。

|  |
| --- |
| #示例：使用InMemoryMemoryService  # 这适用于本地开发和测试，其中数据  # 应用程序重启时不需要持久化。  # 应用程序停止时，内存内容将丢失。  from google.adk.memory import InMemoryMemoryService  memory\_service = InMemoryMemoryService() |

会话和状态可以被概念化为单个聊天会话的短期记忆，而由MemoryService管理的长期知识则充当一个持久且可搜索的存储库。这个存储库可能包含来自多个过去交互或外部来源的信息。由BaseMemoryService接口定义的MemoryService为管理这种可搜索的长期知识制定了标准。其主要功能包括添加信息（涉及从会话中提取内容并使用add\_session\_to\_memory方法进行存储）和检索信息（允许代理使用search\_memory方法查询存储库并接收相关数据）。

ADK提供了几种创建这种长期知识存储的实现方式。InMemoryMemoryService提供了一种适合测试目的的临时存储解决方案，但数据在应用程序重启后不会保留。对于生产环境，通常会使用VertexAiRagMemoryService。该服务利用谷歌云的检索增强生成（RAG）服务，实现可扩展、持久和语义搜索功能（另请参阅第14章关于RAG的内容）。

|  |
| --- |
| #示例：使用VertexAiRagMemoryService  #这适用于在GCP上进行可扩展生产，利用  # Vertex AI RAG（检索增强生成）用于持久化，  #可搜索内存。  # 要求：pip install google-adk[vertexai]，GCP  # 设置/身份验证，以及 Vertex AI RAG 语料库。  from google.adk.memory import VertexAiRagMemoryService  # 您的 Vertex AI RAG 语料库的资源名称  RAG\_CORPUS\_RESOURCE\_NAME = "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id" # 替换为您的语料库资源名称  # 检索行为的可选配置  SIMILARITY\_TOP\_K = 5 # 要检索的前几名结果的数量  VECTOR\_DISTANCE\_THRESHOLD = 0.7 # 向量相似度阈值  memory\_service = VertexAiRagMemoryService(  rag\_corpus=RAG\_CORPUS\_RESOURCE\_NAME,  similarity\_top\_k=SIMILARITY\_TOP\_K,  vector\_distance\_threshold=VECTOR\_DISTANCE\_THRESHOLD  )  # 使用此服务时，像 add\_session\_to\_memory 这样的方法  # 并且搜索内存将与指定的 Vertex AI 进行交互  # RAG语料库。 |

**实践代码：LangChain和LangGraph中的内存管理**

在LangChain和LangGraph中，记忆是创建智能且自然流畅的对话应用程序的关键组件。它使AI智能体能够记住过去交互中的信息，从反馈中学习，并适应用户偏好。LangChain的记忆功能通过引用存储的历史记录来丰富当前提示，然后记录最新的交流以供将来使用，从而为这一切奠定了基础。随着智能体处理更复杂的任务，这种能力对于效率和用户满意度都变得至关重要。

**短期记忆：**这是线程作用域的，意味着它跟踪单个会话或线程内正在进行的对话。它提供即时上下文，但完整的历史记录可能会超出大语言模型（LLM）的上下文窗口，从而可能导致错误或性能不佳。LangGraph将短期记忆作为代理状态的一部分进行管理，该状态通过检查点器持久化，允许线程在任何时候恢复。

**长期记忆：**它跨会话存储用户特定或应用程序级别的数据，并在对话线程之间共享。它保存在自定义的“命名空间”中，可以在任何线程的任何时间被调用。LangGraph提供存储和调用长期记忆的功能，使智能体能够无限期地保留知识。

LangChain提供了多种管理对话历史的工具，从手动控制到链内自动集成不等。

**ChatMessageHistory：手动管理对话历史。**如果需要在正式链之外直接且简单地控制对话历史，ChatMessageHistory类是理想之选。它允许手动跟踪对话交流。

|  |
| --- |
| from langchain.memory import ChatMessageHistory  # 初始化历史对象  history = ChatMessageHistory()  # 添加用户和AI消息  history.add\_user\_message("我下周要去纽约。")  history.add\_ai\_message("太棒了！这是一座很棒的城市。")  # 访问消息列表  print(history.messages) |

**对话缓冲记忆：链的自动记忆**。若要将记忆直接集成到链中，对话缓冲记忆是常见的选择。它保存对话的缓冲区，并使其可用于你的提示。其行为可通过两个关键参数进行定制：

* memory\_key：一个字符串，用于指定提示中保存聊天历史的变量名。默认值为 "history"。
* return\_messages：一个布尔值，用于指定历史记录的格式。
* 如果为False（默认值），则返回单个格式化字符串，这对于标准大语言模型（LLMs）来说是理想的。
* 如果为True，则返回消息对象列表，这是聊天模型推荐的格式。

|  |
| --- |
| from langchain.memory import ConversationBufferMemory  # 初始化内存  memory = ConversationBufferMemory()  #保存对话轮次  memory.save\_context({"input": "天气怎么样？"}, {"output": "今天天气晴朗。"})  # 将内存加载为字符串  print(memory.load\_memory\_variables({})) |

将此记忆集成到LLMChain中，可使模型访问对话历史并提供与上下文相关的响应

|  |
| --- |
| from langchain\_openai import OpenAI  从langchain.chains导入LLMChain  from langchain.prompts import PromptTemplate  from langchain.memory import ConversationBufferMemory  # 1. 定义大语言模型（LLM）和提示词  llm = OpenAI(temperature=0)  模板 = """你是一位乐于助人的旅行社代理人。  之前的对话：  {历史}  新问题：{question}  响应："""  prompt = PromptTemplate.from\_template(template)  # 2. 配置内存  # 内存键 "history" 与提示中的变量匹配  memory = ConversationBufferMemory(memory\_key="history")  # 3. 构建链  conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)  # 4. 运行对话  response = conversation.predict(question="我想预订航班。")  打印(response)  response = conversation.predict(question="顺便说一下，我叫山姆。")  打印(response)  response = conversation.predict(question="我的名字是什么来着？")  打印(response) |

为提高聊天模型的有效性，建议通过设置 `return\_messages=True` 使用结构化的消息对象列表。

|  |
| --- |
| from langchain\_openai import ChatOpenAI  从langchain.chains导入LLMChain  from langchain.memory import ConversationBufferMemory  from langchain\_core.prompts import (  ChatPromptTemplate  消息占位符  系统消息提示模板  人类消息提示模板  )  # 1. 定义聊天模型和提示  llm = ChatOpenAI()  prompt = ChatPromptTemplate(  messages=[  SystemMessagePromptTemplate.from\_template("你是一个友好的助手。"),  MessagesPlaceholder(变量名="chat\_history"),  HumanMessagePromptTemplate.from\_template("{question}")  ]  )  # 2. 配置内存  # return\_messages=True对于聊天模型至关重要  memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)  # 3. 构建链  conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)  # 4. 运行对话  response = conversation.predict(question="Hi, I'm Jane.")  打印(response)  response = conversation.predict(question="你还记得我的名字吗？")  打印(response) |

**长期记忆的类型**：长期记忆使系统能够在不同对话中保留信息，提供更深入的上下文和个性化服务。它可以分为三种类似于人类记忆的类型：

* **语义记忆：事实记忆**：这涉及保留特定的事实和概念，如用户偏好或领域知识。它用于为智能体的响应提供依据，从而实现更个性化和相关的交互。这些信息可以作为持续更新的用户“档案”（JSON文档）或单个事实文档的“集合”进行管理。
* **情景记忆：回忆经历：**这涉及回忆过去的事件或行动。对于AI智能体来说，情景记忆通常用于记住如何完成一项任务。在实践中，它经常通过少样本示例提示来实现，即智能体从过去成功的交互序列中学习，以正确执行任务。
* **程序性记忆：规则记忆：**这是关于如何执行任务的记忆——即代理的核心指令和行为，通常包含在其系统提示中。代理通常会修改自己的提示以适应和改进。一种有效的技术是“反思”，即向代理提供其当前指令和最近的交互信息，然后要求其完善自己的指令。

以下是一段伪代码，展示了智能体如何使用反思来更新存储在LangGraph BaseStore中的过程性记忆

|  |
| --- |
| # 更新代理指令的节点  def update\_instructions(state: State, store: BaseStore):  namespace = ("instructions",)  # 从存储中获取当前指令  current\_instructions = store.search(namespace)[0]    # 创建一个提示，要求大语言模型（LLM）对对话进行反思  # 并生成新的、改进的指令  prompt = prompt\_template.format(  instructions=current\_instructions.value["instructions"],  conversation=state["messages"]  )    # 从大语言模型获取新指令  output = llm.invoke(prompt)  new\_instructions = output['new\_instructions']    # 将更新后的说明保存回存储库  store.put(("agent\_instructions",), "agent\_a", {"instructions": new\_instructions})  # 使用指令生成响应的节点  def call\_model(state: State, store: BaseStore):  namespace = ("agent\_instructions", )  # 从存储中检索最新指令  instructions = store.get(namespace, key="agent\_a")[0]    # 使用检索到的说明来格式化提示  prompt = prompt\_template.format(instructions=instructions.value["instructions"])  #...应用逻辑继续 |

LangGraph将长期记忆以JSON文档的形式存储在存储库中。每个记忆都组织在自定义命名空间（如文件夹）和独特的键（如文件名）之下。这种分层结构便于信息的组织和检索。以下代码展示了如何使用InMemoryStore来存储、获取和搜索记忆。

|  |
| --- |
| from langgraph.store.memory import InMemoryStore  # 真实嵌入函数的占位符  def embed(texts: list[str]) -> list[list[float]]:  # 在实际应用中，请使用合适的嵌入模型  return [[1.0, 2.0] for \_ in texts]  # 初始化一个内存存储。在生产环境中，请使用数据库支持的存储。  store = InMemoryStore(index={"embed": embed, "dims": 2})  # 为特定用户和应用程序上下文定义一个命名空间  user\_id = "my-user"  application\_context = "闲聊"  命名空间 = (用户ID, 应用程序上下文)  # 1. 将一个内存放入存储中  store.put(  命名空间  "a-memory", # 此内存的键  {  "规则": [  "用户喜欢简短、直接的语言",  "用户只会说英语和Python",  ],  "my-key": "my-value",  },  )  # 2. 通过命名空间和键获取内存  item = store.get(namespace, "a-memory")  print("已检索到的项目:", item)  # 3. 在命名空间内搜索记忆，按内容进行过滤  #并按与查询的向量相似度进行排序。  items = store.search(  命名空间  filter={"my-key": "my-value"},  query="语言偏好"  )  print("搜索结果:", items) |

**顶点内存库**

内存库（Memory Bank）是Vertex AI智能体引擎中的一项托管服务，它为智能体提供持久的长期记忆。该服务使用Gemini模型异步分析对话历史，以提取关键事实和用户偏好。

这些信息被持久化存储，按用户 ID 等定义的范围进行组织，并智能更新以整合新数据和解决矛盾。在开始新会话时，智能体通过全量数据召回或使用嵌入向量的相似度搜索来检索相关记忆。这一过程使智能体能够在不同会话间保持连贯性，并根据召回的信息个性化响应。

代理的运行器与VertexAiMemoryBankService进行交互，该服务首先被初始化。此服务处理在代理对话期间生成的记忆的自动存储。每个记忆都标记有唯一的USER\_ID和APP\_NAME，确保未来能够准确检索。

|  |
| --- |
| from google.adk.memory import VertexAiMemoryBankService  agent\_engine\_id = agent\_engine.api\_resource.name.split("/")[-1]  memory\_service = VertexAiMemoryBankService(  project="PROJECT\_ID",  location="LOCATION",  agent\_engine\_id=agent\_engine\_id  )  session = await session\_service.get\_session(  app\_name=应用名称,  user\_id="USER\_ID",  session\_id=session.id  )  await memory\_service.add\_session\_to\_memory(session) |

Memory Bank与谷歌ADK无缝集成，提供即时开箱即用的体验。对于其他代理框架（如LangGraph和CrewAI）的用户，Memory Bank还通过直接API调用提供支持。展示这些集成的在线代码示例随时可供感兴趣的读者使用。

**概览**

**什么**：能动系统需要记住过往交互中的信息，才能执行复杂任务并提供连贯的体验。如果没有记忆机制，智能体就是无状态的，无法维持对话上下文、从经验中学习或为用户定制响应。这从根本上将它们限制在简单的一次互中，无法处理多步骤流程或不断变化的用户需求。核心问题在于如何有效管理单个对话的即时、临时信息以及随着时间积累的大量、持久知识。

**原因：**标准化的解决方案是实施一个双组件内存系统，该系统区分短期和长期存储。短期的上下文记忆在大语言模型（LLM）的上下文窗口内保存最近的交互数据，以维持对话的流畅性。对于必须持久保存的信息，长期记忆解决方案使用外部数据库（通常是向量存储）进行高效的语义检索。像谷歌ADK这样的智能体框架提供了管理这一功能的特定组件，如用于对话线程的会话（Session）和用于临时数据的状态（State）。专门的内存服务（MemoryService）用于与长期知识库交互，使智能体能够检索相关的过往信息并将其融入当前上下文。

**经验法则：**当智能体需要做的不仅仅是回答单个问题时，请使用此模式。对于必须在整个对话中保持上下文、跟踪多步骤任务的进度，或通过回忆用户偏好和历史来个性化交互的智能体来说，这一点至关重要。只要期望智能体根据过去的成功、失败或新获取的信息进行学习或调整，就应实施内存管理。

**可视化总结**

图1：内存管理设计模式

**要点总结**

为快速总结内存管理的要点：

* 记忆对于智能体追踪事物、学习和个性化交互来说非常重要。
* 对话式AI既依赖短期记忆来处理单个聊天中的即时上下文，也依赖长期记忆来存储跨多个会话的持久知识。
* 短期记忆（即时信息）是暂时的，通常受大语言模型（LLM）的上下文窗口或框架传递上下文的方式限制。
* 长期记忆（留存的信息）使用向量数据库等外部存储在不同对话中保存信息，并通过搜索进行访问。
* 像ADK这样的框架有特定的组件，如会话（聊天线程）、状态（临时聊天数据）和内存服务（可搜索的长期知识）来管理内存。
* ADK的会话服务处理聊天会话的整个生命周期，包括其历史记录（事件）和临时数据（状态）。
* ADK的session.state是一个用于临时聊天数据的字典。前缀（user:、app:、temp:）告诉你数据所属的位置以及是否会保留。
* 在ADK中，添加事件时应使用EventActions.state\_delta或output\_key来更新状态，而不是直接更改状态字典。
* ADK的内存服务用于将信息存入长期存储，并允许代理（通常使用工具）对其进行搜索。
* LangChain提供了诸如ConversationBufferMemory之类的实用工具，可自动将单个对话的历史记录注入提示中，使智能体能够回忆起即时上下文。
* LangGraph通过使用存储来保存和检索语义事实、情景体验，甚至不同用户会话间可更新的程序规则，从而实现高级的长期记忆。
* 记忆库是一项托管服务，它通过自动提取、存储和调用用户特定信息，为智能体提供持久的长期记忆，从而实现跨框架（如谷歌的ADK、LangGraph和CrewAI）的个性化、持续性对话。

**结论**

本章深入探讨了代理系统内存管理这一非常重要的工作，展示了短期上下文与长期留存的知识之间的区别。我们讨论了这些类型的内存是如何设置的，以及在构建能够记忆事物的更智能代理时，它们在哪些地方得到应用。我们详细研究了谷歌ADK如何为你提供诸如会话、状态和内存服务等特定组件来处理这一问题。既然我们已经介绍了代理如何进行短期和长期记忆，接下来就可以探讨它们如何学习和适应。下一个模式“学习与适应”涉及代理基于新的经验或数据改变其思维方式、行为方式或知识储备。

**参考文献**

1. ADK内存<https://google.github.io/adk-docs/sessions/memory/>
2. LangGraph 内存<https://langchain-ai.github.io/langgraph/concepts/memory/>
3. Vertex AI代理引擎内存库<https://cloud.google.com/blog/products/ai-machine-learning/vertex-ai-memory-bank-in-public-preview>