adk-python Tutorial

This chapter introduces the fundamental building block of the `adk-python` project: the **Agent**. Think of building a complex AI system like constructing a large factory. Agents are the individual workers in this factory, each responsible for a specific task. By combining different agents, you can create sophisticated and powerful workflows.

Let's start with a simple use case: imagine you need an AI system that summarizes news articles. We could design an agent specifically for this job.

What is an Agent?

An Agent in `adk-python` represents a single AI entity capable of performing a specific task. It's a self-contained unit that receives input, processes it, and produces output. Different types of agents exist, each with its own capabilities. For our news summarization example, we might create an "ArticleSummarizerAgent".

Key Concepts

The core functionality of an Agent is encapsulated in its `run_async` method. This method handles the execution of the agent's task. We will discuss the different types of agents and their internal implementation later in this chapter.

Our First Agent: A Simple Example

Let's create a very basic agent using the `BaseAgent` class. While this agent doesn't do much, it provides a foundational understanding.

```
from google.adk.agents.base_agent import BaseAgent
from google.adk.events.event import Event

class MyFirstAgent(BaseAgent):
    name = "my_first_agent"
    description = "This is my first agent!"

    async def _run_async_impl(self, ctx): # This is where the agent's logic goes
        yield Event(author=self.name, content="Hello, world!") # Output an event

# Example usage (simplified):
agent = MyFirstAgent()
async def run_agent():
  async for event in agent.run_async(InvocationContext()): # InvocationContext will be disc
    print(event.content.parts[0].text)

asyncio.run(run_agent())
```

This code defines a simple agent named "my_first_agent". The `_run_async_impl` method is where the core logic resides. In this case, it simply yields a single event containing the text "Hello, world!". The `run_async` method is used to execute the agent. We will delve deeper into `InvocationContext` and the mechanics of the `run_async` method shortly.

Under the Hood: How Agents Work

Let's visualize the process of running an agent:

```
sequenceDiagram
    participant User
    participant Agent
    participant Event Queue

    User->>Agent: Sends request
    activate Agent
    Agent->>Event Queue: Generates Event
    Event Queue->>Agent: Event processed
    deactivate Agent
    Agent->>User: Returns result
```

When a request is sent to the Agent, it generates one or more events that are added to the Event Queue. These events represent the progression of the agent's task. The Agent then returns the result to the user. The exact implementation of event generation depends on the type of agent and its specific functionality.

We will see more complex agent examples involving multiple events and interactions with other components in later chapters. For now, this is a sufficient overview of Agent execution.

Different Types of Agents

The `adk-python` library provides several types of agents, including:

- `LlmAgent`: Agents that utilize Large Language Models (LLM Request/Response).
- `LoopAgent`: Agents that repeatedly execute sub-agents (Session).
- `ParallelAgent`: Agents that run sub-agents concurrently.
- `SequentialAgent`: Agents that run sub-agents sequentially.

Each of these agent types offers different ways to combine and orchestrate AI tasks. We'll explore these in detail in subsequent chapters.

Conclusion

This chapter introduced the core concept of an Agent in `adk-python`. We saw how Agents are the fundamental building blocks of complex AI systems and how

a simple agent can be created and executed. In the next chapter, we'll explore the concept of Tools, which are crucial for extending the capabilities of Agents.

In the previous chapter, we learned about Agents, the individual workers in our AI factory. But what if our agents need specialized tools to do their jobs? That's where the concept of a "Tool" comes in.

Imagine our news summarization agent from Chapter 1. It needs to *access* news articles to summarize them. It can't do that on its own; it needs a tool to fetch articles from a website or a news API. That's exactly what a tool provides: specific functionality to extend an agent's capabilities.

What is a Tool?

A Tool in `adk-python` is a piece of code that performs a specific task. It's like a specialized tool in a toolbox. Each tool is designed for a single job, such as accessing an external service (like a website), performing calculations, or interacting with a database. Agents use these tools to complete their work, making them far more versatile and powerful.

Key Concepts

The core of a tool is its `run_async` method. This method executes the tool's task and returns a result. Tools also have a `name` and a `description` which tell the agent what the tool does.

Let's create a simple tool that adds two numbers.

```
from google.adk.tools.base_tool import BaseTool

class AdderTool(BaseTool):
  name = "adder"
  description = "Adds two numbers together."

  async def run_async(self, *, args, tool_context): # This is where the tool does its work
    a = args['a']
    b = args['b']
    return a + b

# Example usage (simplified):
tool = AdderTool()
result = asyncio.run(tool.run_async(args={'a': 5, 'b': 3}, tool_context=None))
print(f"Result: {result}") # Output: Result: 8
```

This code defines a tool called `adder`. The `run_async` method takes a dictionary `args` containing the numbers to add. The `tool_context` is where you'd typically access external resources if needed. It returns the sum of the numbers.

This simple example demonstrates the fundamental structure of a tool. More complex tools might interact with external APIs or databases, making them much more powerful.

Under the Hood: How Tools Work

Let's visualize the interaction between an agent and a tool:

```
sequenceDiagram
    participant Agent
    participant Tool
    participant Result

    Agent->>Tool: Sends request with arguments
    activate Tool
    Tool->>Result: Performs calculation/action
    activate Result
    Tool<-<Result: Receives result
    deactivate Result
    Tool->>Agent: Returns result
    deactivate Tool
```

The agent sends a request (with arguments) to the tool. The tool performs its task and returns the result to the agent.

Different Types of Tools

The `adk-python` library provides many different types of tools, and you can even create your own. Some examples include:

- `FunctionTool`: Wraps a standard Python function.
- `APIHubToolset`: Generates tools from API Hub resources.
- `GoogleSearchTool`: A built-in tool to use Google Search (Gemini-specific).

We will explore more complex tool examples in later chapters.

Conclusion

This chapter introduced the concept of a Tool in `adk-python`. We learned how tools provide specialized functionality to agents, extending their capabilities. We built a simple `AdderTool` and discussed the underlying mechanisms of tool execution. In the next chapter, we'll explore the concept of Sessions, which manage the interactions between agents and tools over time.

In the previous chapter, we learned about Tools, which extend the capabilities of our Agents. But how do we manage a conversation, or a series of interactions, between a user and multiple agents using these tools? That's where the concept of a "Session" comes in.

Imagine you're building a chatbot that helps users plan a trip. The user might ask about flights, hotels, and activities, all within the same conversation. A session keeps track of this entire interaction, storing all the messages and actions, so each agent has the necessary context.

What is a Session?

A Session in `adk-python` represents a single conversation between a user and agents. Think of it as a chat log that remembers everything said and done during the interaction. This "memory" persists across multiple calls to different agents. This ensures that each agent has the full context of the conversation so far, leading to a more coherent and helpful AI system.

Key Concepts

The core of a session is its state and its list of events.

Session State: This is a dictionary that stores information about the conversation. For example, in our trip-planning chatbot, the session state might store the user's desired destination, travel dates, budget, and any other preferences they've expressed. This information is updated as the conversation progresses, providing valuable context to subsequent agents.

Session Events: A session also records every event that happens during the conversation. An event might represent a user message, an agent's response, the result of a tool's execution, or an error. This chronological log provides a complete history of the interaction.

Using Sessions: A Simple Example

Let's create a simple session and add some events to it. We'll use the `InMemorySessionService` for simplicity.

```
from google.adk.sessions import InMemorySessionService, Session, Event

session_service = InMemorySessionService()
session = session_service.create_session(app_name="my_app", user_id="user123")

# Add an event representing a user message
user_message = Event(author="user123", content={"text": "Plan a trip to Paris."})
session = session_service.append_event(session, user_message)

#Add an event representing an agent response
agent_response = Event(author="agent1", content={"text": "Okay, I'm planning a trip to Paris
session = session_service.append_event(session, agent_response)

print(session.events) # This will print the list of events added to the session
print(session.state) # This will print the state of the session, which will be empty in this
```

This code first creates a session using `InMemorySessionService`. Then, it adds two events: one for a user message and one for an agent's response. Finally, it prints the list of events and the session state.

Under the Hood: How Sessions Work

Let's visualize the process:

```
sequenceDiagram
    participant User
    participant Agent 1
    participant Agent 2
    participant Session Service
    participant Session

    User->>Session Service: Starts Session
    activate Session Service
    Session Service->>Session: Creates Session
    deactivate Session Service
    User->>Agent 1: Sends message
    activate Agent 1
    Agent 1->>Session Service: Adds Event
    activate Session Service
    Session Service->>Session: Updates Session with Event
    deactivate Session Service
    deactivate Agent 1
    Agent 2->>Session Service: Queries Session (Gets Context)
    activate Session Service
    Session Service->>Agent 2: Returns Session data
    deactivate Session Service
```

The `Session Service` (which could be `InMemorySessionService`, `DatabaseSessionService`, or `VertexAiSessionService`) manages the creation and updating of sessions. Agents interact with the `Session Service` to add events (like user messages or agent responses) and retrieve the session's state and events for context.

The core implementation is in `src/google/adk/sessions/base_session_service.py` and its subclasses, such as `src/google/adk/sessions/in_memory_session_service.py` for in-memory storage, `src/google/adk/sessions/database_session_service.py` for database persistence, and `src/google/adk/sessions/vertex_ai_session_service.py` for using Vertex AI's managed session service. These files contain the logic for creating, retrieving, updating, and deleting sessions. The example above uses `InMemorySessionService`, a simple in-memory implementation suitable for experimentation.

Conclusion

This chapter introduced the `Session` concept in `adk-python`, a crucial component for managing conversations with multiple agents. We've seen how sessions track events and state, providing crucial context for ongoing interactions. In the next chapter, we'll delve into LLM Request/Response, focusing on how Large Language Models (LLMs) interact within the session context.

In the previous chapter, we learned about Sessions, which help manage conversations between users and agents. But how do agents actually talk to a Large Language Model (LLM)? That's where `LlmRequest` and `LlmResponse` come in. These are like the "envelopes" for sending information to and receiving information from the LLM.

Let's imagine you're building a chatbot that answers questions about history. Your agent needs to ask the LLM for information, and the LLM needs to send its answer back in a way that the agent can easily understand. This is exactly what `LlmRequest` and `LlmResponse` handle.

Key Concepts: `LlmRequest` and `LlmResponse`

We'll focus on two key classes:

1. `LlmRequest`: This class packages all the information needed to send a request to the LLM. Think of it as the envelope containing your question and any other relevant details. These details might include:

```
* **`model`:** The name of the LLM you want to use (e.g., "gemini-pro").
* **`contents`:**  The actual message you're sending to the LLM. This is usually a list of m
* **`config`:**  Additional settings for the LLM, such as temperature (how creative the resp
```

2. `LlmResponse`: This class unpacks the response from the LLM. It's the "reply envelope" containing the LLM's answer. The `LlmResponse` may include:

```
* **`content`:** The LLM's response, usually text.
* **`error_code` and `error_message`:** Information about any errors that occurred during th
```

Example: Asking the LLM a History Question

Let's build a simple example where an agent asks the LLM about the first President of the United States. We'll use a simplified `Gemini` LLM (from `google.adk.models.google_llm`). Don't worry about the specifics of `Gemini` for now; just think of it as a way to interact with an LLM.

First, let's create the `LlmRequest`:

```
from google.adk.models.google_llm import Gemini
from google.adk.models.llm_request import LlmRequest
from google.genai import types
```

```
# Simplified example - many details are omitted for clarity
llm = Gemini(model="gemini-pro") # Replace with your actual LLM
request = LlmRequest(
    model="gemini-pro", #specify model
    contents=[types.Content(role="user", parts=[types.Part.from_text("Who was the first pres
    config=types.GenerateContentConfig(), # Additional settings (optional)
)
```

This code creates an `LlmRequest` object. We provide the model name, the user's question, and an empty config for now.

Next, we'll send the request to the LLM and get the response:

```
async def get_llm_response(llm, request):
  async for response in llm.generate_content_async(request):
    return response # Return the first response

response = asyncio.run(get_llm_response(llm, request))
print(response.content.parts[0].text) # Print the LLM's answer
```

This code snippet uses the `generate_content_async` method of our simplified `Gemini` LLM to get the response. It then prints out the response text. The actual output will depend on the LLM, but you would expect something like "George Washington".

Under the Hood: LLM Interaction

Here's a simplified view of how the LLM interaction works:

```
sequenceDiagram
    participant Agent
    participant LlmRequest
    participant LLM
    participant LlmResponse
    participant Agent Output

    Agent->>LlmRequest: Creates request with question
    activate LlmRequest
    LlmRequest->>LLM: Sends request
    activate LLM
    LLM->>LlmResponse: Generates response
    deactivate LLM
    activate LlmResponse
    LlmResponse->>Agent: Returns response
    deactivate LlmResponse
    Agent->>Agent Output: Processes response and outputs the answer
    deactivate Agent
```

The Agent creates an `LlmRequest` object, which is then sent to the LLM. The

LLM processes the request and returns an `LlmResponse`. The Agent then receives and processes this response.

The core implementation for `LlmRequest` and `LlmResponse` is in `src/google/adk/models/llm_request.py` and `src/google/adk/models/llm_response.py` respectively. These files contain detailed code for handling LLM requests and responses, including error handling and various LLM configurations. The simplified examples above omit many details for clarity.

Conclusion

This chapter introduced the `LlmRequest` and `LlmResponse` classes, which simplify communication with LLMs. We built a simple example showing how to ask an LLM a question and retrieve its response. In the next chapter, we will look at LLM Flow, which helps manage the sequence of interactions with LLMs within a complex agent system.

In the previous chapter, we learned how to send requests to and receive responses from a Large Language Model (LLM Request/Response). But in real-world scenarios, interacting with an LLM is rarely a single, simple request-response cycle. Often, agents need to interact with the LLM multiple times, perhaps using the LLM's output to guide subsequent requests or to control the flow of the conversation. This is where the concept of "LLM Flow" comes in.

LLM Flow defines the control flow for how an agent interacts with an LLM. It's like a recipe that dictates the steps involved in getting a response from the LLM, handling tools (Tool), and processing the response. It manages the entire conversation with the LLM, ensuring a smooth and efficient exchange of information.

A Simple Use Case: Summarizing a News Article

Let's say we want to build an agent that summarizes news articles. This agent will need to:

1. Fetch the article content (using a "FetchArticleTool").
2. Send the content to the LLM with a prompt asking for a summary.
3. Receive the summary from the LLM.
4. Return the summary as the final output.

A simple LLM Flow ensures these steps happen in the correct order, and handles any potential errors along the way.

Key Concepts: LLM Flow Types

The `adk-python` library provides several pre-built LLM flow types to manage different interaction patterns with LLMs. Here are some of the most common ones:

- **`SingleFlow`:** This is the simplest flow. It sends a single request to the LLM and receives a single response. This is suitable for simple tasks where only one LLM interaction is needed.
- **`AutoFlow`:** This flow automatically manages multiple LLM calls, potentially transferring control to other agents based on the LLM's response. It's more sophisticated and suitable for complex tasks involving multiple steps.

These flows handle the complexities of interacting with the LLM, including preprocessing the request (e.g., formatting the prompt), sending the request, receiving the response, and post-processing the response (e.g., extracting relevant information).

Using `SingleFlow`: Our News Summarizer Agent

Let's create a simplified agent using `SingleFlow` for our news summarization use case. We'll omit many details for clarity.

```
#Simplified Example
from google.adk.flows.llm_flows import SingleFlow
from google.adk.agents.base_agent import BaseAgent
from google.adk.models.google_llm import Gemini #Simplified LLM interaction
from google.adk.models.llm_request import LlmRequest
from google.genai import types

class NewsSummarizerAgent(BaseAgent):
    # ... (Agent definition, including tools) ...
    flow = SingleFlow()

    async def _run_async_impl(self, ctx):
        article_content = await self.tools['fetch_article'].run_async(...) # Fetch article
        prompt = f"Summarize this article:\n{article_content}"
        request = LlmRequest(model="gemini-pro", contents=[types.Content(role="user", parts=
        async for event in self.flow.run_async(ctx): # Run the flow
          yield event #Yield the summary event
```

This code shows a simplified version. The `SingleFlow` handles the interaction with the LLM. The agent fetches the article content using a tool, creates an `LlmRequest`, and then lets the `SingleFlow` manage the interaction with the LLM. The flow's `run_async` method handles sending the request and processing the response, yielding an event that contains the summary.

Under the Hood: `SingleFlow`

Let's visualize how `SingleFlow` works:

```
sequenceDiagram
```

```
participant Agent
participant SingleFlow
participant LLM
participant LlmRequest
participant LlmResponse

Agent->>SingleFlow: Sends LlmRequest
activate SingleFlow
SingleFlow->>LLM: Sends LlmRequest to LLM
activate LLM
LLM->>SingleFlow: Returns LlmResponse
deactivate LLM
SingleFlow->>Agent: Returns processed LlmResponse as Event
deactivate SingleFlow
```

The agent sends the LLM request to `SingleFlow`. `SingleFlow` then sends it to the LLM. The LLM returns a response, which `SingleFlow` processes and converts into an event before sending back to the agent.

The core implementation of `SingleFlow` is in `src/google/adk/flows/llm_flows/single_flow.py`. It contains the `run_async` method that handles the LLM interaction. The code is more complex than shown above, including error handling, pre- and post-processing of LLM requests and responses, but the core idea is the same.

Conclusion

This chapter introduced the concept of LLM Flow, which manages complex interactions between agents and LLMs. We saw how different flow types, like `SingleFlow`, can simplify the process of building sophisticated AI systems. In the next chapter, we'll explore Runners, which orchestrate the execution of agents and their flows.

In the previous chapter, we learned about LLM Flow, which manages the interactions between our agents and Large Language Models (LLMs). But how do we bring everything together? How do we start and manage the execution of our agents, their tools, and their LLM interactions within a single user request? That's where the `Runner` comes in.

Imagine you're a chef preparing a complex meal. You have various agents (like sous chefs) each responsible for a different part of the dish (e.g., preparing the vegetables, cooking the meat). The `Runner` is like the head chef, orchestrating the entire process, ensuring that each agent performs its task in the correct order and at the right time, coordinating the use of tools (like kitchen appliances) and managing the overall flow of the cooking process.

What is a Runner?

A `Runner` in `adk-python` is the orchestrator of agent execution. It manages the entire lifecycle of processing a single user request, coordinating agents, tools, external services (like memory and artifacts), and LLM interactions. It keeps track of the session, ensuring that each agent has the necessary context to do its job.

Key Concepts: The Runner's Role

The `Runner` plays several crucial roles:

1. **Agent Initialization:** The `Runner` is responsible for creating and initializing the root agent and its sub-agents before starting execution.
2. **Session Management:** The `Runner` manages the session – the conversation history and state – providing this context to all participating agents and tools.
3. **Event Handling:** It receives events from agents, tools, and the LLM, managing their flow and ensuring that the appropriate actions are taken.
4. **Resource Management:** It interacts with memory and artifact services to store and retrieve data, allowing agents to share information and maintain context across multiple interactions.
5. **Flow Orchestration:** The `Runner` executes the LLM flow for each agent, facilitating communication with the LLM.
6. **Error Handling:** It handles any errors that may occur during the execution process, reporting them appropriately.

Example: Using a Runner to Answer a Question

Let's build a simple example using the `InMemoryRunner`. This uses in-memory services, which makes it lightweight and ideal for testing. We'll create a simple agent that answers a question using an LLM.

```
# Simplified Example (InMemoryRunner)
from google.adk.runners import InMemoryRunner
from google.adk.agents.llm_agent import LlmAgent #Simplified LLM Agent
from google.adk.models.google_llm import Gemini #Simplified LLM interaction

#Simplified LLM Agent
class QuestionAnswererAgent(LlmAgent):
    name = "question_answerer"
    #...other parts of the agent definition

runner = InMemoryRunner(QuestionAnswererAgent())
response = runner.run(user_id="test_user", session_id="123", new_message={"text":"What is th
```

```
for event in response:
  print(event.content.parts[0].text) # Prints the LLM's answer
```

This code creates an `InMemoryRunner` with a simplified `LlmAgent`. The `run` method executes the agent, and the `for` loop iterates through the generated events.

This simplified example demonstrates how to execute an agent using a `Runner`. The actual LLM interaction and agent logic are omitted for brevity but will be covered in detail in other chapters.

Under the Hood: Runner Execution

Here's a simplified view of the execution flow:

```
sequenceDiagram
    participant User
    participant Runner
    participant Agent
    participant LLM
    participant Session

    User->>Runner: Submits request
    activate Runner
    Runner->>Session: Creates/Retrieves session
    activate Session
    Runner->>Agent: Starts agent execution
    activate Agent
    Agent->>LLM: Sends LLM request (via LLM Flow)
    activate LLM
    LLM->>Agent: Returns LLM response
    deactivate LLM
    Agent->>Runner: Returns events
    deactivate Agent
    Runner->>User: Sends response (Events)
    deactivate Runner
    deactivate Session
```

The `Runner` manages the entire process, starting with session creation and agent initialization. It coordinates the interaction between the agent, the LLM, and the session, ultimately producing a sequence of events that represent the system's response to the user's request.

The core implementation of `Runner` is in `src/google/adk/runners.py`. The `run_async` method contains the main logic for agent execution, including session management, event handling, and resource coordination. The simplified `InMemoryRunner` in the example above provides a simplified interface for testing and experimentation, using in-memory services for session, artifact, and

memory.

Conclusion

This chapter introduced the `Runner`, the core orchestrator of agent execution in `adk-python`. We've seen how it manages the complete lifecycle of a user request, coordinating agents, tools, and external services. In the next chapter, we'll explore Memory Services, which are crucial for maintaining context across multiple agent interactions.

In the previous chapter, we learned how the Runner orchestrates the execution of our agents. But what if our agents need to remember past interactions to provide better responses? This is where the "Memory Service" comes in.

Imagine building a chatbot that helps users plan a trip. The user might start by saying "I want to go to Paris". Later, they might ask "What's the weather like?". For the chatbot to answer accurately, it needs to remember the user's intended destination (Paris) from the earlier interaction. This is the role of the Memory Service – to provide access to previous interactions within a conversation or to external knowledge.

What is a Memory Service?

A Memory Service is like an agent's memory. It stores past interactions or relevant information (like the user's trip destination) to inform future responses, enabling the agent to learn and maintain context throughout a conversation. It's crucial for creating AI systems that feel more natural and helpful because it remembers important details.

Key Concepts

The Memory Service has two core functionalities:

1. **Adding Sessions to Memory:** After a conversation (a Session) ends, the entire conversation history is added to the Memory Service for future reference.
2. **Searching Memory:** When a new user request comes in, the Memory Service can search its stored conversations to find relevant information that might be helpful in responding. This allows the agent to use previous conversations to improve its current response.

Using the Memory Service: A Simple Example

Let's use the `InMemoryMemoryService` (a simple in-memory implementation for testing) to demonstrate. We'll add a sample conversation to memory and then search for information about Paris.

First, let's create a sample session:

```
from google.adk.sessions import Session, Event
from google.adk.memory import InMemoryMemoryService

memory_service = InMemoryMemoryService()
session = Session(app_name="travel_planner", user_id="user123", id="session456")
session.events.append(Event(author="user123", content={"text": "I want to go to Paris."}))
session.events.append(Event(author="agent1", content={"text": "Okay, I'm planning a trip to
```

This code creates a session with two events: a user request and an agent response.

Next, we add the session to memory:

```
memory_service.add_session_to_memory(session)
```

Finally, we search the memory for information about Paris:

```
result = memory_service.search_memory(app_name="travel_planner", user_id="user123", query="P
print(result.memories) # Prints a list of matching memories (sessions in this case)
```

This will print a list of sessions that contain "Paris" in their conversation history. For our simple example, it would contain the `session` we added.

Under the Hood: Memory Service Implementation

Let's visualize a simplified workflow:

```
sequenceDiagram
    participant User
    participant Agent
    participant Runner
    participant Memory Service
    participant Session

    User->>Agent: "I want to go to Paris"
    activate Agent
    Agent->>Runner: Processes request
    activate Runner
    Runner->>Session: Updates session
    activate Session
    Session->>Runner: Session complete
    deactivate Session
    Runner->>Memory Service: Adds session to memory
    activate Memory Service
    Memory Service->>Runner: Session added
    deactivate Memory Service
    deactivate Runner
    User->>Agent: "What's the weather like?"
    activate Agent
```

```
Agent->>Runner: Processes request
activate Runner
Runner->>Memory Service: Searches for "Paris"
activate Memory Service
Memory Service->>Runner: Returns relevant session data
deactivate Memory Service
Runner->>Agent: Provides Paris context
deactivate Runner
Agent->>User: Answers based on context
deactivate Agent
```

The core implementation of `InMemoryMemoryService` is in `src/google/adk/memory/in_memory_memory_servi` It uses a simple in-memory dictionary to store sessions, making it suitable for simple testing and prototyping. More robust implementations, such as `VertexAiRagMemoryService` (`src/google/adk/memory/vertex_ai_rag_memory_service.py`), might use external databases or vector databases for more scalable and efficient storage and retrieval. These utilize more advanced search techniques for better results.

Conclusion

This chapter introduced the Memory Service, a vital component for creating context-aware AI systems. We saw how it enables agents to maintain context across multiple interactions, making the AI more natural and helpful. In the next chapter, we'll discuss Artifact Services, another important component for managing and sharing data within an agent system.

In the previous chapter, we learned how the Runner manages the execution of our agents. But what if our agents need to work with files? What if an agent needs to receive a document from a user, process it, and then return a modified version? This is where the "Artifact Service" comes in.

Imagine you're building an AI system that edits documents. A user uploads a document, the system processes it, and returns the edited version. The Artifact Service manages the storage and retrieval of these files—it's the file cabinet for our AI system.

What is an Artifact Service?

The Artifact Service manages the storage and retrieval of files associated with a session (Session). Think of it as a central file repository. Agents use it to save and load files, like documents or images, during a conversation. This allows agents to handle file uploads and downloads, greatly extending their capabilities beyond just text-based interactions.

Key Concepts

The Artifact Service has two main functions:

1. **Saving Artifacts:** Agents use this to store files. The service keeps track of which files belong to which session.
2. **Loading Artifacts:** Agents use this to retrieve files previously saved. This ensures that the agent has access to the correct file and version from the past.

Using the Artifact Service: A Simple Example

Let's build a super-simplified example using the `InMemoryArtifactService` (an in-memory version, perfect for testing). We'll save a simple text file and then load it back.

First, let's save a file:

```
from google.adk.artifacts import InMemoryArtifactService
from google.genai import types

artifact_service = InMemoryArtifactService()
file_content = "This is my document."
artifact = types.Part.from_text(file_content)

saved_version = artifact_service.save_artifact(
    app_name="my_app", user_id="user123", session_id="session456", filename="my_doc.txt", ar
)
print(f"Saved version: {saved_version}") # Output: Saved version: 0
```

This code creates an `InMemoryArtifactService`, creates a `types.Part` object representing our document, and saves it using `save_artifact`. The `save_artifact` function returns the version number (0 for the first version).

Now, let's load it back:

```
loaded_artifact = artifact_service.load_artifact(
    app_name="my_app", user_id="user123", session_id="session456", filename="my_doc.txt"
)
print(f"Loaded content: {loaded_artifact.text}") # Output: Loaded content: This is my docume
```

This retrieves the file using `load_artifact`. We use the `text` attribute to get the text content.

Under the Hood: Artifact Service Implementation

Here's a simplified view of how the Artifact Service works:

```
sequenceDiagram
    participant Agent
    participant Artifact Service
    participant Storage
    participant File
```

```
Agent->>Artifact Service: Save File
activate Artifact Service
Artifact Service->>Storage: Stores File
activate Storage
Storage-->>Artifact Service: Confirmation
deactivate Storage
Artifact Service->>Agent: Version number
deactivate Artifact Service
Agent->>Artifact Service: Load File (version 0)
activate Artifact Service
Artifact Service->>Storage: Retrieves File (version 0)
activate Storage
Storage-->>Artifact Service: File
deactivate Storage
Artifact Service->>Agent: File
deactivate Artifact Service
```

The `InMemoryArtifactService` stores files in a simple dictionary in memory. For a real-world application, you would use a more robust service like `GcsArtifactService` which stores files in Google Cloud Storage. The implementation details for `GcsArtifactService` are within `src/google/adk/artifacts/gcs_artifact_service.py`.

Conclusion

This chapter introduced the Artifact Service, a crucial component for managing files within our AI system. We've seen how it allows agents to interact with files, enabling more complex and versatile applications. In the next chapter, we'll explore Events, which are fundamental for communication and tracking progress within the agent framework.

In the previous chapter, we learned about Artifact Services, which manage files for our agents. But how do agents communicate with each other and with the user? How do we track the progress of a conversation? This is where the "Event" concept comes in.

Imagine you're building a simple chatbot. The user asks a question, the agent processes it, and then returns an answer. Each of these actions (user question, agent processing, agent answer) are all considered individual *events*. The sequence of events forms the entire conversation history.

What is an Event?

An event in `adk-python` is a single message or action within a session. Think of it as a single entry in a conversation log. Every message sent or action taken (like fetching a file using an Artifact Service or calling an LLM using an LLM

Flow) creates a new event. Events are the building blocks that track the entire conversation flow.

Key Concepts: Understanding Events

An event is a simple data structure containing important information:

- **`author`:** Who generated the event? This could be "user", or the name of an agent.
- **`content`:** The actual data of the event. This could be text from a user message, the result of a tool's execution, or the response from an LLM.
- **`id`:** A unique identifier for the event. This is automatically generated.
- **`timestamp`:** The time the event was created.

Creating and Using Events: A Simple Example

Let's create a simple event representing a user message:

```
from google.adk.events.event import Event
from google.genai import types

user_message = Event(author="user", content=types.Content(parts=[types.Part.from_text("Hello
print(user_message.author) # Output: user
print(user_message.content.parts[0].text) # Output: Hello, chatbot!
```

This code creates an `Event` object. The `author` is "user", and the `content` is the user's message. The `id` and `timestamp` are automatically created.

Now let's create an event representing an agent's response:

```
agent_response = Event(author="my_agent", content=types.Content(parts=[types.Part.from_text(
print(agent_response.author) # Output: my_agent
print(agent_response.content.parts[0].text) # Output: Hello there!
```

This creates another `Event` representing the agent's response.

Under the Hood: Event Generation and Flow

Here's how events flow within the `adk-python` system:

```
sequenceDiagram
    participant User
    participant Agent
    participant Runner
    participant Event Queue
    participant Session

    User->>Agent: Sends message
    activate Agent
```

```
    Agent->>Runner: Processes message
    activate Runner
    Runner->>Agent: Generates Event
    activate Agent
    Agent->>Event Queue: Adds Event
    activate Event Queue
    Event Queue->>Session: Appends Event
    activate Session
    Session->>Event Queue: Event saved
    deactivate Session
    deactivate Event Queue
    Agent->>Runner: Event Sent
    deactivate Agent
    Runner->>User: Returns Event
    deactivate Runner
```

The user sends a message to the agent. The agent, through the Runner, generates an event and adds it to an event queue which is ultimately added to the Session. This sequence captures the entire conversation history.

The core implementation of the `Event` class is in `src/google/adk/events/event.py`. This file contains the detailed logic for creating, handling, and storing event data.

Conclusion

This chapter introduced the concept of "Event" in `adk-python`, a fundamental building block for tracking conversations and managing interactions within the agent framework. We learned how events capture individual actions and messages, forming a complete history of the conversation. In the next chapter, we will explore how to integrate all the components we've discussed into a complete AI application. (There is no next chapter in the provided structure.)

This Python project, `adk-python`, is an **Agent Development Kit (ADK)** that simplifies building complex AI applications. It provides abstractions for common AI components like *agents*, *tools*, and *LLMs*, allowing developers to easily create and manage interactions. The ADK handles the *orchestration* of these components, including *memory management* and *artifact storage*, enabling the creation of sophisticated AI workflows.

Source Repository: https://github.com/google/adk-python

```
flowchart TD
    A0["Agent"]
    A1["Tool"]
    A2["Session"]
    A3["Event"]
    A4["LLM Request/Response"]
```

```
A5["Runner"]
A6["LLM Flow"]
A7["Memory Service"]
A8["Artifact Service"]
A0 -- "Uses" --> A1
A0 -- "Accesses" --> A7
A0 -- "Accesses" --> A8
A0 -- "Contributes to" --> A2
A0 -- "Generates" --> A3
A0 -- "Uses" --> A6
A5 -- "Manages" --> A0
A5 -- "Manages" --> A2
A5 -- "Uses" --> A7
A5 -- "Uses" --> A8
A6 -- "Handles" --> A4
A2 -- "Contains" --> A3
A1 -- "Contributes to" --> A4
A4 -- "Sent to" --> A0
```

Chapters

1. Agent
2. Tool
3. Session
4. LLM Request/Response
5. LLM Flow
6. Runner
7. Memory Service
8. Artifact Service
9. Event