Build Your First Intelligent Agent Team: A Progressive Weather Bot with ADK

Open in Colab

Share to:

This tutorial extends from the Quickstart example for Agent Development Kit. Now, you're ready to dive deeper and construct a more sophisticated, **multi-agent system**.

We'll embark on building a **Weather Bot agent team**, progressively layering advanced features onto a simple foundation. Starting with a single agent that can look up weather, we will incrementally add capabilities like:

- Leveraging different AI models (Gemini, GPT, Claude).
- Designing specialized sub-agents for distinct tasks (like greetings and farewells).
- Enabling intelligent delegation between agents.
- Giving agents memory using persistent session state.
- Implementing crucial safety guardrails using callbacks.

Why a Weather Bot Team?

This use case, while seemingly simple, provides a practical and relatable canvas to explore core ADK concepts essential for building complex, real-world agentic applications. You'll learn how to structure interactions, manage state, ensure safety, and orchestrate multiple AI "brains" working together.

What is ADK Again?

As a reminder, ADK is a Python framework designed to streamline the development of applications powered by Large Language Models (LLMs). It offers robust building blocks for creating agents that can reason, plan, utilize tools, interact dynamically with users, and collaborate effectively within a team.

In this advanced tutorial, you will master:

- ✅ **Tool Definition & Usage:** Crafting Python functions (`tools`) that grant agents specific abilities (like fetching data) and instructing agents on how to use them effectively.
- ✅ **Multi-LLM Flexibility:** Configuring agents to utilize various leading LLMs (Gemini, GPT-4o, Claude Sonnet) via LiteLLM integration, allowing you to choose the best model for each task.
- ✅ **Agent Delegation & Collaboration:** Designing specialized sub-agents and enabling automatic routing (`auto flow`) of user requests to the most appropriate agent within a team.
- ✅ **Session State for Memory:** Utilizing `Session State` and `ToolContext` to enable agents to remember information across conversational turns, leading to more contextual interactions.
- ✅ **Safety Guardrails with Callbacks:** Implementing `before_model_callback` and `before_tool_callback` to inspect, modify, or block requests/tool usage based on predefined rules, enhancing application safety and control.

End State Expectation:

By completing this tutorial, you will have built a functional multi-agent Weather Bot system. This system will not only provide weather information but also handle conversational niceties, remember the last city checked, and operate within defined safety boundaries, all orchestrated using ADK.

Prerequisites:

- ✅ **Solid understanding of Python programming.**
- ✅ **Familiarity with Large Language Models (LLMs), APIs, and the concept of agents.**
- ❗ **Crucially: Completion of the ADK Quickstart tutorial(s) or equivalent foundational knowledge of ADK basics (Agent, Runner, SessionService, basic Tool usage).** This tutorial builds directly upon those concepts.
- ✅ **API Keys** for the LLMs you intend to use (e.g., Google AI Studio for Gemini, OpenAI Platform, Anthropic Console).

Note on Execution Environment:

This tutorial is structured for interactive notebook environments like Google Colab, Colab Enterprise, or Jupyter notebooks. Please keep the following in mind:

- **Running Async Code:** Notebook environments handle asynchronous code differently. You'll see examples using `await` (suitable when an event loop is already running, common in notebooks) or `asyncio.run()` (often needed when running as a standalone `.py` script or in specific notebook setups). The code blocks provide guidance for both scenarios.
- **Manual Runner/Session Setup:** The steps involve explicitly creating `Runner` and `SessionService` instances. This approach is shown because it gives you fine-grained control over the agent's execution lifecycle, session management, and state persistence.

Alternative: Using ADK's Built-in Tools (Web UI / CLI / API Server)

If you prefer a setup that handles the runner and session management automatically using ADK's standard tools, you can find the equivalent code structured for that purpose here. That version is designed to be run directly with commands like `adk web` (for a web UI), `adk run` (for CLI interaction), or `adk api_server` (to expose an API). Please follow the `README.md` instructions provided in that alternative resource.

Ready to build your agent team? Let's dive in!

> **Note:** This tutorial works with adk version 1.0.0 and above

```
# @title Step 0: Setup and Installation
# Install ADK and LiteLLM for multi-model support

!pip install google-adk -q
!pip install litellm -q

print("Installation complete.")
```

```
# @title Import necessary libraries
import os
import asyncio
from google.adk.agents import Agent
from google.adk.models.lite_llm import LiteLlm # For multi-
model support
```

```
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner
from google.genai import types # For creating message
Content/Parts

import warnings
# Ignore all warnings
warnings.filterwarnings("ignore")

import logging
logging.basicConfig(level=logging.ERROR)

print("Libraries imported.")
```

```
# @title Configure API Keys (Replace with your actual keys!)

# --- IMPORTANT: Replace placeholders with your real API keys -
--

# Gemini API Key (Get from Google AI Studio:
https://aistudio.google.com/app/apikey)
os.environ["GOOGLE_API_KEY"] = "YOUR_GOOGLE_API_KEY" # <---
REPLACE

# [Optional]
# OpenAI API Key (Get from OpenAI Platform:
https://platform.openai.com/api-keys)
os.environ['OPENAI_API_KEY'] = 'YOUR_OPENAI_API_KEY' # <---
REPLACE

# [Optional]
# Anthropic API Key (Get from Anthropic Console:
https://console.anthropic.com/settings/keys)
os.environ['ANTHROPIC_API_KEY'] = 'YOUR_ANTHROPIC_API_KEY' # <-
-- REPLACE

# --- Verify Keys (Optional Check) ---
print("API Keys Set:")
print(f"Google API Key set: {'Yes' if
os.environ.get('GOOGLE_API_KEY') and
os.environ['GOOGLE_API_KEY'] != 'YOUR_GOOGLE_API_KEY' else 'No
(REPLACE PLACEHOLDER!)'}")
print(f"OpenAI API Key set: {'Yes' if
os.environ.get('OPENAI_API_KEY') and
os.environ['OPENAI_API_KEY'] != 'YOUR_OPENAI_API_KEY' else 'No
(REPLACE PLACEHOLDER!)'}")
print(f"Anthropic API Key set: {'Yes' if
os.environ.get('ANTHROPIC_API_KEY') and
os.environ['ANTHROPIC_API_KEY'] != 'YOUR_ANTHROPIC_API_KEY'
else 'No (REPLACE PLACEHOLDER!)'}")
```

```
# Configure ADK to use API keys directly (not Vertex AI for
this multi-model setup)
os.environ["GOOGLE_GENAI_USE_VERTEXAI"] = "False"


# @markdown **Security Note:** It's best practice to manage API
keys securely (e.g., using Colab Secrets or environment
variables) rather than hardcoding them directly in the
notebook. Replace the placeholder strings above.
```

```
# --- Define Model Constants for easier use ---

# More supported models can be referenced here:
https://ai.google.dev/gemini-api/docs/models#model-variations
MODEL_GEMINI_2_0_FLASH = "gemini-2.0-flash"

# More supported models can be referenced here:
https://docs.litellm.ai/docs/providers/openai#openai-chat-
completion-models
MODEL_GPT_4O = "openai/gpt-4.1" # You can also try: gpt-4.1-
mini, gpt-4o etc.

# More supported models can be referenced here:
https://docs.litellm.ai/docs/providers/anthropic
MODEL_CLAUDE_SONNET = "anthropic/claude-sonnet-4-20250514" #
You can also try: claude-opus-4-20250514 , claude-3-7-sonnet-
20250219 etc

print("\nEnvironment configured.")
```

Step 1: Your First Agent - Basic Weather Lookup

Let's begin by building the fundamental component of our Weather Bot: a single agent capable of performing a specific task – looking up weather information. This involves creating two core pieces:

1. **A Tool:** A Python function that equips the agent with the *ability* to fetch weather data.
2. **An Agent:** The AI "brain" that understands the user's request, knows it has a weather tool, and decides when and how to use it.

1. Define the Tool (`get_weather`)

In ADK, **Tools** are the building blocks that give agents concrete capabilities beyond just text generation. They are typically regular Python functions that perform specific actions, like calling an API, querying a database, or performing calculations.

Our first tool will provide a *mock* weather report. This allows us to focus on the agent structure without needing external API keys yet. Later, you could easily swap this mock function with one that calls a real weather service.

Key Concept: Docstrings are Crucial! The agent's LLM relies heavily on the function's **docstring** to understand:

- *What* the tool does.
- *When* to use it.
- *What arguments* it requires (`city: str`).
- *What information* it returns.

Best Practice: Write clear, descriptive, and accurate docstrings for your tools. This is essential for the LLM to use the tool correctly.

```
# @title Define the get_weather Tool
def get_weather(city: str) -> dict:
    """Retrieves the current weather report for a specified
city.

    Args:
        city (str): The name of the city (e.g., "New York",
"London", "Tokyo").

    Returns:
        dict: A dictionary containing the weather information.
              Includes a 'status' key ('success' or 'error').
              If 'success', includes a 'report' key with
weather details.
              If 'error', includes an 'error_message' key.
    """
    print(f"--- Tool: get_weather called for city: {city} ---")
# Log tool execution
    city_normalized = city.lower().replace(" ", "") # Basic
normalization

    # Mock weather data
    mock_weather_db = {
        "newyork": {"status": "success", "report": "The weather
in New York is sunny with a temperature of 25°C."},
        "london": {"status": "success", "report": "It's cloudy
in London with a temperature of 15°C."},
        "tokyo": {"status": "success", "report": "Tokyo is
experiencing light rain and a temperature of 18°C."},
    }

    if city_normalized in mock_weather_db:
        return mock_weather_db[city_normalized]
    else:
        return {"status": "error", "error_message": f"Sorry, I
don't have weather information for '{city}'."}

# Example tool usage (optional test)
print(get_weather("New York"))
print(get_weather("Paris"))
```

---

2. Define the Agent (`weather_agent`)

Now, let's create the **Agent** itself. An `Agent` in ADK orchestrates the interaction between the user, the LLM, and the available tools.

We configure it with several key parameters:

- `name`: A unique identifier for this agent (e.g., "weather_agent_v1").
- `model`: Specifies which LLM to use (e.g., `MODEL_GEMINI_2_0_FLASH`). We'll start with a specific Gemini model.
- `description`: A concise summary of the agent's overall purpose. This becomes crucial later when other agents need to decide whether to delegate tasks to *this* agent.
- `instruction`: Detailed guidance for the LLM on how to behave, its persona, its goals, and specifically *how and when* to utilize its assigned `tools`.
- `tools`: A list containing the actual Python tool functions the agent is allowed to use (e.g., `[get_weather]`).

Best Practice: Provide clear and specific `instruction` prompts. The more detailed the instructions, the better the LLM can understand its role and how to use its tools effectively. Be explicit about error handling if needed.

Best Practice: Choose descriptive `name` and `description` values. These are used internally by ADK and are vital for features like automatic delegation (covered later).

```
# @title Define the Weather Agent
# Use one of the model constants defined earlier
AGENT_MODEL = MODEL_GEMINI_2_0_FLASH # Starting with Gemini

weather_agent = Agent(
    name="weather_agent_v1",
    model=AGENT_MODEL, # Can be a string for Gemini or a
LiteLlm object
    description="Provides weather information for specific
cities.",
    instruction="You are a helpful weather assistant. "
                "When the user asks for the weather in a
specific city, "
                "use the 'get_weather' tool to find the
information. "
                "If the tool returns an error, inform the user
politely. "
                "If the tool is successful, present the weather
report clearly.",
    tools=[get_weather], # Pass the function directly
)

print(f"Agent '{weather_agent.name}' created using model
'{AGENT_MODEL}'.")
```

3. Setup Runner and Session Service

To manage conversations and execute the agent, we need two more components:

- `SessionService`: Responsible for managing conversation history and state for different users and sessions. The `InMemorySessionService` is a simple implementation that stores everything in memory, suitable for testing and simple applications. It keeps track of the messages exchanged. We'll explore state persistence more in Step 4.
- `Runner`: The engine that orchestrates the interaction flow. It takes user input, routes it to the appropriate agent, manages calls to the LLM and tools based on the agent's logic, handles session updates via the `SessionService`, and yields events representing the progress of the interaction.

```
# @title Setup Session Service and Runner

# --- Session Management ---
# Key Concept: SessionService stores conversation history &
state.
```

```
# InMemorySessionService is simple, non-persistent storage for
this tutorial.
session_service = InMemorySessionService()

# Define constants for identifying the interaction context
APP_NAME = "weather_tutorial_app"
USER_ID = "user_1"
SESSION_ID = "session_001" # Using a fixed ID for simplicity

# Create the specific session where the conversation will
happen
session = await session_service.create_session(
    app_name=APP_NAME,
    user_id=USER_ID,
    session_id=SESSION_ID
)
print(f"Session created: App='{APP_NAME}', User='{USER_ID}',
Session='{SESSION_ID}'")

# --- Runner ---
# Key Concept: Runner orchestrates the agent execution loop.
runner = Runner(
    agent=weather_agent, # The agent we want to run
    app_name=APP_NAME,   # Associates runs with our app
    session_service=session_service # Uses our session manager
)
print(f"Runner created for agent '{runner.agent.name}'.")
```

4. Interact with the Agent

We need a way to send messages to our agent and receive its responses. Since LLM calls and tool executions can take time, ADK's `Runner` operates asynchronously.

We'll define an `async` helper function (`call_agent_async`) that:

1. Takes a user query string.
2. Packages it into the ADK `Content` format.
3. Calls `runner.run_async`, providing the user/session context and the new message.
4. Iterates through the **Events** yielded by the runner. Events represent steps in the agent's execution (e.g., tool call requested, tool result received, intermediate LLM thought, final response).
5. Identifies and prints the **final response** event using `event.is_final_response()`.

Why `async`? Interactions with LLMs and potentially tools (like external APIs) are I/O-bound operations. Using `asyncio` allows the program to handle these operations efficiently without blocking execution.

```
# @title Define Agent Interaction Function

from google.genai import types # For creating message
Content/Parts

async def call_agent_async(query: str, runner, user_id,
session_id):
  """Sends a query to the agent and prints the final
response."""
  print(f"\n>>> User Query: {query}")

  # Prepare the user's message in ADK format
  content = types.Content(role='user', parts=
[types.Part(text=query)])

  final_response_text = "Agent did not produce a final
response." # Default

  # Key Concept: run_async executes the agent logic and yields
Events.
  # We iterate through events to find the final answer.
  async for event in runner.run_async(user_id=user_id,
session_id=session_id, new_message=content):
      # You can uncomment the line below to see *all* events
during execution
      # print(f"  [Event] Author: {event.author}, Type:
{type(event).__name__}, Final: {event.is_final_response()},
Content: {event.content}")

      # Key Concept: is_final_response() marks the concluding
message for the turn.
      if event.is_final_response():
          if event.content and event.content.parts:
             # Assuming text response in the first part
             final_response_text = event.content.parts[0].text
          elif event.actions and event.actions.escalate: #
Handle potential errors/escalations
             final_response_text = f"Agent escalated:
{event.error_message or 'No specific message.'}"
          # Add more checks here if needed (e.g., specific
error codes)
          break # Stop processing events once the final
response is found

  print(f"<<< Agent Response: {final_response_text}")
```

5. Run the Conversation

Finally, let's test our setup by sending a few queries to the agent. We wrap our `async` calls in a main `async` function and run it using `await`.

Watch the output:

- See the user queries.
- Notice the `--- Tool: get_weather called... ---` logs when the agent uses the tool.
- Observe the agent's final responses, including how it handles the case where weather data isn't available (for Paris).

```
# @title Run the Initial Conversation

# We need an async function to await our interaction helper
async def run_conversation():
    await call_agent_async("What is the weather like in
London?",
                                       runner=runner,
                                       user_id=USER_ID,
                                       session_id=SESSION_ID)

    await call_agent_async("How about Paris?",
                                       runner=runner,
                                       user_id=USER_ID,
                                       session_id=SESSION_ID) #
Expecting the tool's error message

    await call_agent_async("Tell me the weather in New York",
                                       runner=runner,
                                       user_id=USER_ID,
                                       session_id=SESSION_ID)

# Execute the conversation using await in an async context
(like Colab/Jupyter)
await run_conversation()

# --- OR ---

# Uncomment the following lines if running as a standard Python
script (.py file):
# import asyncio
# if __name__ == "__main__":
#     try:
#         asyncio.run(run_conversation())
#     except Exception as e:
#         print(f"An error occurred: {e}")
```

Congratulations! You've successfully built and interacted with your first ADK agent. It understands the user's request, uses a tool to find information, and responds appropriately based on the tool's result.

In the next step, we'll explore how to easily switch the underlying Language Model powering this agent.

Step 2: Going Multi-Model with LiteLLM [Optional]

In Step 1, we built a functional Weather Agent powered by a specific Gemini model. While effective, real-world applications often benefit from the flexibility to use *different* Large Language Models (LLMs). Why?

- **Performance:** Some models excel at specific tasks (e.g., coding, reasoning, creative writing).
- **Cost:** Different models have varying price points.
- **Capabilities:** Models offer diverse features, context window sizes, and fine-tuning options.
- **Availability/Redundancy:** Having alternatives ensures your application remains functional even if one provider experiences issues.

ADK makes switching between models seamless through its integration with the **LiteLLM** library. LiteLLM acts as a consistent interface to over 100 different LLMs.

In this step, we will:

1. Learn how to configure an ADK `Agent` to use models from providers like OpenAI (GPT) and Anthropic (Claude) using the `LiteLlm` wrapper.
2. Define, configure (with their own sessions and runners), and immediately test instances of our Weather Agent, each backed by a different LLM.
3. Interact with these different agents to observe potential variations in their responses, even when using the same underlying tool.

1. Import `LiteLlm`

We imported this during the initial setup (Step 0), but it's the key component for multi-model support:

```
# @title 1. Import LiteLlm
from google.adk.models.lite_llm import LiteLlm
```

2. Define and Test Multi-Model Agents

Instead of passing only a model name string (which defaults to Google's Gemini models), we wrap the desired model identifier string within the `LiteLlm` class.

- **Key Concept: `LiteLlm` Wrapper:** The `LiteLlm(model="provider/model_name")` syntax tells ADK to route requests for this agent through the LiteLLM library to the specified model provider.

Make sure you have configured the necessary API keys for OpenAI and Anthropic in Step 0. We'll use the `call_agent_async` function (defined earlier, which now accepts `runner`, `user_id`, and `session_id`) to interact with each agent immediately after its setup.

Each block below will:

- Define the agent using a specific LiteLLM model (`MODEL_GPT_4O` or `MODEL_CLAUDE_SONNET`).
- Create a *new, separate* `InMemorySessionService` and session specifically for that agent's test run. This keeps the conversation histories isolated for this demonstration.
- Create a `Runner` configured for the specific agent and its session service.
- Immediately call `call_agent_async` to send a query and test the agent.

Best Practice: Use constants for model names (like `MODEL_GPT_4O`, `MODEL_CLAUDE_SONNET` defined in Step 0) to avoid typos and make code easier to manage.

Error Handling: We wrap the agent definitions in `try...except` blocks. This prevents the entire code cell from failing if an API key for a specific provider is missing or invalid, allowing the tutorial to proceed with the models that *are* configured.

First, let's create and test the agent using OpenAI's GPT-4o.

```
# @title Define and Test GPT Agent
# Make sure 'get_weather' function from Step 1 is defined in
your environment.
# Make sure 'call_agent_async' is defined from earlier.

# --- Agent using GPT-4o ---
weather_agent_gpt = None # Initialize to None
runner_gpt = None      # Initialize runner to None

try:
    weather_agent_gpt = Agent(
        name="weather_agent_gpt",
        # Key change: Wrap the LiteLLM model identifier
        model=LiteLlm(model=MODEL_GPT_4O),
        description="Provides weather information (using GPT-
4o).",
        instruction="You are a helpful weather assistant
powered by GPT-4o. "
                    "Use the 'get_weather' tool for city
weather requests. "
                    "Clearly present successful reports or
polite error messages based on the tool's output status.",
        tools=[get_weather], # Re-use the same tool
    )
    print(f"Agent '{weather_agent_gpt.name}' created using
model '{MODEL_GPT_4O}'.")

    # InMemorySessionService is simple, non-persistent storage
for this tutorial.
    session_service_gpt = InMemorySessionService() # Create a
dedicated service

    # Define constants for identifying the interaction context
    APP_NAME_GPT = "weather_tutorial_app_gpt" # Unique app name
for this test
    USER_ID_GPT = "user_1_gpt"
    SESSION_ID_GPT = "session_001_gpt" # Using a fixed ID for
simplicity

    # Create the specific session where the conversation will
happen
    session_gpt = await session_service_gpt.create_session(
        app_name=APP_NAME_GPT,
        user_id=USER_ID_GPT,
        session_id=SESSION_ID_GPT
    )
    print(f"Session created: App='{APP_NAME_GPT}',
User='{USER_ID_GPT}', Session='{SESSION_ID_GPT}'")

    # Create a runner specific to this agent and its session
service
    runner_gpt = Runner(
        agent=weather_agent_gpt,
        app_name=APP_NAME_GPT,       # Use the specific app
name
        session_service=session_service_gpt # Use the specific
session service
        )
    print(f"Runner created for agent
'{runner_gpt.agent.name}'.")

    # --- Test the GPT Agent ---
    print("\n--- Testing GPT Agent ---")
    # Ensure call_agent_async uses the correct runner, user_id,
session_id
    await call_agent_async(query = "What's the weather in
Tokyo?",
                           runner=runner_gpt,
                           user_id=USER_ID_GPT,
                           session_id=SESSION_ID_GPT)
    # --- OR ---

    # Uncomment the following lines if running as a standard
Python script (.py file):
    # import asyncio
    # if __name__ == "__main__":
    #     try:
    #         asyncio.run(call_agent_async(query = "What's the
weather in Tokyo?",
    #                      runner=runner_gpt,
    #                       user_id=USER_ID_GPT,
    #                       session_id=SESSION_ID_GPT)
    #     except Exception as e:
    #         print(f"An error occurred: {e}")

except Exception as e:
    print(f"❌ Could not create or run GPT agent
'{MODEL_GPT_4O}'. Check API Key and model name. Error: {e}")
```

Next, we'll do the same for Anthropic's Claude Sonnet.

```
# @title Define and Test Claude Agent

# Make sure 'get_weather' function from Step 1 is defined in
your environment.
# Make sure 'call_agent_async' is defined from earlier.

# --- Agent using Claude Sonnet ---
weather_agent_claude = None # Initialize to None
runner_claude = None      # Initialize runner to None

try:
    weather_agent_claude = Agent(
        name="weather_agent_claude",
        # Key change: Wrap the LiteLLM model identifier
        model=LiteLlm(model=MODEL_CLAUDE_SONNET),
        description="Provides weather information (using Claude
Sonnet).",
        instruction="You are a helpful weather assistant
powered by Claude Sonnet. "
                    "Use the 'get_weather' tool for city
weather requests. "
                    "Analyze the tool's dictionary output
('status', 'report'/'error_message'). "
                    "Clearly present successful reports or
polite error messages.",
        tools=[get_weather], # Re-use the same tool
    )
    print(f"Agent '{weather_agent_claude.name}' created using
model '{MODEL_CLAUDE_SONNET}'.")

    # InMemorySessionService is simple, non-persistent storage
for this tutorial.
    session_service_claude = InMemorySessionService() # Create
a dedicated service

    # Define constants for identifying the interaction context
    APP_NAME_CLAUDE = "weather_tutorial_app_claude" # Unique
app name
    USER_ID_CLAUDE = "user_1_claude"
    SESSION_ID_CLAUDE = "session_001_claude" # Using a fixed ID
for simplicity

    # Create the specific session where the conversation will
happen
    session_claude = await
session_service_claude.create_session(
        app_name=APP_NAME_CLAUDE,
        user_id=USER_ID_CLAUDE,
        session_id=SESSION_ID_CLAUDE
    )
    print(f"Session created: App='{APP_NAME_CLAUDE}',
User='{USER_ID_CLAUDE}', Session='{SESSION_ID_CLAUDE}'")

    # Create a runner specific to this agent and its session
service
    runner_claude = Runner(
        agent=weather_agent_claude,
        app_name=APP_NAME_CLAUDE,       # Use the specific app
name
        session_service=session_service_claude # Use the
specific session service
        )
    print(f"Runner created for agent
'{runner_claude.agent.name}'.")

    # --- Test the Claude Agent ---
    print("\n--- Testing Claude Agent ---")
    # Ensure call_agent_async uses the correct runner, user_id,
session_id
    await call_agent_async(query = "Weather in London please.",
                           runner=runner_claude,
                           user_id=USER_ID_CLAUDE,
                           session_id=SESSION_ID_CLAUDE)

    # --- OR ---

    # Uncomment the following lines if running as a standard
Python script (.py file):
    # import asyncio
    # if __name__ == "__main__":
    #     try:
    #         asyncio.run(call_agent_async(query = "Weather in
London please.",
    #                                  runner=runner_claude,
    #                                   user_id=USER_ID_CLAUDE,
    #                                   session_id=SESSION_ID_CLAUDE)
    #     except Exception as e:
    #         print(f"An error occurred: {e}")


except Exception as e:
    print(f"❌ Could not create or run Claude agent
'{MODEL_CLAUDE_SONNET}'. Check API Key and model name. Error:
{e}")
```

Observe the output carefully from both code blocks. You should see:

1. Each agent (`weather_agent_gpt` , `weather_agent_claude`) is created successfully (if API keys are valid).
2. A dedicated session and runner are set up for each.
3. Each agent correctly identifies the need to use the `get_weather` tool when processing the query (you'll see the `--- Tool: get_weather called... ---` log).
4. The *underlying tool logic* remains identical, always returning our mock data.
5. However, the **final textual response** generated by each agent might differ slightly in phrasing, tone, or formatting. This is because the instruction prompt is interpreted and executed by different LLMs (GPT-4o vs. Claude Sonnet).

This step demonstrates the power and flexibility ADK + LiteLLM provide. You can easily experiment with and deploy agents using various LLMs while keeping your core application logic (tools, fundamental agent structure) consistent.

In the next step, we'll move beyond a single agent and build a small team where agents can delegate tasks to each other!

Step 3: Building an Agent Team - Delegation for Greetings & Farewells

In Steps 1 and 2, we built and experimented with a single agent focused solely on weather lookups. While effective for its specific task, real-world applications often involve handling a wider variety of user interactions. We *could* keep adding more tools and complex instructions to our single weather agent, but this can quickly become unmanageable and less efficient.

A more robust approach is to build an **Agent Team**. This involves:

1. Creating multiple, **specialized agents**, each designed for a specific capability (e.g., one for weather, one for greetings, one for calculations).
2. Designating a **root agent** (or orchestrator) that receives the initial user request.
3. Enabling the root agent to **delegate** the request to the most appropriate specialized sub-agent based on the user's intent.

Why build an Agent Team?

- **Modularity:** Easier to develop, test, and maintain individual agents.
- **Specialization:** Each agent can be fine-tuned (instructions, model choice) for its specific task.
- **Scalability:** Simpler to add new capabilities by adding new agents.
- **Efficiency:** Allows using potentially simpler/cheaper models for simpler tasks (like greetings).

In this step, we will:

1. Define simple tools for handling greetings (`say_hello`) and farewells (`say_goodbye`).

2. Create two new specialized sub-agents: `greeting_agent` and `farewell_agent`.
3. Update our main weather agent (`weather_agent_v2`) to act as the **root agent**.
4. Configure the root agent with its sub-agents, enabling **automatic delegation**.
5. Test the delegation flow by sending different types of requests to the root agent.

---

1. Define Tools for Sub-Agents

First, let's create the simple Python functions that will serve as tools for our new specialist agents. Remember, clear docstrings are vital for the agents that will use them.

```python
# @title Define Tools for Greeting and Farewell Agents
from typing import Optional # Make sure to import Optional

# Ensure 'get_weather' from Step 1 is available if running this
step independently.
# def get_weather(city: str) -> dict: ... (from Step 1)

def say_hello(name: Optional[str] = None) -> str:
    """Provides a simple greeting. If a name is provided, it
will be used.

    Args:
        name (str, optional): The name of the person to greet.
Defaults to a generic greeting if not provided.

    Returns:
        str: A friendly greeting message.
    """
    if name:
        greeting = f"Hello, {name}!"
        print(f"--- Tool: say_hello called with name: {name} --
-")
    else:
        greeting = "Hello there!" # Default greeting if name is
None or not explicitly passed
        print(f"--- Tool: say_hello called without a specific
name (name_arg_value: {name}) ---")
    return greeting

def say_goodbye() -> str:
    """Provides a simple farewell message to conclude the
conversation."""
    print(f"--- Tool: say_goodbye called ---")
    return "Goodbye! Have a great day."

print("Greeting and Farewell tools defined.")

# Optional self-test
print(say_hello("Alice"))
print(say_hello()) # Test with no argument (should use default
"Hello there!")
print(say_hello(name=None)) # Test with name explicitly as None
(should use default "Hello there!")
```

---

2. Define the Sub-Agents (Greeting & Farewell)

Now, create the `Agent` instances for our specialists. Notice their highly focused `instruction` and, critically, their clear `description`. The `description` is the primary information the *root agent* uses to decide *when* to delegate to these sub-agents.

Best Practice: Sub-agent `description` fields should accurately and concisely summarize their specific capability. This is crucial for effective automatic delegation.

Best Practice: Sub-agent `instruction` fields should be tailored to their limited scope, telling them exactly what to do and *what not* to do (e.g., "Your *only* task is...").

```
# @title Define Greeting and Farewell Sub-Agents

# If you want to use models other than Gemini, Ensure LiteLlm
is imported and API keys are set (from Step 0/2)
# from google.adk.models.lite_llm import LiteLlm
# MODEL_GPT_4O, MODEL_CLAUDE_SONNET etc. should be defined
# Or else, continue to use: model = MODEL_GEMINI_2_0_FLASH

# --- Greeting Agent ---
greeting_agent = None
try:
    greeting_agent = Agent(
        # Using a potentially different/cheaper model for a
simple task
        model = MODEL_GEMINI_2_0_FLASH,
        # model=LiteLlm(model=MODEL_GPT_4O), # If you would
like to experiment with other models
        name="greeting_agent",
        instruction="You are the Greeting Agent. Your ONLY task
is to provide a friendly greeting to the user. "
                    "Use the 'say_hello' tool to generate the
greeting. "
                    "If the user provides their name, make sure
to pass it to the tool. "
                    "Do not engage in any other conversation or
tasks.",
        description="Handles simple greetings and hellos using
the 'say_hello' tool.", # Crucial for delegation
        tools=[say_hello],
    )
    print(f"✅ Agent '{greeting_agent.name}' created using
model '{greeting_agent.model}'.")
except Exception as e:
    print(f"❌ Could not create Greeting agent. Check API Key
({greeting_agent.model}). Error: {e}")

# --- Farewell Agent ---
farewell_agent = None
try:
    farewell_agent = Agent(
        # Can use the same or a different model
        model = MODEL_GEMINI_2_0_FLASH,
        # model=LiteLlm(model=MODEL_GPT_4O), # If you would
like to experiment with other models
        name="farewell_agent",
        instruction="You are the Farewell Agent. Your ONLY task
is to provide a polite goodbye message. "
                    "Use the 'say_goodbye' tool when the user
indicates they are leaving or ending the conversation "
                    "(e.g., using words like 'bye', 'goodbye',
'thanks bye', 'see you'). "
                    "Do not perform any other actions.",
        description="Handles simple farewells and goodbyes
using the 'say_goodbye' tool.", # Crucial for delegation
        tools=[say_goodbye],
    )
    print(f"✅ Agent '{farewell_agent.name}' created using
model '{farewell_agent.model}'.")
except Exception as e:
    print(f"❌ Could not create Farewell agent. Check API Key
({farewell_agent.model}). Error: {e}")
```

3. Define the Root Agent (Weather Agent v2) with Sub-Agents

Now, we upgrade our `weather_agent`. The key changes are:

- Adding the `sub_agents` parameter: We pass a list containing the `greeting_agent` and `farewell_agent` instances we just created.

- Updating the `instruction`: We explicitly tell the root agent *about* its sub-agents and *when* it should delegate tasks to them.

Key Concept: Automatic Delegation (Auto Flow) By providing the `sub_agents` list, ADK enables automatic delegation. When the root agent receives a user query, its LLM considers not only its own instructions and tools but also the `description` of each sub-agent. If the LLM determines that a query aligns better with a sub-agent's described capability (e.g., "Handles simple greetings"), it will automatically generate a special internal action to *transfer control* to that sub-agent for that turn. The sub-agent then processes the query using its own model, instructions, and tools.

Best Practice: Ensure the root agent's instructions clearly guide its delegation decisions. Mention the sub-agents by name and describe the conditions under which delegation should occur.

```
# @title Define the Root Agent with Sub-Agents

# Ensure sub-agents were created successfully before defining
the root agent.
# Also ensure the original 'get_weather' tool is defined.
root_agent = None
runner_root = None # Initialize runner

if greeting_agent and farewell_agent and 'get_weather' in
globals():
    # Let's use a capable Gemini model for the root agent to
handle orchestration
    root_agent_model = MODEL_GEMINI_2_0_FLASH

    weather_agent_team = Agent(
        name="weather_agent_v2", # Give it a new version name
        model=root_agent_model,
        description="The main coordinator agent. Handles
weather requests and delegates greetings/farewells to
specialists.",
        instruction="You are the main Weather Agent
coordinating a team. Your primary responsibility is to provide
weather information. "
                    "Use the 'get_weather' tool ONLY for
specific weather requests (e.g., 'weather in London'). "
                    "You have specialized sub-agents: "
                    "1. 'greeting_agent': Handles simple
greetings like 'Hi', 'Hello'. Delegate to it for these. "
                    "2. 'farewell_agent': Handles simple
farewells like 'Bye', 'See you'. Delegate to it for these. "
                    "Analyze the user's query. If it's a
greeting, delegate to 'greeting_agent'. If it's a farewell,
delegate to 'farewell_agent'. "
```

```
          "If it's a weather request, handle it
yourself using 'get_weather'. "
          "For anything else, respond appropriately
or state you cannot handle it.",
    tools=[get_weather], # Root agent still needs the
weather tool for its core task
    # Key change: Link the sub-agents here!
    sub_agents=[greeting_agent, farewell_agent]
  )
  print(f"✅ Root Agent '{weather_agent_team.name}' created
using model '{root_agent_model}' with sub-agents: {[sa.name for
sa in weather_agent_team.sub_agents]}")

else:
  print("❌ Cannot create root agent because one or more sub-
agents failed to initialize or 'get_weather' tool is missing.")
  if not greeting_agent: print(" - Greeting Agent is
missing.")
  if not farewell_agent: print(" - Farewell Agent is
missing.")
  if 'get_weather' not in globals(): print(" - get_weather
function is missing.")
```

4. Interact with the Agent Team

Now that we've defined our root agent (`weather_agent_team` - *Note: Ensure this variable name matches the one defined in the previous code block, likely* `# @title Define the Root Agent with Sub-Agents`*, which might have named it* `root_agent`) with its specialized sub-agents, let's test the delegation mechanism.

The following code block will:

1. Define an `async` function `run_team_conversation`.
2. Inside this function, create a *new, dedicated* `InMemorySessionService` and a specific session (`session_001_agent_team`) just for this test run. This isolates the conversation history for testing the team dynamics.
3. Create a `Runner` (`runner_agent_team`) configured to use our `weather_agent_team` (the root agent) and the dedicated session service.
4. Use our updated `call_agent_async` function to send different types of queries (greeting, weather request, farewell) to the `runner_agent_team`. We explicitly pass the runner, user ID, and session ID for this specific test.
5. Immediately execute the `run_team_conversation` function.

We expect the following flow:

1. The "Hello there!" query goes to `runner_agent_team`.
2. The root agent (`weather_agent_team`) receives it and, based on its instructions and the `greeting_agent`'s description, delegates the task.
3. `greeting_agent` handles the query, calls its `say_hello` tool, and generates the response.
4. The "What is the weather in New York?" query is *not* delegated and is handled directly by the root agent using its `get_weather` tool.
5. The "Thanks, bye!" query is delegated to the `farewell_agent`, which uses its `say_goodbye` tool.

```
# @title Interact with the Agent Team
import asyncio # Ensure asyncio is imported

# Ensure the root agent (e.g., 'weather_agent_team' or
'root_agent' from the previous cell) is defined.
# Ensure the call_agent_async function is defined.

# Check if the root agent variable exists before defining the
conversation function
root_agent_var_name = 'root_agent' # Default name from Step 3
guide
if 'weather_agent_team' in globals(): # Check if user used this
name instead
    root_agent_var_name = 'weather_agent_team'
elif 'root_agent' not in globals():
    print("⚠️ Root agent ('root_agent' or 'weather_agent_team')
not found. Cannot define run_team_conversation.")
    # Assign a dummy value to prevent NameError later if the
code block runs anyway
    root_agent = None # Or set a flag to prevent execution

# Only define and run if the root agent exists
if root_agent_var_name in globals() and globals()
[root_agent_var_name]:
    # Define the main async function for the conversation
logic.
    # The 'await' keywords INSIDE this function are necessary
for async operations.
    async def run_team_conversation():
        print("\n--- Testing Agent Team Delegation ---")
        session_service = InMemorySessionService()
        APP_NAME = "weather_tutorial_agent_team"
        USER_ID = "user_1_agent_team"
        SESSION_ID = "session_001_agent_team"
        session = await session_service.create_session(
            app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID
        )
        print(f"Session created: App='{APP_NAME}',
User='{USER_ID}', Session='{SESSION_ID}'")

        actual_root_agent = globals()[root_agent_var_name]
        runner_agent_team = Runner( # Or use InMemoryRunner
            agent=actual_root_agent,
            app_name=APP_NAME,
            session_service=session_service
        )
        print(f"Runner created for agent
'{actual_root_agent.name}'.")

        # --- Interactions using await (correct within async
def) ---
        await call_agent_async(query = "Hello there!",
                               runner=runner_agent_team,
                               user_id=USER_ID,
                               session_id=SESSION_ID)
        await call_agent_async(query = "What is the weather in
New York?",
                               runner=runner_agent_team,
                               user_id=USER_ID,
                               session_id=SESSION_ID)
        await call_agent_async(query = "Thanks, bye!",
                               runner=runner_agent_team,
                               user_id=USER_ID,
                               session_id=SESSION_ID)

    # --- Execute the `run_team_conversation` async function --
-
    # Choose ONE of the methods below based on your
environment.
    # Note: This may require API keys for the models used!

    # METHOD 1: Direct await (Default for Notebooks/Async
REPLs)
    # If your environment supports top-level await (like
Colab/Jupyter notebooks),
    # it means an event loop is already running, so you can
directly await the function.
    print("Attempting execution using 'await' (default for
notebooks)...")
    await run_team_conversation()

    # METHOD 2: asyncio.run (For Standard Python Scripts [.py])
    # If running this code as a standard Python script from
your terminal,
    # the script context is synchronous. `asyncio.run()` is
needed to
```

```
    # create and manage an event loop to execute your async
function.
    # To use this method:
    # 1. Comment out the `await run_team_conversation()` line
above.
    # 2. Uncomment the following block:
    """
    import asyncio
    if __name__ == "__main__": # Ensures this runs only when
script is executed directly
        print("Executing using 'asyncio.run()' (for standard
Python scripts)...")
        try:
            # This creates an event loop, runs your async
function, and closes the loop.
            asyncio.run(run_team_conversation())
        except Exception as e:
            print(f"An error occurred: {e}")
    """

else:
    # This message prints if the root agent variable wasn't
found earlier
    print("\n⚠️ Skipping agent team conversation execution as
the root agent was not successfully defined in a previous
step.")
```

Look closely at the output logs, especially the `--- Tool: ... called ---` messages. You should observe:

- For "Hello there!", the `say_hello` tool was called (indicating `greeting_agent` handled it).
- For "What is the weather in New York?", the `get_weather` tool was called (indicating the root agent handled it).
- For "Thanks, bye!", the `say_goodbye` tool was called (indicating `farewell_agent` handled it).

This confirms successful **automatic delegation**! The root agent, guided by its instructions and the `description`s of its `sub_agents`, correctly routed user requests to the appropriate specialist agent within the team.

You've now structured your application with multiple collaborating agents. This modular design is fundamental for building more complex and capable agent systems. In the next step, we'll give our agents the ability to remember information across turns using session state.

Step 4: Adding Memory and Personalization with Session State

So far, our agent team can handle different tasks through delegation, but each interaction starts fresh – the agents have no memory of past conversations or user preferences within a session. To create more sophisticated and context-aware experiences, agents need **memory**. ADK provides this through **Session State**.

What is Session State?

- It's a Python dictionary (`session.state`) tied to a specific user session (identified by `APP_NAME`, `USER_ID`, `SESSION_ID`).
- It persists information *across multiple conversational turns* within that session.
- Agents and Tools can read from and write to this state, allowing them to remember details, adapt behavior, and personalize responses.

How Agents Interact with State:

1. **`ToolContext` (Primary Method):** Tools can accept a `ToolContext` object (automatically provided by ADK if declared as the last argument). This object gives direct access to the session state via `tool_context.state`, allowing tools to read preferences or save results *during* execution.
2. **`output_key` (Auto-Save Agent Response):** An `Agent` can be configured with an `output_key="your_key"`. ADK will then automatically save the agent's final textual response for a turn into `session.state["your_key"]`.

In this step, we will enhance our Weather Bot team by:

1. Using a **new** `InMemorySessionService` to demonstrate state in isolation.
2. Initializing session state with a user preference for `temperature_unit`.
3. Creating a state-aware version of the weather tool (`get_weather_stateful`) that reads this preference via `ToolContext` and adjusts its output format (Celsius/Fahrenheit).
4. Updating the root agent to use this stateful tool and configuring it with an `output_key` to automatically save its final weather report to the

session state.

5. Running a conversation to observe how the initial state affects the tool, how manual state changes alter subsequent behavior, and how `output_key` persists the agent's response.

---

1. Initialize New Session Service and State

To clearly demonstrate state management without interference from prior steps, we'll instantiate a new `InMemorySessionService`. We'll also create a session with an initial state defining the user's preferred temperature unit.

```python
# @title 1. Initialize New Session Service and State

# Import necessary session components
from google.adk.sessions import InMemorySessionService

# Create a NEW session service instance for this state
demonstration
session_service_stateful = InMemorySessionService()
print("✅ New InMemorySessionService created for state
demonstration.")

# Define a NEW session ID for this part of the tutorial
SESSION_ID_STATEFUL = "session_state_demo_001"
USER_ID_STATEFUL = "user_state_demo"

# Define initial state data - user prefers Celsius initially
initial_state = {
    "user_preference_temperature_unit": "Celsius"
}

# Create the session, providing the initial state
session_stateful = await
session_service_stateful.create_session(
    app_name=APP_NAME, # Use the consistent app name
    user_id=USER_ID_STATEFUL,
    session_id=SESSION_ID_STATEFUL,
    state=initial_state # <<< Initialize state during creation
)
print(f"✅ Session '{SESSION_ID_STATEFUL}' created for user
'{USER_ID_STATEFUL}'.")

# Verify the initial state was set correctly
retrieved_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,
session_id = SESSION_ID_STATEFUL)
print("\n--- Initial Session State ---")
if retrieved_session:
    print(retrieved_session.state)
else:
    print("Error: Could not retrieve session.")
```

2. Create State-Aware Weather Tool (`get_weather_stateful`)

Now, we create a new version of the weather tool. Its key feature is accepting `tool_context: ToolContext` which allows it to access `tool_context.state`. It will read the `user_preference_temperature_unit` and format the temperature accordingly.

- **Key Concept:** `ToolContext` This object is the bridge allowing your tool logic to interact with the session's context, including reading and writing state variables. ADK injects it automatically if defined as the last parameter of your tool function.
- **Best Practice:** When reading from state, use `dictionary.get('key', default_value)` to handle cases where the key might not exist yet, ensuring your tool doesn't crash.

```
from google.adk.tools.tool_context import ToolContext

def get_weather_stateful(city: str, tool_context: ToolContext)
-> dict:
    """Retrieves weather, converts temp unit based on session
state."""
    print(f"--- Tool: get_weather_stateful called for {city} --
-")

    # --- Read preference from state ---
    preferred_unit =
tool_context.state.get("user_preference_temperature_unit",
"Celsius") # Default to Celsius
    print(f"--- Tool: Reading state
'user_preference_temperature_unit': {preferred_unit} ---")

    city_normalized = city.lower().replace(" ", "")

    # Mock weather data (always stored in Celsius internally)
    mock_weather_db = {
        "newyork": {"temp_c": 25, "condition": "sunny"},
        "london": {"temp_c": 15, "condition": "cloudy"},
        "tokyo": {"temp_c": 18, "condition": "light rain"},
    }

    if city_normalized in mock_weather_db:
        data = mock_weather_db[city_normalized]
        temp_c = data["temp_c"]
        condition = data["condition"]

        # Format temperature based on state preference
        if preferred_unit == "Fahrenheit":
            temp_value = (temp_c * 9/5) + 32 # Calculate
Fahrenheit
            temp_unit = "°F"
        else: # Default to Celsius
            temp_value = temp_c
            temp_unit = "°C"

        report = f"The weather in {city.capitalize()} is
{condition} with a temperature of {temp_value:.0f}{temp_unit}."
        result = {"status": "success", "report": report}
        print(f"--- Tool: Generated report in {preferred_unit}.
Result: {result} ---")

        # Example of writing back to state (optional for this
tool)
        tool_context.state["last_city_checked_stateful"] = city
        print(f"--- Tool: Updated state
'last_city_checked_stateful': {city} ---")

        return result
    else:
        # Handle city not found
        error_msg = f"Sorry, I don't have weather information
for '{city}'."
        print(f"--- Tool: City '{city}' not found. ---")
        return {"status": "error", "error_message": error_msg}

print("✅ State-aware 'get_weather_stateful' tool defined.")
```

3. Redefine Sub-Agents and Update Root Agent

To ensure this step is self-contained and builds correctly, we first redefine the `greeting_agent` and `farewell_agent` exactly as they were in Step 3. Then, we define our new root agent (`weather_agent_v4_stateful`):

- It uses the new `get_weather_stateful` tool.
- It includes the greeting and farewell sub-agents for delegation.

- **Crucially**, it sets `output_key="last_weather_report"` which automatically saves its final weather response to the session state.

```
# @title 3. Redefine Sub-Agents and Update Root Agent with
output_key

# Ensure necessary imports: Agent, LiteLlm, Runner
from google.adk.agents import Agent
from google.adk.models.lite_llm import LiteLlm
from google.adk.runners import Runner
# Ensure tools 'say_hello', 'say_goodbye' are defined (from
Step 3)
# Ensure model constants MODEL_GPT_4O, MODEL_GEMINI_2_0_FLASH
etc. are defined

# --- Redefine Greeting Agent (from Step 3) ---
greeting_agent = None
try:
    greeting_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="greeting_agent",
        instruction="You are the Greeting Agent. Your ONLY task
is to provide a friendly greeting using the 'say_hello' tool.
Do nothing else.",
        description="Handles simple greetings and hellos using
the 'say_hello' tool.",
        tools=[say_hello],
    )
    print(f"✅ Agent '{greeting_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Greeting agent. Error: {e}")

# --- Redefine Farewell Agent (from Step 3) ---
farewell_agent = None
try:
    farewell_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="farewell_agent",
        instruction="You are the Farewell Agent. Your ONLY task
is to provide a polite goodbye message using the 'say_goodbye'
tool. Do not perform any other actions.",
        description="Handles simple farewells and goodbyes
using the 'say_goodbye' tool.",
        tools=[say_goodbye],
    )
    print(f"✅ Agent '{farewell_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Farewell agent. Error: {e}")

# --- Define the Updated Root Agent ---
root_agent_stateful = None
runner_root_stateful = None # Initialize runner
```

```
# Check prerequisites before creating the root agent
if greeting_agent and farewell_agent and 'get_weather_stateful'
in globals():

    root_agent_model = MODEL_GEMINI_2_0_FLASH # Choose
orchestration model

    root_agent_stateful = Agent(
        name="weather_agent_v4_stateful", # New version name
        model=root_agent_model,
        description="Main agent: Provides weather (state-aware
unit), delegates greetings/farewells, saves report to state.",
        instruction="You are the main Weather Agent. Your job
is to provide weather using 'get_weather_stateful'. "
                    "The tool will format the temperature based
on user preference stored in state. "
                    "Delegate simple greetings to
'greeting_agent' and farewells to 'farewell_agent'. "
                    "Handle only weather requests, greetings,
and farewells.",
        tools=[get_weather_stateful], # Use the state-aware
tool
        sub_agents=[greeting_agent, farewell_agent], # Include
sub-agents
        output_key="last_weather_report" # <<< Auto-save
agent's final weather response
    )
    print(f"✅ Root Agent '{root_agent_stateful.name}' created
using stateful tool and output_key.")

    # --- Create Runner for this Root Agent & NEW Session
Service ---
    runner_root_stateful = Runner(
        agent=root_agent_stateful,
        app_name=APP_NAME,
        session_service=session_service_stateful # Use the NEW
stateful session service
    )
    print(f"✅ Runner created for stateful root agent
'{runner_root_stateful.agent.name}' using stateful session
service.")

else:
    print("❌ Cannot create stateful root agent. Prerequisites
missing.")
    if not greeting_agent: print(" - greeting_agent definition
missing.")
    if not farewell_agent: print(" - farewell_agent definition
missing.")
    if 'get_weather_stateful' not in globals(): print(" -
get_weather_stateful tool missing.")
```

4. Interact and Test State Flow

Now, let's execute a conversation designed to test the state interactions using the `runner_root_stateful` (associated with our stateful agent and the `session_service_stateful`). We'll use the `call_agent_async` function defined earlier, ensuring we pass the correct runner, user ID (`USER_ID_STATEFUL`), and session ID (`SESSION_ID_STATEFUL`).

The conversation flow will be:

1. **Check weather (London):** The `get_weather_stateful` tool should read the initial "Celsius" preference from the session state initialized in Section 1. The root agent's final response (the weather report in Celsius) should get saved to `state['last_weather_report']` via the `output_key` configuration.
2. **Manually update state:** We will *directly modify* the state stored within the `InMemorySessionService` instance (`session_service_stateful`).
    - **Why direct modification?** The `session_service.get_session()` method returns a *copy* of the session. Modifying that copy wouldn't affect the state used in subsequent agent runs. For this testing scenario with `InMemorySessionService`, we access the internal `sessions` dictionary to change the *actual* stored state value for `user_preference_temperature_unit` to "Fahrenheit". *Note: In real applications, state changes are typically triggered by tools or agent logic returning `EventActions(state_delta=...)`, not direct manual updates.*
3. **Check weather again (New York):** The `get_weather_stateful` tool should now read the updated "Fahrenheit" preference from the state and convert the temperature accordingly. The root agent's *new* response (weather in Fahrenheit) will overwrite the previous value in `state['last_weather_report']` due to the `output_key`.
4. **Greet the agent:** Verify that delegation to the `greeting_agent` still works correctly alongside the stateful operations. This interaction will become the *last* response saved by `output_key` in this specific sequence.
5. **Inspect final state:** After the conversation, we retrieve the session one last time (getting a copy) and print its state to confirm the `user_preference_temperature_unit` is indeed "Fahrenheit", observe the

final value saved by `output_key` (which will be the greeting in this run), and see the `last_city_checked_stateful` value written by the tool.

```
# @title 4. Interact to Test State Flow and output_key
import asyncio # Ensure asyncio is imported

# Ensure the stateful runner (runner_root_stateful) is
available from the previous cell
# Ensure call_agent_async, USER_ID_STATEFUL,
SESSION_ID_STATEFUL, APP_NAME are defined

if 'runner_root_stateful' in globals() and
runner_root_stateful:
    # Define the main async function for the stateful
conversation logic.
    # The 'await' keywords INSIDE this function are necessary
for async operations.
    async def run_stateful_conversation():
        print("\n--- Testing State: Temp Unit Conversion &
output_key ---")

        # 1. Check weather (Uses initial state: Celsius)
        print("--- Turn 1: Requesting weather in London (expect
Celsius) ---")
        await call_agent_async(query= "What's the weather in
London?",
                               runner=runner_root_stateful,
                               user_id=USER_ID_STATEFUL,
                               session_id=SESSION_ID_STATEFUL
                              )

        # 2. Manually update state preference to Fahrenheit -
DIRECTLY MODIFY STORAGE
        print("\n--- Manually Updating State: Setting unit to
Fahrenheit ---")
        try:
            # Access the internal storage directly - THIS IS
SPECIFIC TO InMemorySessionService for testing
            # NOTE: In production with persistent services
(Database, VertexAI), you would
            # typically update state via agent actions or
specific service APIs if available,
            # not by direct manipulation of internal storage.
            stored_session =
session_service_stateful.sessions[APP_NAME][USER_ID_STATEFUL]
[SESSION_ID_STATEFUL]

stored_session.state["user_preference_temperature_unit"] =
"Fahrenheit"
            # Optional: You might want to update the timestamp
as well if any logic depends on it
            # import time
            # stored_session.last_update_time = time.time()
            print(f"--- Stored session state updated. Current
'user_preference_temperature_unit':
{stored_session.state.get('user_preference_temperature_unit',
'Not Set')} ---") # Added .get for safety
        except KeyError:
            print(f"--- Error: Could not retrieve session
'{SESSION_ID_STATEFUL}' from internal storage for user
'{USER_ID_STATEFUL}' in app '{APP_NAME}' to update state. Check
IDs and if session was created. ---")
        except Exception as e:
             print(f"--- Error updating internal session state:
{e} ---")

        # 3. Check weather again (Tool should now use
Fahrenheit)
        # This will also update 'last_weather_report' via
output_key
        print("\n--- Turn 2: Requesting weather in New York
(expect Fahrenheit) ---")
        await call_agent_async(query= "Tell me the weather in
New York.",
                                runner=runner_root_stateful,
                                user_id=USER_ID_STATEFUL,
                                session_id=SESSION_ID_STATEFUL
                               )

        # 4. Test basic delegation (should still work)
        # This will update 'last_weather_report' again,
overwriting the NY weather report
        print("\n--- Turn 3: Sending a greeting ---")
        await call_agent_async(query= "Hi!",
                                runner=runner_root_stateful,
                                user_id=USER_ID_STATEFUL,
                                session_id=SESSION_ID_STATEFUL
                               )

    # --- Execute the `run_stateful_conversation` async
function ---
    # Choose ONE of the methods below based on your
environment.

    # METHOD 1: Direct await (Default for Notebooks/Async
REPLs)
    # If your environment supports top-level await (like
Colab/Jupyter notebooks),
    # it means an event loop is already running, so you can
directly await the function.
    print("Attempting execution using 'await' (default for
notebooks)...")
    await run_stateful_conversation()

    # METHOD 2: asyncio.run (For Standard Python Scripts [.py])
    # If running this code as a standard Python script from
your terminal,
    # the script context is synchronous. `asyncio.run()` is
needed to
    # create and manage an event loop to execute your async
function.
    # To use this method:
    # 1. Comment out the `await run_stateful_conversation()`
line above.
    # 2. Uncomment the following block:
    """
    import asyncio
    if __name__ == "__main__": # Ensures this runs only when
script is executed directly
        print("Executing using 'asyncio.run()' (for standard
Python scripts)...")
        try:
            # This creates an event loop, runs your async
function, and closes the loop.
            asyncio.run(run_stateful_conversation())
        except Exception as e:
            print(f"An error occurred: {e}")
    """

    # --- Inspect final session state after the conversation --
-
    # This block runs after either execution method completes.
    print("\n--- Inspecting Final Session State ---")
    final_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id= USER_ID_STATEFUL,

session_id=SESSION_ID_STATEFUL)
    if final_session:
        # Use .get() for safer access to potentially missing
keys
        print(f"Final Preference:
{final_session.state.get('user_preference_temperature_unit',
'Not Set')}")
        print(f"Final Last Weather Report (from output_key):
{final_session.state.get('last_weather_report', 'Not Set')}")
        print(f"Final Last City Checked (by tool):
{final_session.state.get('last_city_checked_stateful', 'Not
Set')}")
        # Print full state for detailed view
        # print(f"Full State Dict: {final_session.state}") #
For detailed view
    else:
        print("\n❌ Error: Could not retrieve final session
state.")

else:
    print("\n⚠️ Skipping state test conversation. Stateful root
agent runner ('runner_root_stateful') is not available.")
```

By reviewing the conversation flow and the final session state printout, you can confirm:

- **State Read:** The weather tool (`get_weather_stateful`) correctly read `user_preference_temperature_unit` from state, initially using "Celsius" for London.
- **State Update:** The direct modification successfully changed the stored preference to "Fahrenheit".
- **State Read (Updated):** The tool subsequently read "Fahrenheit" when asked for New York's weather and performed the conversion.
- **Tool State Write:** The tool successfully wrote the `last_city_checked_stateful` ("New York" after the second weather check) into the state via `tool_context.state` .
- **Delegation:** The delegation to the `greeting_agent` for "Hi!" functioned correctly even after state modifications.
- **`output_key` :** The `output_key="last_weather_report"` successfully saved the root agent's *final* response for *each turn* where the root agent was the one ultimately responding. In this sequence, the last response was the greeting ("Hello, there!"), so that overwrote the weather report in the state key.
- **Final State:** The final check confirms the preference persisted as "Fahrenheit".

You've now successfully integrated session state to personalize agent behavior using `ToolContext` , manually manipulated state for testing `InMemorySessionService` , and observed how `output_key` provides a simple mechanism for saving the agent's last response to state. This foundational understanding of state management is key as we proceed to implement safety guardrails using callbacks in the next steps.

Step 5: Adding Safety - Input Guardrail with `before_model_callback`

Our agent team is becoming more capable, remembering preferences and using tools effectively. However, in real-world scenarios, we often need safety mechanisms to control the agent's behavior *before* potentially problematic requests even reach the core Large Language Model (LLM).

ADK provides **Callbacks** – functions that allow you to hook into specific points in the agent's execution lifecycle. The `before_model_callback` is particularly useful for input safety.

What is `before_model_callback`?

- It's a Python function you define that ADK executes *just before* an agent sends its compiled request (including conversation history, instructions, and the latest user message) to the underlying LLM.
- **Purpose:** Inspect the request, modify it if necessary, or block it entirely based on predefined rules.

Common Use Cases:

- **Input Validation/Filtering:** Check if user input meets criteria or contains disallowed content (like PII or keywords).
- **Guardrails:** Prevent harmful, off-topic, or policy-violating requests from being processed by the LLM.
- **Dynamic Prompt Modification:** Add timely information (e.g., from session state) to the LLM request context just before sending.

How it Works:

1. Define a function accepting `callback_context: CallbackContext` and `llm_request: LlmRequest`.
    - `callback_context`: Provides access to agent info, session state (`callback_context.state`), etc.
    - `llm_request`: Contains the full payload intended for the LLM (`contents`, `config`).
2. Inside the function:
    - **Inspect:** Examine `llm_request.contents` (especially the last user message).
    - **Modify (Use Caution):** You *can* change parts of `llm_request`.

- **Block (Guardrail):** Return an `LlmResponse` object. ADK will send this response back immediately, *skipping* the LLM call for that turn.
- **Allow:** Return `None`. ADK proceeds to call the LLM with the (potentially modified) request.

In this step, we will:

1. Define a `before_model_callback` function (`block_keyword_guardrail`) that checks the user's input for a specific keyword ("BLOCK").
2. Update our stateful root agent (`weather_agent_v4_stateful` from Step 4) to use this callback.
3. Create a new runner associated with this updated agent but using the *same stateful session service* to maintain state continuity.
4. Test the guardrail by sending both normal and keyword-containing requests.

---

1. Define the Guardrail Callback Function

This function will inspect the last user message within the `llm_request` content. If it finds "BLOCK" (case-insensitive), it constructs and returns an `LlmResponse` to block the flow; otherwise, it returns `None`.

```
# @title 1. Define the before_model_callback Guardrail

# Ensure necessary imports are available
from google.adk.agents.callback_context import CallbackContext
from google.adk.models.llm_request import LlmRequest
from google.adk.models.llm_response import LlmResponse
from google.genai import types # For creating response content
from typing import Optional

def block_keyword_guardrail(
    callback_context: CallbackContext, llm_request: LlmRequest
) -> Optional[LlmResponse]:
    """
    Inspects the latest user message for 'BLOCK'. If found,
blocks the LLM call
    and returns a predefined LlmResponse. Otherwise, returns
None to proceed.
    """
    agent_name = callback_context.agent_name # Get the name of
the agent whose model call is being intercepted
    print(f"--- Callback: block_keyword_guardrail running for
agent: {agent_name} ---")
```

```
    # Extract the text from the latest user message in the
request history
    last_user_message_text = ""
    if llm_request.contents:
        # Find the most recent message with role 'user'
        for content in reversed(llm_request.contents):
            if content.role == 'user' and content.parts:
                # Assuming text is in the first part for
simplicity
                if content.parts[0].text:
                    last_user_message_text =
content.parts[0].text
                    break # Found the last user message text

    print(f"--- Callback: Inspecting last user message:
'{last_user_message_text[:100]}...' ---") # Log first 100 chars

    # --- Guardrail Logic ---
    keyword_to_block = "BLOCK"
    if keyword_to_block in last_user_message_text.upper(): #
Case-insensitive check
        print(f"--- Callback: Found '{keyword_to_block}'.
Blocking LLM call! ---")
        # Optionally, set a flag in state to record the block
event

callback_context.state["guardrail_block_keyword_triggered"] =
True
        print(f"--- Callback: Set state
'guardrail_block_keyword_triggered': True ---")

        # Construct and return an LlmResponse to stop the flow
and send this back instead
        return LlmResponse(
            content=types.Content(
                role="model", # Mimic a response from the
agent's perspective
                parts=[types.Part(text=f"I cannot process this
request because it contains the blocked keyword
'{keyword_to_block}'.")],
            )
            # Note: You could also set an error_message field
here if needed
        )
    else:
        # Keyword not found, allow the request to proceed to
the LLM
        print(f"--- Callback: Keyword not found. Allowing LLM
call for {agent_name}. ---")
        return None # Returning None signals ADK to continue
normally
```

```
print("✅ block_keyword_guardrail function defined.")
```

2. Update Root Agent to Use the Callback

We redefine the root agent, adding the `before_model_callback` parameter and pointing it to our new guardrail function. We'll give it a new version name for clarity.

Important: We need to redefine the sub-agents (`greeting_agent` , `farewell_agent`) and the stateful tool (`get_weather_stateful`) within this context if they are not already available from previous steps, ensuring the root agent definition has access to all its components.

```
# @title 2. Update Root Agent with before_model_callback


# --- Redefine Sub-Agents (Ensures they exist in this context)
---
greeting_agent = None
try:
    # Use a defined model constant
    greeting_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="greeting_agent", # Keep original name for
consistency
        instruction="You are the Greeting Agent. Your ONLY task
is to provide a friendly greeting using the 'say_hello' tool.
Do nothing else.",
        description="Handles simple greetings and hellos using
the 'say_hello' tool.",
        tools=[say_hello],
    )
    print(f"✅ Sub-Agent '{greeting_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Greeting agent. Check
Model/API Key ({greeting_agent.model}). Error: {e}")

farewell_agent = None
try:
    # Use a defined model constant
    farewell_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="farewell_agent", # Keep original name
        instruction="You are the Farewell Agent. Your ONLY task
is to provide a polite goodbye message using the 'say_goodbye'
tool. Do not perform any other actions.",
        description="Handles simple farewells and goodbyes
using the 'say_goodbye' tool.",
        tools=[say_goodbye],
    )
    print(f"✅ Sub-Agent '{farewell_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Farewell agent. Check
Model/API Key ({farewell_agent.model}). Error: {e}")


# --- Define the Root Agent with the Callback ---
root_agent_model_guardrail = None
runner_root_model_guardrail = None

# Check all components before proceeding
if greeting_agent and farewell_agent and 'get_weather_stateful'
in globals() and 'block_keyword_guardrail' in globals():

    # Use a defined model constant
    root_agent_model = MODEL_GEMINI_2_0_FLASH

    root_agent_model_guardrail = Agent(
        name="weather_agent_v5_model_guardrail", # New version
name for clarity
        model=root_agent_model,
        description="Main agent: Handles weather, delegates
greetings/farewells, includes input keyword guardrail.",
        instruction="You are the main Weather Agent. Provide
weather using 'get_weather_stateful'. "
                    "Delegate simple greetings to
'greeting_agent' and farewells to 'farewell_agent'. "
                    "Handle only weather requests, greetings,
and farewells.",
        tools=[get_weather],
        sub_agents=[greeting_agent, farewell_agent], #
Reference the redefined sub-agents
        output_key="last_weather_report", # Keep output_key
from Step 4
        before_model_callback=block_keyword_guardrail # <<<
Assign the guardrail callback
    )
    print(f"✅ Root Agent '{root_agent_model_guardrail.name}'
created with before_model_callback.")

    # --- Create Runner for this Agent, Using SAME Stateful
Session Service ---
    # Ensure session_service_stateful exists from Step 4
    if 'session_service_stateful' in globals():
        runner_root_model_guardrail = Runner(
            agent=root_agent_model_guardrail,
            app_name=APP_NAME, # Use consistent APP_NAME
            session_service=session_service_stateful # <<< Use
the service from Step 4
        )
        print(f"✅ Runner created for guardrail agent
'{runner_root_model_guardrail.agent.name}', using stateful
session service.")
    else:
        print("❌ Cannot create runner.
'session_service_stateful' from Step 4 is missing.")

else:
    print("❌ Cannot create root agent with model guardrail.
One or more prerequisites are missing or failed
initialization:")
    if not greeting_agent: print("   - Greeting Agent")
    if not farewell_agent: print("   - Farewell Agent")
    if 'get_weather_stateful' not in globals(): print("   -
'get_weather_stateful' tool")
    if 'block_keyword_guardrail' not in globals(): print("   -
'block_keyword_guardrail' callback")
```

3. Interact to Test the Guardrail

Let's test the guardrail's behavior. We'll use the *same session* (`SESSION_ID_STATEFUL`) as in Step 4 to show that state persists across these changes.

1. Send a normal weather request (should pass the guardrail and execute).
2. Send a request containing "BLOCK" (should be intercepted by the callback).
3. Send a greeting (should pass the root agent's guardrail, be delegated, and execute normally).

```
# @title 3. Interact to Test the Model Input Guardrail
import asyncio # Ensure asyncio is imported

# Ensure the runner for the guardrail agent is available
if 'runner_root_model_guardrail' in globals() and
runner_root_model_guardrail:
    # Define the main async function for the guardrail test
conversation.
    # The 'await' keywords INSIDE this function are necessary
for async operations.
    async def run_guardrail_test_conversation():
        print("\n--- Testing Model Input Guardrail ---")

        # Use the runner for the agent with the callback and
the existing stateful session ID
        # Define a helper lambda for cleaner interaction calls
        interaction_func = lambda query:
call_agent_async(query,

runner_root_model_guardrail,

USER_ID_STATEFUL, # Use existing user ID

SESSION_ID_STATEFUL # Use existing session ID
                                                        )
        # 1. Normal request (Callback allows, should use
Fahrenheit from previous state change)
        print("--- Turn 1: Requesting weather in London (expect
allowed, Fahrenheit) ---")
        await interaction_func("What is the weather in
London?")

        # 2. Request containing the blocked keyword (Callback
intercepts)
        print("\n--- Turn 2: Requesting with blocked keyword
(expect blocked) ---")
        await interaction_func("BLOCK the request for weather
in Tokyo") # Callback should catch "BLOCK"

        # 3. Normal greeting (Callback allows root agent,
delegation happens)
        print("\n--- Turn 3: Sending a greeting (expect
allowed) ---")
        await interaction_func("Hello again")

    # --- Execute the `run_guardrail_test_conversation` async
function ---
    # Choose ONE of the methods below based on your
environment.

    # METHOD 1: Direct await (Default for Notebooks/Async
REPLs)
    # If your environment supports top-level await (like
Colab/Jupyter notebooks),
    # it means an event loop is already running, so you can
directly await the function.
    print("Attempting execution using 'await' (default for
notebooks)...")
    await run_guardrail_test_conversation()

    # METHOD 2: asyncio.run (For Standard Python Scripts [.py])
    # If running this code as a standard Python script from
your terminal,
    # the script context is synchronous. `asyncio.run()` is
needed to
    # create and manage an event loop to execute your async
function.
    # To use this method:
```

```
    # 1. Comment out the `await
run_guardrail_test_conversation()` line above.
    # 2. Uncomment the following block:
    """
    import asyncio
    if __name__ == "__main__": # Ensures this runs only when
script is executed directly
        print("Executing using 'asyncio.run()' (for standard
Python scripts)...")
        try:
            # This creates an event loop, runs your async
function, and closes the loop.
            asyncio.run(run_guardrail_test_conversation())
        except Exception as e:
            print(f"An error occurred: {e}")
    """

    # --- Inspect final session state after the conversation --
-
    # This block runs after either execution method completes.
    # Optional: Check state for the trigger flag set by the
callback
    print("\n--- Inspecting Final Session State (After
Guardrail Test) ---")
    # Use the session service instance associated with this
stateful session
    final_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,

session_id=SESSION_ID_STATEFUL)
    if final_session:
        # Use .get() for safer access
        print(f"Guardrail Triggered Flag:
{final_session.state.get('guardrail_block_keyword_triggered',
'Not Set (or False)')}")
        print(f"Last Weather Report:
{final_session.state.get('last_weather_report', 'Not Set')}") #
Should be London weather if successful
        print(f"Temperature Unit:
{final_session.state.get('user_preference_temperature_unit',
'Not Set')}") # Should be Fahrenheit
        # print(f"Full State Dict: {final_session.state}") #
For detailed view
    else:
        print("\n❌ Error: Could not retrieve final session
state.")

else:
    print("\n⚠️ Skipping model guardrail test. Runner
('runner_root_model_guardrail') is not available.")
```

Observe the execution flow:

1. **London Weather:** The callback runs for `weather_agent_v5_model_guardrail`, inspects the message, prints "Keyword not found. Allowing LLM call.", and returns `None`. The agent proceeds, calls the `get_weather_stateful` tool (which uses the "Fahrenheit" preference from Step 4's state change), and returns the weather. This response updates `last_weather_report` via `output_key`.
2. **BLOCK Request:** The callback runs again for `weather_agent_v5_model_guardrail`, inspects the message, finds "BLOCK", prints "Blocking LLM call!", sets the state flag, and returns the predefined `LlmResponse`. The agent's underlying LLM is *never called* for this turn. The user sees the callback's blocking message.
3. **Hello Again:** The callback runs for `weather_agent_v5_model_guardrail`, allows the request. The root agent then delegates to `greeting_agent`. *Note: The `before_model_callback` defined on the root agent does NOT automatically apply to sub-agents.* The `greeting_agent` proceeds normally, calls its `say_hello` tool, and returns the greeting.

You have successfully implemented an input safety layer! The `before_model_callback` provides a powerful mechanism to enforce rules and control agent behavior *before* expensive or potentially risky LLM calls are made. Next, we'll apply a similar concept to add guardrails around tool usage itself.

Step 6: Adding Safety - Tool Argument Guardrail (`before_tool_callback`)

In Step 5, we added a guardrail to inspect and potentially block user input *before* it reached the LLM. Now, we'll add another layer of control *after* the LLM has decided to use a tool but *before* that tool actually executes. This is useful for validating the *arguments* the LLM wants to pass to the tool.

ADK provides the `before_tool_callback` for this precise purpose.

What is `before_tool_callback`?

- It's a Python function executed just *before* a specific tool function runs, after the LLM has requested its use and decided on the arguments.

- **Purpose:** Validate tool arguments, prevent tool execution based on specific inputs, modify arguments dynamically, or enforce resource usage policies.

Common Use Cases:

- **Argument Validation:** Check if arguments provided by the LLM are valid, within allowed ranges, or conform to expected formats.
- **Resource Protection:** Prevent tools from being called with inputs that might be costly, access restricted data, or cause unwanted side effects (e.g., blocking API calls for certain parameters).
- **Dynamic Argument Modification:** Adjust arguments based on session state or other contextual information before the tool runs.

How it Works:

1. Define a function accepting `tool: BaseTool`, `args: Dict[str, Any]`, and `tool_context: ToolContext`.
    - `tool`: The tool object about to be called (inspect `tool.name`).
    - `args`: The dictionary of arguments the LLM generated for the tool.
    - `tool_context`: Provides access to session state (`tool_context.state`), agent info, etc.
2. Inside the function:
    - **Inspect:** Examine the `tool.name` and the `args` dictionary.
    - **Modify:** Change values within the `args` dictionary *directly*. If you return `None`, the tool runs with these modified args.
    - **Block/Override (Guardrail):** Return a **dictionary**. ADK treats this dictionary as the *result* of the tool call, completely *skipping* the execution of the original tool function. The dictionary should ideally match the expected return format of the tool it's blocking.
    - **Allow:** Return `None`. ADK proceeds to execute the actual tool function with the (potentially modified) arguments.

In this step, we will:

1. Define a `before_tool_callback` function (`block_paris_tool_guardrail`) that specifically checks if the `get_weather_stateful` tool is called with the city "Paris".

2. If "Paris" is detected, the callback will block the tool and return a custom error dictionary.
3. Update our root agent (`weather_agent_v6_tool_guardrail`) to include *both* the `before_model_callback` and this new `before_tool_callback`.
4. Create a new runner for this agent, using the same stateful session service.
5. Test the flow by requesting weather for allowed cities and the blocked city ("Paris").

---

1. Define the Tool Guardrail Callback Function

This function targets the `get_weather_stateful` tool. It checks the `city` argument. If it's "Paris", it returns an error dictionary that looks like the tool's own error response. Otherwise, it allows the tool to run by returning `None`.

```
# @title 1. Define the before_tool_callback Guardrail

# Ensure necessary imports are available
from google.adk.tools.base_tool import BaseTool
from google.adk.tools.tool_context import ToolContext
from typing import Optional, Dict, Any # For type hints

def block_paris_tool_guardrail(
    tool: BaseTool, args: Dict[str, Any], tool_context:
ToolContext
) -> Optional[Dict]:
    """
    Checks if 'get_weather_stateful' is called for 'Paris'.
    If so, blocks the tool execution and returns a specific
error dictionary.
    Otherwise, allows the tool call to proceed by returning
None.
    """
    tool_name = tool.name
    agent_name = tool_context.agent_name # Agent attempting the
tool call
    print(f"--- Callback: block_paris_tool_guardrail running
for tool '{tool_name}' in agent '{agent_name}' ---")
    print(f"--- Callback: Inspecting args: {args} ---")

    # --- Guardrail Logic ---
    target_tool_name = "get_weather_stateful" # Match the
function name used by FunctionTool
    blocked_city = "paris"
    # Check if it's the correct tool and the city argument
matches the blocked city
    if tool_name == target_tool_name:
        city_argument = args.get("city", "") # Safely get the
'city' argument
        if city_argument and city_argument.lower() ==
blocked_city:
            print(f"--- Callback: Detected blocked city
'{city_argument}'. Blocking tool execution! ---")
            # Optionally update state

tool_context.state["guardrail_tool_block_triggered"] = True
            print(f"--- Callback: Set state
'guardrail_tool_block_triggered': True ---")

            # Return a dictionary matching the tool's expected
output format for errors
            # This dictionary becomes the tool's result,
skipping the actual tool run.
            return {
                "status": "error",
                "error_message": f"Policy restriction: Weather
checks for '{city_argument.capitalize()}' are currently
disabled by a tool guardrail."
            }
        else:
             print(f"--- Callback: City '{city_argument}' is
allowed for tool '{tool_name}'. ---")
    else:
        print(f"--- Callback: Tool '{tool_name}' is not the
target tool. Allowing. ---")


    # If the checks above didn't return a dictionary, allow the
tool to execute
    print(f"--- Callback: Allowing tool '{tool_name}' to
proceed. ---")
    return None # Returning None allows the actual tool
function to run

print("✅ block_paris_tool_guardrail function defined.")
```

2. Update Root Agent to Use Both Callbacks

We redefine the root agent again (`weather_agent_v6_tool_guardrail`), this time adding the `before_tool_callback` parameter alongside the `before_model_callback` from Step 5.

Self-Contained Execution Note: Similar to Step 5, ensure all prerequisites (sub-agents, tools, `before_model_callback`) are defined or available in the execution context before defining this agent.

```
# @title 2. Update Root Agent with BOTH Callbacks (Self-
Contained)

# --- Ensure Prerequisites are Defined ---
# (Include or ensure execution of definitions for: Agent,
LiteLlm, Runner, ToolContext,
#  MODEL constants, say_hello, say_goodbye, greeting_agent,
farewell_agent,
#  get_weather_stateful, block_keyword_guardrail,
block_paris_tool_guardrail)

# --- Redefine Sub-Agents (Ensures they exist in this context)
---
greeting_agent = None
try:
    # Use a defined model constant
    greeting_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="greeting_agent", # Keep original name for
consistency
        instruction="You are the Greeting Agent. Your ONLY task
is to provide a friendly greeting using the 'say_hello' tool.
Do nothing else.",
        description="Handles simple greetings and hellos using
the 'say_hello' tool.",
        tools=[say_hello],
    )
    print(f"✅ Sub-Agent '{greeting_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Greeting agent. Check
Model/API Key ({greeting_agent.model}). Error: {e}")

farewell_agent = None
try:
    # Use a defined model constant
    farewell_agent = Agent(
        model=MODEL_GEMINI_2_0_FLASH,
        name="farewell_agent", # Keep original name
        instruction="You are the Farewell Agent. Your ONLY task
is to provide a polite goodbye message using the 'say_goodbye'
tool. Do not perform any other actions.",
        description="Handles simple farewells and goodbyes
using the 'say_goodbye' tool.",
        tools=[say_goodbye],
    )
    print(f"✅ Sub-Agent '{farewell_agent.name}' redefined.")
except Exception as e:
    print(f"❌ Could not redefine Farewell agent. Check
Model/API Key ({farewell_agent.model}). Error: {e}")

# --- Define the Root Agent with Both Callbacks ---
root_agent_tool_guardrail = None
runner_root_tool_guardrail = None

if ('greeting_agent' in globals() and greeting_agent and
    'farewell_agent' in globals() and farewell_agent and
    'get_weather_stateful' in globals() and
    'block_keyword_guardrail' in globals() and
    'block_paris_tool_guardrail' in globals()):

    root_agent_model = MODEL_GEMINI_2_0_FLASH

    root_agent_tool_guardrail = Agent(
        name="weather_agent_v6_tool_guardrail", # New version
name
        model=root_agent_model,
        description="Main agent: Handles weather, delegates,
includes input AND tool guardrails.",
        instruction="You are the main Weather Agent. Provide
weather using 'get_weather_stateful'. "
                    "Delegate greetings to 'greeting_agent' and
farewells to 'farewell_agent'. "
                    "Handle only weather, greetings, and
farewells.",
        tools=[get_weather_stateful],
        sub_agents=[greeting_agent, farewell_agent],
        output_key="last_weather_report",
        before_model_callback=block_keyword_guardrail, # Keep
model guardrail
        before_tool_callback=block_paris_tool_guardrail # <<<
Add tool guardrail
    )
    print(f"✅ Root Agent '{root_agent_tool_guardrail.name}'
created with BOTH callbacks.")

    # --- Create Runner, Using SAME Stateful Session Service --
-
    if 'session_service_stateful' in globals():
        runner_root_tool_guardrail = Runner(
            agent=root_agent_tool_guardrail,
            app_name=APP_NAME,
            session_service=session_service_stateful # <<< Use
the service from Step 4/5
        )
        print(f"✅ Runner created for tool guardrail agent
'{runner_root_tool_guardrail.agent.name}', using stateful
session service.")
    else:
        print("❌ Cannot create runner.
'session_service_stateful' from Step 4/5 is missing.")
```

```
else:
    print("❌ Cannot create root agent with tool guardrail.
Prerequisites missing.")
```

3. Interact to Test the Tool Guardrail

Let's test the interaction flow, again using the same stateful session (`SESSION_ID_STATEFUL`) from the previous steps.

1. Request weather for "New York": Passes both callbacks, tool executes (using Fahrenheit preference from state).
2. Request weather for "Paris": Passes `before_model_callback`. LLM decides to call `get_weather_stateful(city='Paris')`. `before_tool_callback` intercepts, blocks the tool, and returns the error dictionary. Agent relays this error.
3. Request weather for "London": Passes both callbacks, tool executes normally.

```
# @title 3. Interact to Test the Tool Argument Guardrail
import asyncio # Ensure asyncio is imported

# Ensure the runner for the tool guardrail agent is available
if 'runner_root_tool_guardrail' in globals() and
runner_root_tool_guardrail:
    # Define the main async function for the tool guardrail
test conversation.
    # The 'await' keywords INSIDE this function are necessary
for async operations.
    async def run_tool_guardrail_test():
        print("\n--- Testing Tool Argument Guardrail ('Paris'
blocked) ---")

        # Use the runner for the agent with both callbacks and
the existing stateful session
        # Define a helper lambda for cleaner interaction calls
        interaction_func = lambda query:
call_agent_async(query,

runner_root_tool_guardrail,

USER_ID_STATEFUL, # Use existing user ID

SESSION_ID_STATEFUL # Use existing session ID
                                                        )
        # 1. Allowed city (Should pass both callbacks, use
Fahrenheit state)
        print("--- Turn 1: Requesting weather in New York
(expect allowed) ---")
        await interaction_func("What's the weather in New
York?")

        # 2. Blocked city (Should pass model callback, but be
blocked by tool callback)
        print("\n--- Turn 2: Requesting weather in Paris
(expect blocked by tool guardrail) ---")
        await interaction_func("How about Paris?") # Tool
callback should intercept this

        # 3. Another allowed city (Should work normally again)
        print("\n--- Turn 3: Requesting weather in London
(expect allowed) ---")
        await interaction_func("Tell me the weather in
London.")

    # --- Execute the `run_tool_guardrail_test` async function
---
    # Choose ONE of the methods below based on your
environment.

    # METHOD 1: Direct await (Default for Notebooks/Async
REPLs)
    # If your environment supports top-level await (like
Colab/Jupyter notebooks),
    # it means an event loop is already running, so you can
directly await the function.
    print("Attempting execution using 'await' (default for
notebooks)...")
    await run_tool_guardrail_test()

    # METHOD 2: asyncio.run (For Standard Python Scripts [.py])
    # If running this code as a standard Python script from
your terminal,
    # the script context is synchronous. `asyncio.run()` is
needed to
    # create and manage an event loop to execute your async
function.
    # To use this method:
    # 1. Comment out the `await run_tool_guardrail_test()` line
above.
    # 2. Uncomment the following block:
    """
    import asyncio
    if __name__ == "__main__": # Ensures this runs only when
script is executed directly
        print("Executing using 'asyncio.run()' (for standard
Python scripts)...")
        try:
            # This creates an event loop, runs your async
function, and closes the loop.
            asyncio.run(run_tool_guardrail_test())
        except Exception as e:
            print(f"An error occurred: {e}")
    """

    # --- Inspect final session state after the conversation --
-
    # This block runs after either execution method completes.
    # Optional: Check state for the tool block trigger flag
    print("\n--- Inspecting Final Session State (After Tool
Guardrail Test) ---")
    # Use the session service instance associated with this
stateful session
    final_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,

session_id= SESSION_ID_STATEFUL)
    if final_session:
        # Use .get() for safer access
        print(f"Tool Guardrail Triggered Flag:
{final_session.state.get('guardrail_tool_block_triggered', 'Not
Set (or False)')}")
        print(f"Last Weather Report:
{final_session.state.get('last_weather_report', 'Not Set')}") #
Should be London weather if successful
        print(f"Temperature Unit:
{final_session.state.get('user_preference_temperature_unit',
'Not Set')}") # Should be Fahrenheit
        # print(f"Full State Dict: {final_session.state}") #
For detailed view
    else:
        print("\n❌ Error: Could not retrieve final session
state.")

else:
    print("\n⚠️ Skipping tool guardrail test. Runner
('runner_root_tool_guardrail') is not available.")
```

Analyze the output:

1. **New York:** The `before_model_callback` allows the request. The LLM requests `get_weather_stateful`. The `before_tool_callback` runs, inspects the args (`{'city': 'New York'}`), sees it's not "Paris", prints "Allowing tool..." and returns `None`. The actual `get_weather_stateful`

function executes, reads "Fahrenheit" from state, and returns the weather report. The agent relays this, and it gets saved via `output_key`.

2. **Paris:** The `before_model_callback` allows the request. The LLM requests `get_weather_stateful(city='Paris')`. The `before_tool_callback` runs, inspects the args, detects "Paris", prints "Blocking tool execution!", sets the state flag, and returns the error dictionary `{'status': 'error', 'error_message': 'Policy restriction...'}`. The actual `get_weather_stateful` function is **never executed**. The agent receives the error dictionary *as if it were the tool's output* and formulates a response based on that error message.
3. **London:** Behaves like New York, passing both callbacks and executing the tool successfully. The new London weather report overwrites the `last_weather_report` in the state.

You've now added a crucial safety layer controlling not just *what* reaches the LLM, but also *how* the agent's tools can be used based on the specific arguments generated by the LLM. Callbacks like `before_model_callback` and `before_tool_callback` are essential for building robust, safe, and policy-compliant agent applications.

Conclusion: Your Agent Team is Ready!

Congratulations! You've successfully journeyed from building a single, basic weather agent to constructing a sophisticated, multi-agent team using the Agent Development Kit (ADK).

Let's recap what you've accomplished:

- You started with a **fundamental agent** equipped with a single tool (`get_weather`).
- You explored ADK's **multi-model flexibility** using LiteLLM, running the same core logic with different LLMs like Gemini, GPT-4o, and Claude.
- You embraced **modularity** by creating specialized sub-agents (`greeting_agent`, `farewell_agent`) and enabling **automatic delegation** from a root agent.
- You gave your agents **memory** using **Session State**, allowing them to remember user preferences (`temperature_unit`) and past interactions

(`output_key`).

- You implemented crucial **safety guardrails** using both `before_model_callback` (blocking specific input keywords) and `before_tool_callback` (blocking tool execution based on arguments like the city "Paris").

Through building this progressive Weather Bot team, you've gained hands-on experience with core ADK concepts essential for developing complex, intelligent applications.

Key Takeaways:

- **Agents & Tools:** The fundamental building blocks for defining capabilities and reasoning. Clear instructions and docstrings are paramount.
- **Runners & Session Services:** The engine and memory management system that orchestrate agent execution and maintain conversational context.
- **Delegation:** Designing multi-agent teams allows for specialization, modularity, and better management of complex tasks. Agent `description` is key for auto-flow.
- **Session State (`ToolContext`, `output_key`):** Essential for creating context-aware, personalized, and multi-turn conversational agents.
- **Callbacks (`before_model`, `before_tool`):** Powerful hooks for implementing safety, validation, policy enforcement, and dynamic modifications *before* critical operations (LLM calls or tool execution).
- **Flexibility (`LiteLlm`):** ADK empowers you to choose the best LLM for the job, balancing performance, cost, and features.

Where to Go Next?

Your Weather Bot team is a great starting point. Here are some ideas to further explore ADK and enhance your application:

1. **Real Weather API:** Replace the `mock_weather_db` in your `get_weather` tool with a call to a real weather API (like OpenWeatherMap, WeatherAPI).
2. **More Complex State:** Store more user preferences (e.g., preferred location, notification settings) or conversation summaries in the session

state.

3. **Refine Delegation:** Experiment with different root agent instructions or sub-agent descriptions to fine-tune the delegation logic. Could you add a "forecast" agent?
4. **Advanced Callbacks:**
    - Use `after_model_callback` to potentially reformat or sanitize the LLM's response *after* it's generated.
    - Use `after_tool_callback` to process or log the results returned by a tool.
    - Implement `before_agent_callback` or `after_agent_callback` for agent-level entry/exit logic.
5. **Error Handling:** Improve how the agent handles tool errors or unexpected API responses. Maybe add retry logic within a tool.
6. **Persistent Session Storage:** Explore alternatives to `InMemorySessionService` for storing session state persistently (e.g., using databases like Firestore or Cloud SQL – requires custom implementation or future ADK integrations).
7. **Streaming UI:** Integrate your agent team with a web framework (like FastAPI, as shown in the ADK Streaming Quickstart) to create a real-time chat interface.

The Agent Development Kit provides a robust foundation for building sophisticated LLM-powered applications. By mastering the concepts covered in this tutorial – tools, state, delegation, and callbacks – you are well-equipped to tackle increasingly complex agentic systems.

Happy building!