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

# 

This tutorial extends from the [Quickstart example](https://google.github.io/adk-docs/get-started/quickstart/) for [Agent Development Kit](https://google.github.io/adk-docs/get-started/). 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](https://github.com/google/adk-docs/tree/main/examples/python/tutorial/agent_team/adk-tutorial). 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!**

# @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 ---

MODEL\_GEMINI\_2\_0\_FLASH = "gemini-2.0-flash"

# Note: Specific model names might change. Refer to LiteLLM/Provider documentation.

MODEL\_GPT\_4O = "openai/gpt-4o"

MODEL\_CLAUDE\_SONNET = "anthropic/claude-3-sonnet-20240229"

print("\nEnvironment configured.")

## Step 1: Your First Agent - Basic Weather Lookup[¶](https://google.github.io/adk-docs/tutorials/agent-team/#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 = 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][¶](https://google.github.io/adk-docs/tutorials/agent-team/#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**](https://github.com/BerriAI/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 = 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 = 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[¶](https://google.github.io/adk-docs/tutorials/agent-team/#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.

# @title Define Tools for Greeting and Farewell Agents

# 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: str = "there") -> str:

"""Provides a simple greeting, optionally addressing the user by name.

Args:

name (str, optional): The name of the person to greet. Defaults to "there".

Returns:

str: A friendly greeting message.

"""

print(f"--- Tool: say\_hello called with name: {name} ---")

return f"Hello, {name}!"

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\_goodbye())

**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 = 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 descriptions 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[¶](https://google.github.io/adk-docs/tutorials/agent-team/#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.

# @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 = 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 = 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 = 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.as\_dict()}") # Use as\_dict() for clarity

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[¶](https://google.github.io/adk-docs/tutorials/agent-team/#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.
2. callback\_context: Provides access to agent info, session state (callback\_context.state), etc.
3. llm\_request: Contains the full payload intended for the LLM (contents, config).
4. Inside the function:
5. **Inspect:** Examine llm\_request.contents (especially the last user message).
6. **Modify (Use Caution):** You *can* change parts of llm\_request.
7. **Block (Guardrail):** Return an LlmResponse object. ADK will send this response back immediately, *skipping* the LLM call for that turn.
8. **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 = 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.as\_dict()}") # 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)[¶](https://google.github.io/adk-docs/tutorials/agent-team/#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.
2. tool: The tool object about to be called (inspect tool.name).
3. args: The dictionary of arguments the LLM generated for the tool.
4. tool\_context: Provides access to session state (tool\_context.state), agent info, etc.
5. Inside the function:
6. **Inspect:** Examine the tool.name and the args dictionary.
7. **Modify:** Change values within the args dictionary *directly*. If you return None, the tool runs with these modified args.
8. **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.
9. **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 = 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.as\_dict()}") # 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![¶](https://google.github.io/adk-docs/tutorials/agent-team/#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.

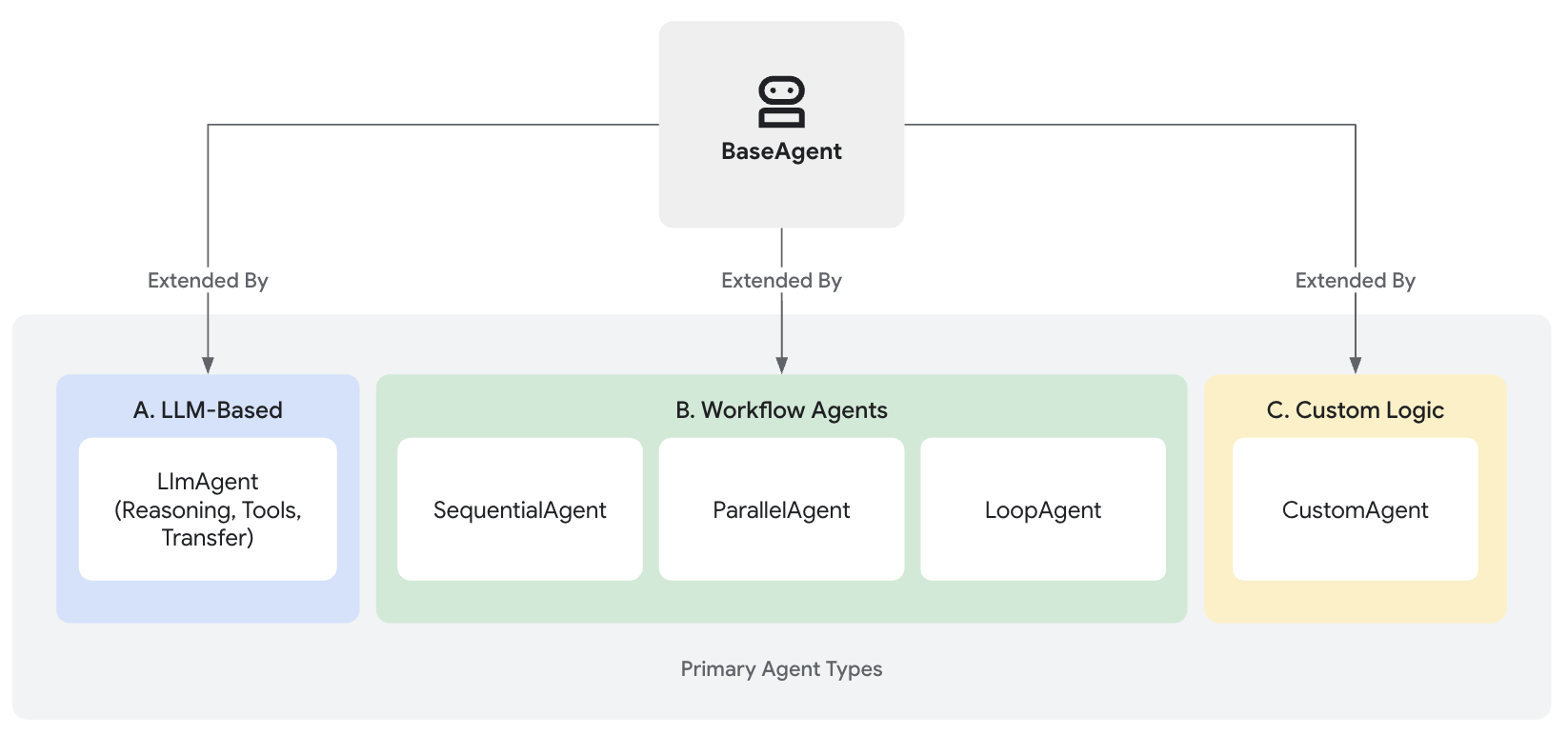
The Agent Development Kit provides a robust foundation for building sophisticated LLM-powered applications. By mastering the concepts covered in this tutorial – tools, state, delegation, and callbacks – you are well-equipped to tackle increasingly complex agentic systems.

Happy building!

# Agents[¶](https://google.github.io/adk-docs/agents/#agents)

In the Agent Development Kit (ADK), an **Agent** is a self-contained execution unit designed to act autonomously to achieve specific goals. Agents can perform tasks, interact with users, utilize external tools, and coordinate with other agents.

The foundation for all agents in ADK is the BaseAgent class. It serves as the fundamental blueprint. To create functional agents, you typically extend BaseAgent in one of three main ways, catering to different needs – from intelligent reasoning to structured process control.



## Core Agent Categories[¶](https://google.github.io/adk-docs/agents/#core-agent-categories)

ADK provides distinct agent categories to build sophisticated applications:

1. [**LLM Agents (LlmAgent, Agent)**](https://google.github.io/adk-docs/agents/llm-agents/): These agents utilize Large Language Models (LLMs) as their core engine to understand natural language, reason, plan, generate responses, and dynamically decide how to proceed or which tools to use, making them ideal for flexible, language-centric tasks. [Learn more about LLM Agents...](https://google.github.io/adk-docs/agents/llm-agents/)
2. [**Workflow Agents (SequentialAgent, ParallelAgent, LoopAgent)**](https://google.github.io/adk-docs/agents/workflow-agents/): These specialized agents control the execution flow of other agents in predefined, deterministic patterns (sequence, parallel, or loop) without using an LLM for the flow control itself, perfect for structured processes needing predictable execution. [Explore Workflow Agents...](https://google.github.io/adk-docs/agents/workflow-agents/)
3. [**Custom Agents**](https://google.github.io/adk-docs/agents/custom-agents/): Created by extending BaseAgent directly, these agents allow you to implement unique operational logic, specific control flows, or specialized integrations not covered by the standard types, catering to highly tailored application requirements. [Discover how to build Custom Agents...](https://google.github.io/adk-docs/agents/custom-agents/)

## Choosing the Right Agent Type[¶](https://google.github.io/adk-docs/agents/#choosing-the-right-agent-type)

The following table provides a high-level comparison to help distinguish between the agent types. As you explore each type in more detail in the subsequent sections, these distinctions will become clearer.

| **Feature** | **LLM Agent (LlmAgent)** | **Workflow Agent** | **Custom Agent (BaseAgent subclass)** |
| --- | --- | --- | --- |
| **Primary Function** | Reasoning, Generation, Tool Use | Controlling Agent Execution Flow | Implementing Unique Logic/Integrations |
| **Core Engine** | Large Language Model (LLM) | Predefined Logic (Sequence, Parallel, Loop) | Custom Python Code |
| **Determinism** | Non-deterministic (Flexible) | Deterministic (Predictable) | Can be either, based on implementation |
| **Primary Use** | Language tasks, Dynamic decisions | Structured processes, Orchestration | Tailored requirements, Specific workflows |

## Agents Working Together: Multi-Agent Systems[¶](https://google.github.io/adk-docs/agents/#agents-working-together-multi-agent-systems)

While each agent type serves a distinct purpose, the true power often comes from combining them. Complex applications frequently employ [multi-agent architectures](https://google.github.io/adk-docs/agents/multi-agents/) where:

* **LLM Agents** handle intelligent, language-based task execution.
* **Workflow Agents** manage the overall process flow using standard patterns.
* **Custom Agents** provide specialized capabilities or rules needed for unique integrations.

Understanding these core types is the first step toward building sophisticated, capable AI applications with ADK.

## What's Next?[¶](https://google.github.io/adk-docs/agents/#whats-next)

Now that you have an overview of the different agent types available in ADK, dive deeper into how they work and how to use them effectively:

* [**LLM Agents:**](https://google.github.io/adk-docs/agents/llm-agents/) Explore how to configure agents powered by large language models, including setting instructions, providing tools, and enabling advanced features like planning and code execution.
* [**Workflow Agents:**](https://google.github.io/adk-docs/agents/workflow-agents/) Learn how to orchestrate tasks using SequentialAgent, ParallelAgent, and LoopAgent for structured and predictable processes.
* [**Custom Agents:**](https://google.github.io/adk-docs/agents/custom-agents/) Discover the principles of extending BaseAgent to build agents with unique logic and integrations tailored to your specific needs.
* [**Multi-Agents:**](https://google.github.io/adk-docs/agents/multi-agents/) Understand how to combine different agent types to create sophisticated, collaborative systems capable of tackling complex problems.
* [**Models:**](https://google.github.io/adk-docs/agents/models/) Learn about the different LLM integrations available and how to select the right model for your agents.

# Workflow Agents[¶](https://google.github.io/adk-docs/agents/workflow-agents/#workflow-agents)

This section introduces "*workflow agents*" - **specialized agents that control the execution flow of its sub-agents**.

Workflow agents are specialized components in ADK designed purely for **orchestrating the execution flow of sub-agents**. Their primary role is to manage *how* and *when* other agents run, defining the control flow of a process.

Unlike [LLM Agents](https://google.github.io/adk-docs/agents/llm-agents/), which use Large Language Models for dynamic reasoning and decision-making, Workflow Agents operate based on **predefined logic**. They determine the execution sequence according to their type (e.g., sequential, parallel, loop) without consulting an LLM for the orchestration itself. This results in **deterministic and predictable execution patterns**.

ADK provides three core workflow agent types, each implementing a distinct execution pattern:

* **Sequential Agents**
* Executes sub-agents one after another, in **sequence**.  
   [Learn more](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/)
* **Loop Agents**
* **Repeatedly** executes its sub-agents until a specific termination condition is met.  
   [Learn more](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)
* **Parallel Agents**
* Executes multiple sub-agents in **parallel**.  
   [Learn more](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)

## Why Use Workflow Agents?[¶](https://google.github.io/adk-docs/agents/workflow-agents/#why-use-workflow-agents)

Workflow agents are essential when you need explicit control over how a series of tasks or agents are executed. They provide:

* **Predictability:** The flow of execution is guaranteed based on the agent type and configuration.
* **Reliability:** Ensures tasks run in the required order or pattern consistently.
* **Structure:** Allows you to build complex processes by composing agents within clear control structures.

While the workflow agent manages the control flow deterministically, the sub-agents it orchestrates can themselves be any type of agent, including intelligent LlmAgent instances. This allows you to combine structured process control with flexible, LLM-powered task execution.

# Sequential agents[¶](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/#sequential-agents)

## The SequentialAgent[¶](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/#the-sequentialagent)

The SequentialAgent is a [workflow agent](https://google.github.io/adk-docs/agents/workflow-agents/) that executes its sub-agents in the order they are specified in the list.

Use the SequentialAgent when you want the execution to occur in a fixed, strict order.

### Example[¶](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/#example)

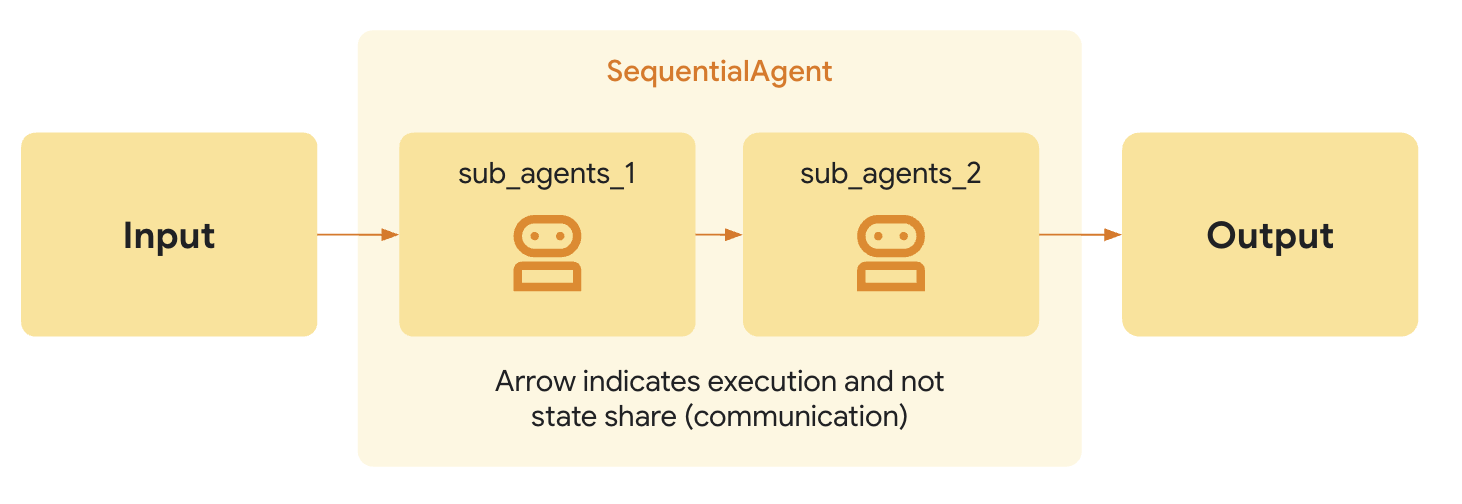
* You want to build an agent that can summarize any webpage, using two tools: get\_page\_contents and summarize\_page. Because the agent must always call get\_page\_contents before calling summarize\_page (you can't summarize from nothing!), you should build your agent using a SequentialAgent.

As with other [workflow agents](https://google.github.io/adk-docs/agents/workflow-agents/), the SequentialAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are concerned only with their execution (i.e. in sequence), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

### How it works[¶](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/#how-it-works)

When the SequentialAgent's run\_async() method is called, it performs the following actions:

1. **Iteration:** It iterates through the sub\_agents list in the order they were provided.
2. **Sub-Agent Execution:** For each sub-agent in the list, it calls the sub-agent's run\_async() method.



### Full Example: Code Development Pipeline[¶](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/#full-example-code-development-pipeline)

Consider a simplified code development pipeline:

* **Code Writer Agent:** An LlmAgent that generates initial code based on a specification.
* **Code Reviewer Agent:** An LlmAgent that reviews the generated code for errors, style issues, and adherence to best practices. It receives the output of the Code Writer Agent.
* **Code Refactorer Agent:** An LlmAgent that takes the reviewed code (and the reviewer's comments) and refactors it to improve quality and address issues.

A SequentialAgent is perfect for this:

SequentialAgent(sub\_agents=[CodeWriterAgent, CodeReviewerAgent, CodeRefactorerAgent])

This ensures the code is written, *then* reviewed, and *finally* refactored, in a strict, dependable order. **The output from each sub-agent is passed to the next by storing them in state via** [**output\_key**](https://google.github.io/adk-docs/agents/llm-agents/).

**Code**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- 1. Define Sub-Agents for Each Pipeline Stage ---

# Code Writer Agent

# Takes the initial specification (from user query) and writes code.

code\_writer\_agent = LlmAgent(

name="CodeWriterAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction

instruction="""You are a Python Code Generator.

Based \*only\* on the user's request, write Python code that fulfills the requirement.

Output \*only\* the complete Python code block, enclosed in triple backticks (```python ... ```).

Do not add any other text before or after the code block.

""",

description="Writes initial Python code based on a specification.",

output\_key="generated\_code" # Stores output in state['generated\_code']

)

# Code Reviewer Agent

# Takes the code generated by the previous agent (read from state) and provides feedback.

code\_reviewer\_agent = LlmAgent(

name="CodeReviewerAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction, correctly using state key injection

instruction="""You are an expert Python Code Reviewer.

Your task is to provide constructive feedback on the provided code.

\*\*Code to Review:\*\*

```python

{generated\_code}

```

\*\*Review Criteria:\*\*

1. \*\*Correctness:\*\* Does the code work as intended? Are there logic errors?

2. \*\*Readability:\*\* Is the code clear and easy to understand? Follows PEP 8 style guidelines?

3. \*\*Efficiency:\*\* Is the code reasonably efficient? Any obvious performance bottlenecks?

4. \*\*Edge Cases:\*\* Does the code handle potential edge cases or invalid inputs gracefully?

5. \*\*Best Practices:\*\* Does the code follow common Python best practices?

\*\*Output:\*\*

Provide your feedback as a concise, bulleted list. Focus on the most important points for improvement.

If the code is excellent and requires no changes, simply state: "No major issues found."

Output \*only\* the review comments or the "No major issues" statement.

""",

description="Reviews code and provides feedback.",

output\_key="review\_comments", # Stores output in state['review\_comments']

)

# Code Refactorer Agent

# Takes the original code and the review comments (read from state) and refactors the code.

code\_refactorer\_agent = LlmAgent(

name="CodeRefactorerAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction, correctly using state key injection

instruction="""You are a Python Code Refactoring AI.

Your goal is to improve the given Python code based on the provided review comments.

\*\*Original Code:\*\*

```python

{generated\_code}

```

\*\*Review Comments:\*\*

{review\_comments}

\*\*Task:\*\*

Carefully apply the suggestions from the review comments to refactor the original code.

If the review comments state "No major issues found," return the original code unchanged.

Ensure the final code is complete, functional, and includes necessary imports and docstrings.

\*\*Output:\*\*

Output \*only\* the final, refactored Python code block, enclosed in triple backticks (```python ... ```).

Do not add any other text before or after the code block.

""",

description="Refactors code based on review comments.",

output\_key="refactored\_code", # Stores output in state['refactored\_code']

)

# --- 2. Create the SequentialAgent ---

# This agent orchestrates the pipeline by running the sub\_agents in order.

code\_pipeline\_agent = SequentialAgent(

name="CodePipelineAgent",

sub\_agents=[code\_writer\_agent, code\_reviewer\_agent, code\_refactorer\_agent],

description="Executes a sequence of code writing, reviewing, and refactoring.",

# The agents will run in the order provided: Writer -> Reviewer -> Refactorer

)

# For ADK tools compatibility, the root agent must be named `root\_agent`

root\_agent = code\_pipeline\_agent

Back to top

[Previous](https://google.github.io/adk-docs/agents/workflow-agents/)

[Workflow Agents](https://google.github.io/adk-docs/agents/workflow-agents/)

[Next](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)

[Loop agents](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)

Copyright Google 2025

Made with [Material for MkDocs](https://squidfunk.github.io/mkdocs-material/)

# Loop agents[¶](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/#loop-agents)

## The LoopAgent[¶](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/#the-loopagent)

The LoopAgent is a workflow agent that executes its sub-agents in a loop (i.e. iteratively). It ***repeatedly runs* a sequence of agents** for a specified number of iterations or until a termination condition is met.

Use the LoopAgent when your workflow involves repetition or iterative refinement, such as like revising code.

### Example[¶](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/#example)

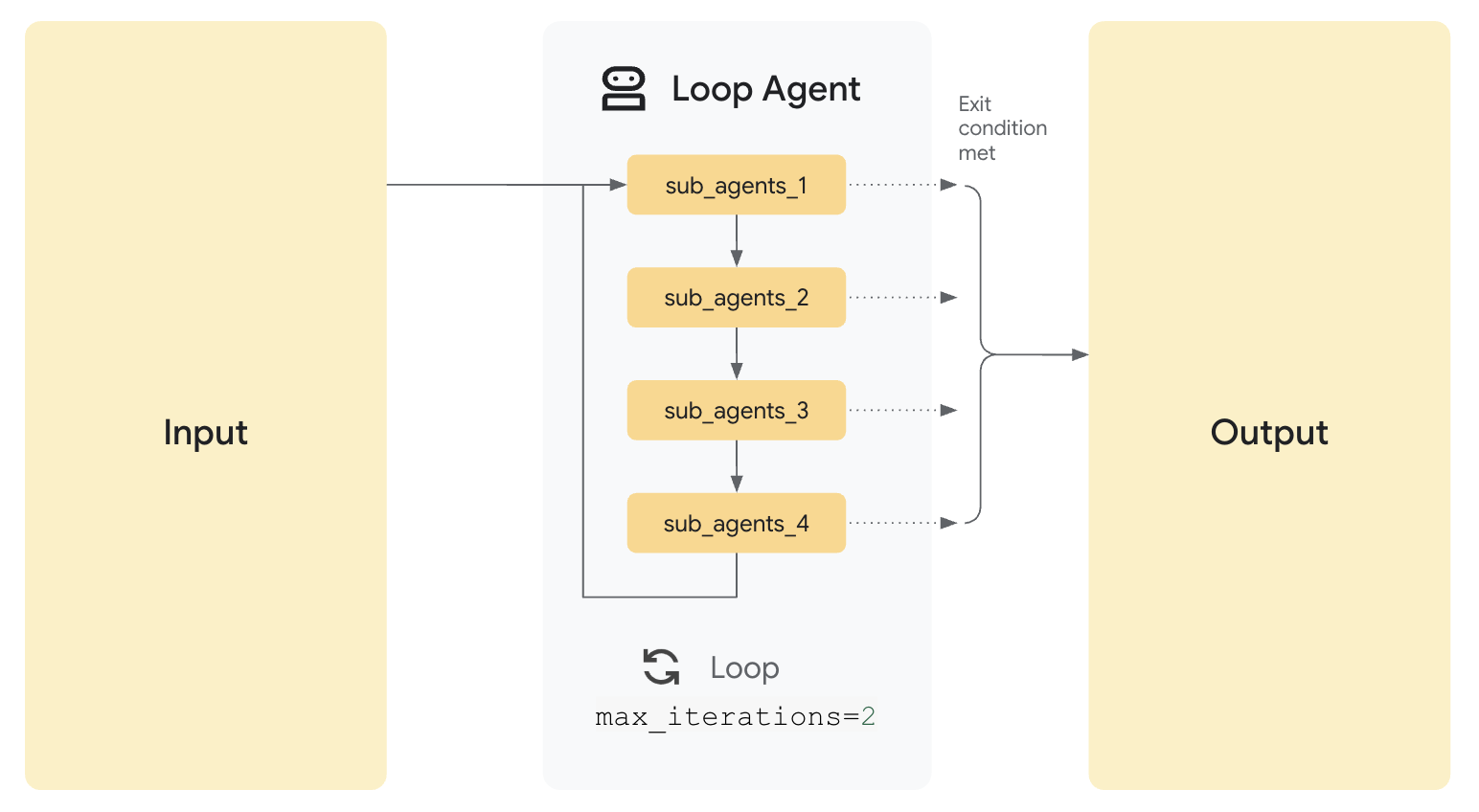
* You want to build an agent that can generate images of food, but sometimes when you want to generate a specific number of items (e.g. 5 bananas), it generates a different number of those items in the image (e.g. an image of 7 bananas). You have two tools: generate\_image, count\_food\_items. Because you want to keep generating images until it either correctly generates the specified number of items, or after a certain number of iterations, you should build your agent using a LoopAgent.

As with other [workflow agents](https://google.github.io/adk-docs/agents/workflow-agents/), the LoopAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are only concerned only with their execution (i.e. in a loop), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

### How it Works[¶](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/#how-it-works)

When the LoopAgent's run\_async() method is called, it performs the following actions:

1. **Sub-Agent Execution:** It iterates through the sub\_agents list *in order*. For *each* sub-agent, it calls the agent's run\_async() method.
2. **Termination Check:***Crucially*, the LoopAgent itself does *not* inherently decide when to stop looping. You *must* implement a termination mechanism to prevent infinite loops. Common strategies include:
   * **max\_iterations**: Set a maximum number of iterations in the LoopAgent. **The loop will terminate after that many iterations**.
   * **Escalation from sub-agent**: Design one or more sub-agents to evaluate a condition (e.g., "Is the document quality good enough?", "Has a consensus been reached?"). If the condition is met, the sub-agent can signal termination (e.g., by raising a custom event, setting a flag in a shared context, or returning a specific value).



### Full Example: Iterative Document Improvement[¶](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/#full-example-iterative-document-improvement)

Imagine a scenario where you want to iteratively improve a document:

* **Writer Agent:** An LlmAgent that generates or refines a draft on a topic.
* **Critic Agent:** An LlmAgent that critiques the draft, identifying areas for improvement.

LoopAgent(sub\_agents=[WriterAgent, CriticAgent], max\_iterations=5)

In this setup, the LoopAgent would manage the iterative process. The CriticAgent could be **designed to return a "STOP" signal when the document reaches a satisfactory quality level**, preventing further iterations. Alternatively, the max\_iterations parameter could be used to limit the process to a fixed number of cycles, or external logic could be implemented to make stop decisions. The **loop would run at most five times**, ensuring the iterative refinement doesn't continue indefinitely.

**Full Code**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- Constants ---

APP\_NAME = "doc\_writing\_app\_v3" # New App Name

USER\_ID = "dev\_user\_01"

SESSION\_ID\_BASE = "loop\_exit\_tool\_session" # New Base Session ID

GEMINI\_MODEL = "gemini-2.0-flash"

STATE\_INITIAL\_TOPIC = "initial\_topic"

# --- State Keys ---

STATE\_CURRENT\_DOC = "current\_document"

STATE\_CRITICISM = "criticism"

# Define the exact phrase the Critic should use to signal completion

COMPLETION\_PHRASE = "No major issues found."

# --- Tool Definition ---

def exit\_loop(tool\_context: ToolContext):

"""Call this function ONLY when the critique indicates no further changes are needed, signaling the iterative process should end."""

print(f" [Tool Call] exit\_loop triggered by {tool\_context.agent\_name}")

tool\_context.actions.escalate = True

# Return empty dict as tools should typically return JSON-serializable output

return {}

# --- Agent Definitions ---

# STEP 1: Initial Writer Agent (Runs ONCE at the beginning)

initial\_writer\_agent = LlmAgent(

name="InitialWriterAgent",

model=GEMINI\_MODEL,

include\_contents='none',

# MODIFIED Instruction: Ask for a slightly more developed start

instruction=f"""You are a Creative Writing Assistant tasked with starting a story.

Write the \*first draft\* of a short story (aim for 2-4 sentences).

Base the content \*only\* on the topic provided below. Try to introduce a specific element (like a character, a setting detail, or a starting action) to make it engaging.

Topic: {{initial\_topic}}

Output \*only\* the story/document text. Do not add introductions or explanations.

""",

description="Writes the initial document draft based on the topic, aiming for some initial substance.",

output\_key=STATE\_CURRENT\_DOC

)

# STEP 2a: Critic Agent (Inside the Refinement Loop)

critic\_agent\_in\_loop = LlmAgent(

name="CriticAgent",

model=GEMINI\_MODEL,

include\_contents='none',

# MODIFIED Instruction: More nuanced completion criteria, look for clear improvement paths.

instruction=f"""You are a Constructive Critic AI reviewing a short document draft (typically 2-6 sentences). Your goal is balanced feedback.

\*\*Document to Review:\*\*

```

{{current\_document}}

```

\*\*Task:\*\*

Review the document for clarity, engagement, and basic coherence according to the initial topic (if known).

IF you identify 1-2 \*clear and actionable\* ways the document could be improved to better capture the topic or enhance reader engagement (e.g., "Needs a stronger opening sentence", "Clarify the character's goal"):

Provide these specific suggestions concisely. Output \*only\* the critique text.

ELSE IF the document is coherent, addresses the topic adequately for its length, and has no glaring errors or obvious omissions:

Respond \*exactly\* with the phrase "{COMPLETION\_PHRASE}" and nothing else. It doesn't need to be perfect, just functionally complete for this stage. Avoid suggesting purely subjective stylistic preferences if the core is sound.

Do not add explanations. Output only the critique OR the exact completion phrase.

""",

description="Reviews the current draft, providing critique if clear improvements are needed, otherwise signals completion.",

output\_key=STATE\_CRITICISM

)

# STEP 2b: Refiner/Exiter Agent (Inside the Refinement Loop)

refiner\_agent\_in\_loop = LlmAgent(

name="RefinerAgent",

model=GEMINI\_MODEL,

# Relies solely on state via placeholders

include\_contents='none',

instruction=f"""You are a Creative Writing Assistant refining a document based on feedback OR exiting the process.

\*\*Current Document:\*\*

```

{{current\_document}}

```

\*\*Critique/Suggestions:\*\*

{{criticism}}

\*\*Task:\*\*

Analyze the 'Critique/Suggestions'.

IF the critique is \*exactly\* "{COMPLETION\_PHRASE}":

You MUST call the 'exit\_loop' function. Do not output any text.

ELSE (the critique contains actionable feedback):

Carefully apply the suggestions to improve the 'Current Document'. Output \*only\* the refined document text.

Do not add explanations. Either output the refined document OR call the exit\_loop function.

""",

description="Refines the document based on critique, or calls exit\_loop if critique indicates completion.",

tools=[exit\_loop], # Provide the exit\_loop tool

output\_key=STATE\_CURRENT\_DOC # Overwrites state['current\_document'] with the refined version

)

# STEP 2: Refinement Loop Agent

refinement\_loop = LoopAgent(

name="RefinementLoop",

# Agent order is crucial: Critique first, then Refine/Exit

sub\_agents=[

critic\_agent\_in\_loop,

refiner\_agent\_in\_loop,

],

max\_iterations=5 # Limit loops

)

# STEP 3: Overall Sequential Pipeline

# For ADK tools compatibility, the root agent must be named `root\_agent`

root\_agent = SequentialAgent(

name="IterativeWritingPipeline",

sub\_agents=[

initial\_writer\_agent, # Run first to create initial doc

refinement\_loop # Then run the critique/refine loop

],

description="Writes an initial document and then iteratively refines it with critique using an exit tool."

)

Back to top

[Previous](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/)

[Sequential agents](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/)

[Next](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)

[Parallel agents](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)

Copyright Google 2025

Made with [Material for MkDocs](https://squidfunk.github.io/mkdocs-material/)

# Parallel agents[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#parallel-agents)

## The ParallelAgent[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#the-parallelagent)

The ParallelAgent is a [workflow agent](https://google.github.io/adk-docs/agents/workflow-agents/) that executes its sub-agents *concurrently*. This dramatically speeds up workflows where tasks can be performed independently.

Use ParallelAgent when: For scenarios prioritizing speed and involving independent, resource-intensive tasks, a ParallelAgent facilitates efficient parallel execution. **When sub-agents operate without dependencies, their tasks can be performed concurrently**, significantly reducing overall processing time.

As with other [workflow agents](https://google.github.io/adk-docs/agents/workflow-agents/), the ParallelAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are only concerned with their execution (i.e. executing sub-agents in parallel), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

### Example[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#example)

This approach is particularly beneficial for operations like multi-source data retrieval or heavy computations, where parallelization yields substantial performance gains. Importantly, this strategy assumes no inherent need for shared state or direct information exchange between the concurrently executing agents.

### How it works[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#how-it-works)

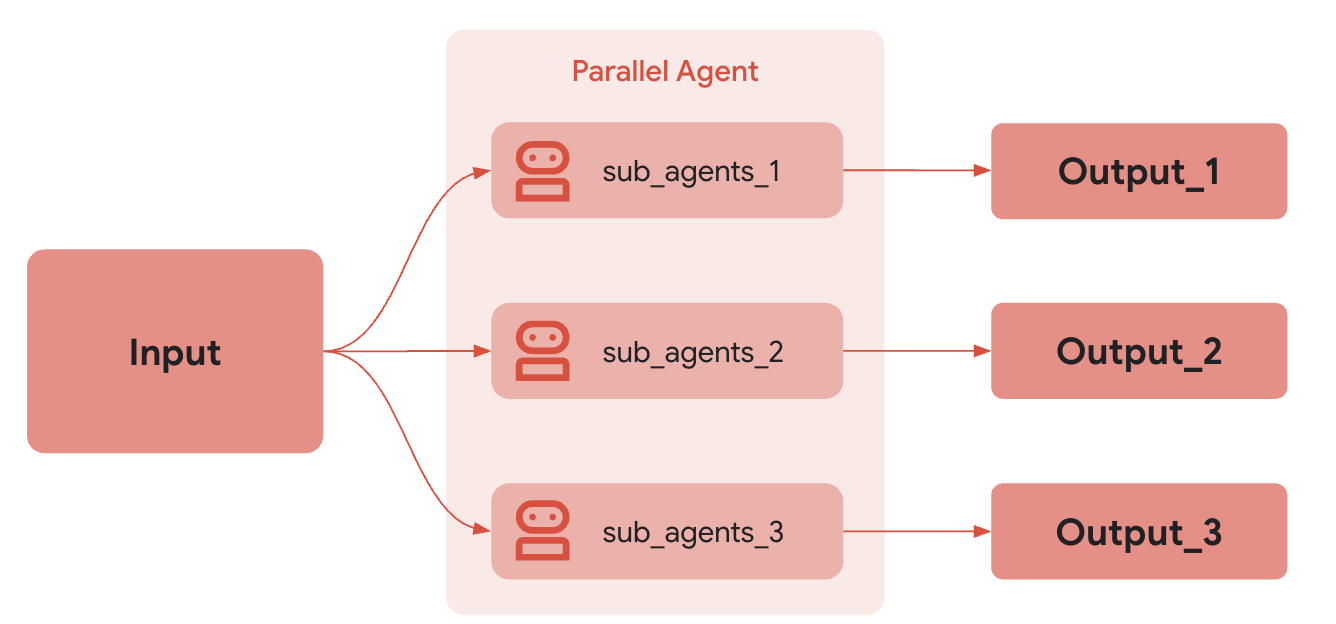
When the ParallelAgent's run\_async() method is called:

1. **Concurrent Execution:** It initiates the run\_async() method of *each* sub-agent present in the sub\_agents list *concurrently*. This means all the agents start running at (approximately) the same time.
2. **Independent Branches:** Each sub-agent operates in its own execution branch. There is ***no* automatic sharing of conversation history or state between these branches** during execution.
3. **Result Collection:** The ParallelAgent manages the parallel execution and, typically, provides a way to access the results from each sub-agent after they have completed (e.g., through a list of results or events). The order of results may not be deterministic.

### Independent Execution and State Management[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#independent-execution-and-state-management)

It's *crucial* to understand that sub-agents within a ParallelAgent run independently. If you *need* communication or data sharing between these agents, you must implement it explicitly. Possible approaches include:

* **Shared InvocationContext:** You could pass a shared InvocationContext object to each sub-agent. This object could act as a shared data store. However, you'd need to manage concurrent access to this shared context carefully (e.g., using locks) to avoid race conditions.
* **External State Management:** Use an external database, message queue, or other mechanism to manage shared state and facilitate communication between agents.
* **Post-Processing:** Collect results from each branch, and then implement logic to coordinate data afterwards.



### Full Example: Parallel Web Research[¶](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/#full-example-parallel-web-research)

Imagine researching multiple topics simultaneously:

1. **Researcher Agent 1:** An LlmAgent that researches "renewable energy sources."
2. **Researcher Agent 2:** An LlmAgent that researches "electric vehicle technology."
3. **Researcher Agent 3:** An LlmAgent that researches "carbon capture methods."

ParallelAgent(sub\_agents=[ResearcherAgent1, ResearcherAgent2, ResearcherAgent3])

These research tasks are independent. Using a ParallelAgent allows them to run concurrently, potentially reducing the total research time significantly compared to running them sequentially. The results from each agent would be collected separately after they finish.

**Code**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- 1. Define Researcher Sub-Agents (to run in parallel) ---

# Researcher 1: Renewable Energy

researcher\_agent\_1 = LlmAgent(

name="RenewableEnergyResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in energy.

Research the latest advancements in 'renewable energy sources'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches renewable energy sources.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="renewable\_energy\_result"

)

# Researcher 2: Electric Vehicles

researcher\_agent\_2 = LlmAgent(

name="EVResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in transportation.

Research the latest developments in 'electric vehicle technology'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches electric vehicle technology.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="ev\_technology\_result"

)

# Researcher 3: Carbon Capture

researcher\_agent\_3 = LlmAgent(

name="CarbonCaptureResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in climate solutions.

Research the current state of 'carbon capture methods'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches carbon capture methods.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="carbon\_capture\_result"

)

# --- 2. Create the ParallelAgent (Runs researchers concurrently) ---

# This agent orchestrates the concurrent execution of the researchers.

# It finishes once all researchers have completed and stored their results in state.

parallel\_research\_agent = ParallelAgent(

name="ParallelWebResearchAgent",

sub\_agents=[researcher\_agent\_1, researcher\_agent\_2, researcher\_agent\_3],

description="Runs multiple research agents in parallel to gather information."

)

# --- 3. Define the Merger Agent (Runs \*after\* the parallel agents) ---

# This agent takes the results stored in the session state by the parallel agents

# and synthesizes them into a single, structured response with attributions.

merger\_agent = LlmAgent(

name="SynthesisAgent",

model=GEMINI\_MODEL, # Or potentially a more powerful model if needed for synthesis

instruction="""You are an AI Assistant responsible for combining research findings into a structured report.

Your primary task is to synthesize the following research summaries, clearly attributing findings to their source areas. Structure your response using headings for each topic. Ensure the report is coherent and integrates the key points smoothly.

\*\*Crucially: Your entire response MUST be grounded \*exclusively\* on the information provided in the 'Input Summaries' below. Do NOT add any external knowledge, facts, or details not present in these specific summaries.\*\*

\*\*Input Summaries:\*\*

\* \*\*Renewable Energy:\*\*

{renewable\_energy\_result}

\* \*\*Electric Vehicles:\*\*

{ev\_technology\_result}

\* \*\*Carbon Capture:\*\*

{carbon\_capture\_result}

\*\*Output Format:\*\*

## Summary of Recent Sustainable Technology Advancements

### Renewable Energy Findings

(Based on RenewableEnergyResearcher's findings)

[Synthesize and elaborate \*only\* on the renewable energy input summary provided above.]

### Electric Vehicle Findings

(Based on EVResearcher's findings)

[Synthesize and elaborate \*only\* on the EV input summary provided above.]

### Carbon Capture Findings

(Based on CarbonCaptureResearcher's findings)

[Synthesize and elaborate \*only\* on the carbon capture input summary provided above.]

### Overall Conclusion

[Provide a brief (1-2 sentence) concluding statement that connects \*only\* the findings presented above.]

Output \*only\* the structured report following this format. Do not include introductory or concluding phrases outside this structure, and strictly adhere to using only the provided input summary content.

""",

description="Combines research findings from parallel agents into a structured, cited report, strictly grounded on provided inputs.",

# No tools needed for merging

# No output\_key needed here, as its direct response is the final output of the sequence

)

# --- 4. Create the SequentialAgent (Orchestrates the overall flow) ---

# This is the main agent that will be run. It first executes the ParallelAgent

# to populate the state, and then executes the MergerAgent to produce the final output.

sequential\_pipeline\_agent = SequentialAgent(

name="ResearchAndSynthesisPipeline",

# Run parallel research first, then merge

sub\_agents=[parallel\_research\_agent, merger\_agent],

description="Coordinates parallel research and synthesizes the results."

)

root\_agent = sequential\_pipeline\_agent

Back to top

[Previous](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)

[Loop agents](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)

[Next](https://google.github.io/adk-docs/agents/custom-agents/)

[Custom agents](https://google.github.io/adk-docs/agents/custom-agents/)

Copyright Google 2025

Made with [Material for MkDocs](https://squidfunk.github.io/mkdocs-material/)

**Advanced Concept**

Building custom agents by directly implementing \_run\_async\_impl provides powerful control but is more complex than using the predefined LlmAgent or standard WorkflowAgent types. We recommend understanding those foundational agent types first before tackling custom orchestration logic.

# Custom agents[¶](https://google.github.io/adk-docs/agents/custom-agents/#custom-agents)

Custom agents provide the ultimate flexibility in ADK, allowing you to define **arbitrary orchestration logic** by inheriting directly from BaseAgent and implementing your own control flow. This goes beyond the predefined patterns of SequentialAgent, LoopAgent, and ParallelAgent, enabling you to build highly specific and complex agentic workflows.

## Introduction: Beyond Predefined Workflows[¶](https://google.github.io/adk-docs/agents/custom-agents/#introduction-beyond-predefined-workflows)

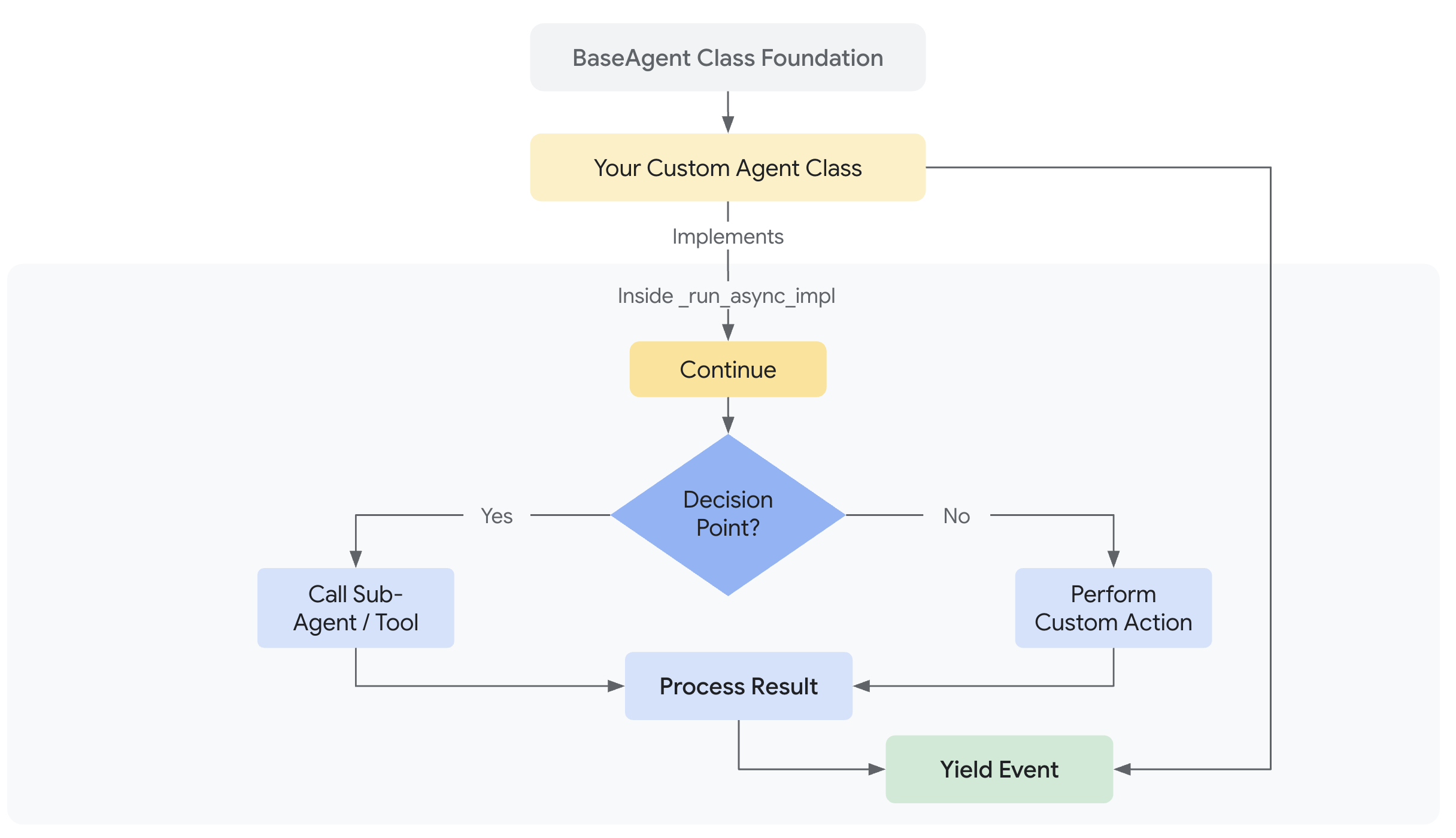
### What is a Custom Agent?[¶](https://google.github.io/adk-docs/agents/custom-agents/#what-is-a-custom-agent)

A Custom Agent is essentially any class you create that inherits from google.adk.agents.BaseAgent and implements its core execution logic within the \_run\_async\_impl asynchronous method. You have complete control over how this method calls other agents (sub-agents), manages state, and handles events.

### Why Use Them?[¶](https://google.github.io/adk-docs/agents/custom-agents/#why-use-them)

While the standard [Workflow Agents](https://google.github.io/adk-docs/agents/workflow-agents/) (SequentialAgent, LoopAgent, ParallelAgent) cover common orchestration patterns, you'll need a Custom agent when your requirements include:

* **Conditional Logic:** Executing different sub-agents or taking different paths based on runtime conditions or the results of previous steps.
* **Complex State Management:** Implementing intricate logic for maintaining and updating state throughout the workflow beyond simple sequential passing.
* **External Integrations:** Incorporating calls to external APIs, databases, or custom Python libraries directly within the orchestration flow control.
* **Dynamic Agent Selection:** Choosing which sub-agent(s) to run next based on dynamic evaluation of the situation or input.
* **Unique Workflow Patterns:** Implementing orchestration logic that doesn't fit the standard sequential, parallel, or loop structures.



## Implementing Custom Logic:[¶](https://google.github.io/adk-docs/agents/custom-agents/#implementing-custom-logic)

The heart of any custom agent is the \_run\_async\_impl method. This is where you define its unique behavior.

* **Signature:** async def \_run\_async\_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:
* **Asynchronous Generator:** It must be an async def function and return an AsyncGenerator. This allows it to yield events produced by sub-agents or its own logic back to the runner.
* **ctx (InvocationContext):** Provides access to crucial runtime information, most importantly ctx.session.state, which is the primary way to share data between steps orchestrated by your custom agent.

**Key Capabilities within \_run\_async\_impl:**

1. **Calling Sub-Agents:** You invoke sub-agents (which are typically stored as instance attributes like self.my\_llm\_agent) using their run\_async method and yield their events:

async for event in self.some\_sub\_agent.run\_async(ctx):

# Optionally inspect or log the event

yield event # Pass the event up

1. **Managing State:** Read from and write to the session state dictionary (ctx.session.state) to pass data between sub-agent calls or make decisions:

# Read data set by a previous agent

previous\_result = ctx.session.state.get("some\_key")

# Make a decision based on state

if previous\_result == "some\_value":

# ... call a specific sub-agent ...

else:

# ... call another sub-agent ...

# Store a result for a later step (often done via a sub-agent's output\_key)

# ctx.session.state["my\_custom\_result"] = "calculated\_value"

1. **Implementing Control Flow:** Use standard Python constructs (if/elif/else, for/while loops, try/except) to create sophisticated, conditional, or iterative workflows involving your sub-agents.

## Managing Sub-Agents and State[¶](https://google.github.io/adk-docs/agents/custom-agents/#managing-sub-agents-and-state)

Typically, a custom agent orchestrates other agents (like LlmAgent, LoopAgent, etc.).

* **Initialization:** You usually pass instances of these sub-agents into your custom agent's \_\_init\_\_ method and store them as instance attributes (e.g., self.story\_generator = story\_generator\_instance). This makes them accessible within \_run\_async\_impl.
* **sub\_agents List:** When initializing the BaseAgent using super().\_\_init\_\_(...), you should pass a sub\_agents list. This list tells the ADK framework about the agents that are part of this custom agent's immediate hierarchy. It's important for framework features like lifecycle management, introspection, and potentially future routing capabilities, even if your \_run\_async\_impl calls the agents directly via self.xxx\_agent. Include the agents that your custom logic directly invokes at the top level.
* **State:** As mentioned, ctx.session.state is the standard way sub-agents (especially LlmAgents using output\_key) communicate results back to the orchestrator and how the orchestrator passes necessary inputs down.

## Design Pattern Example: StoryFlowAgent[¶](https://google.github.io/adk-docs/agents/custom-agents/#design-pattern-example-storyflowagent)

Let's illustrate the power of custom agents with an example pattern: a multi-stage content generation workflow with conditional logic.

**Goal:** Create a system that generates a story, iteratively refines it through critique and revision, performs final checks, and crucially, *regenerates the story if the final tone check fails*.

**Why Custom?** The core requirement driving the need for a custom agent here is the **conditional regeneration based on the tone check**. Standard workflow agents don't have built-in conditional branching based on the outcome of a sub-agent's task. We need custom Python logic (if tone == "negative": ...) within the orchestrator.

### Part 1: Simplified custom agent Initialization[¶](https://google.github.io/adk-docs/agents/custom-agents/#part-1-simplified-custom-agent-initialization)

We define the StoryFlowAgent inheriting from BaseAgent. In \_\_init\_\_, we store the necessary sub-agents (passed in) as instance attributes and tell the BaseAgent framework about the top-level agents this custom agent will directly orchestrate.

class StoryFlowAgent(BaseAgent):

"""

Custom agent for a story generation and refinement workflow.

This agent orchestrates a sequence of LLM agents to generate a story,

critique it, revise it, check grammar and tone, and potentially

regenerate the story if the tone is negative.

"""

# --- Field Declarations for Pydantic ---

# Declare the agents passed during initialization as class attributes with type hints

story\_generator: LlmAgent

critic: LlmAgent

reviser: LlmAgent

grammar\_check: LlmAgent

tone\_check: LlmAgent

loop\_agent: LoopAgent

sequential\_agent: SequentialAgent

# model\_config allows setting Pydantic configurations if needed, e.g., arbitrary\_types\_allowed

model\_config = {"arbitrary\_types\_allowed": True}

def \_\_init\_\_(

self,

name: str,

story\_generator: LlmAgent,

critic: LlmAgent,

reviser: LlmAgent,

grammar\_check: LlmAgent,

tone\_check: LlmAgent,

):

"""

Initializes the StoryFlowAgent.

Args:

name: The name of the agent.

story\_generator: An LlmAgent to generate the initial story.

critic: An LlmAgent to critique the story.

reviser: An LlmAgent to revise the story based on criticism.

grammar\_check: An LlmAgent to check the grammar.

tone\_check: An LlmAgent to analyze the tone.

"""

# Create internal agents \*before\* calling super().\_\_init\_\_

loop\_agent = LoopAgent(

name="CriticReviserLoop", sub\_agents=[critic, reviser], max\_iterations=2

)

sequential\_agent = SequentialAgent(

name="PostProcessing", sub\_agents=[grammar\_check, tone\_check]

)

# Define the sub\_agents list for the framework

sub\_agents\_list = [

story\_generator,

loop\_agent,

sequential\_agent,

]

# Pydantic will validate and assign them based on the class annotations.

super().\_\_init\_\_(

name=name,

story\_generator=story\_generator,

critic=critic,

reviser=reviser,

grammar\_check=grammar\_check,

tone\_check=tone\_check,

loop\_agent=loop\_agent,

sequential\_agent=sequential\_agent,

sub\_agents=sub\_agents\_list, # Pass the sub\_agents list directly

)

### Part 2: Defining the Custom Execution Logic[¶](https://google.github.io/adk-docs/agents/custom-agents/#part-2-defining-the-custom-execution-logic)

This method orchestrates the sub-agents using standard Python async/await and control flow.

@override

async def \_run\_async\_impl(

self, ctx: InvocationContext

) -> AsyncGenerator[Event, None]:

"""

Implements the custom orchestration logic for the story workflow.

Uses the instance attributes assigned by Pydantic (e.g., self.story\_generator).

"""

logger.info(f"[{self.name}] Starting story generation workflow.")

# 1. Initial Story Generation

logger.info(f"[{self.name}] Running StoryGenerator...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# Check if story was generated before proceeding

if "current\_story" not in ctx.session.state or not ctx.session.state["current\_story"]:

logger.error(f"[{self.name}] Failed to generate initial story. Aborting workflow.")

return # Stop processing if initial story failed

logger.info(f"[{self.name}] Story state after generator: {ctx.session.state.get('current\_story')}")

# 2. Critic-Reviser Loop

logger.info(f"[{self.name}] Running CriticReviserLoop...")

# Use the loop\_agent instance attribute assigned during init

async for event in self.loop\_agent.run\_async(ctx):

logger.info(f"[{self.name}] Event from CriticReviserLoop: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

logger.info(f"[{self.name}] Story state after loop: {ctx.session.state.get('current\_story')}")

# 3. Sequential Post-Processing (Grammar and Tone Check)

logger.info(f"[{self.name}] Running PostProcessing...")

# Use the sequential\_agent instance attribute assigned during init

async for event in self.sequential\_agent.run\_async(ctx):

logger.info(f"[{self.name}] Event from PostProcessing: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# 4. Tone-Based Conditional Logic

tone\_check\_result = ctx.session.state.get("tone\_check\_result")

logger.info(f"[{self.name}] Tone check result: {tone\_check\_result}")

if tone\_check\_result == "negative":

logger.info(f"[{self.name}] Tone is negative. Regenerating story...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator (Regen): {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

else:

logger.info(f"[{self.name}] Tone is not negative. Keeping current story.")

pass

logger.info(f"[{self.name}] Workflow finished.")

**Explanation of Logic:**

1. The initial story\_generator runs. Its output is expected to be in ctx.session.state["current\_story"].
2. The loop\_agent runs, which internally calls the critic and reviser sequentially for max\_iterations times. They read/write current\_story and criticism from/to the state.
3. The sequential\_agent runs, calling grammar\_check then tone\_check, reading current\_story and writing grammar\_suggestions and tone\_check\_result to the state.
4. **Custom Part:** The if statement checks the tone\_check\_result from the state. If it's "negative", the story\_generator is called *again*, overwriting the current\_story in the state. Otherwise, the flow ends.

### Part 3: Defining the LLM Sub-Agents[¶](https://google.github.io/adk-docs/agents/custom-agents/#part-3-defining-the-llm-sub-agents)

These are standard LlmAgent definitions, responsible for specific tasks. Their output\_key parameter is crucial for placing results into the session.state where other agents or the custom orchestrator can access them.

GEMINI\_2\_FLASH = "gemini-2.0-flash" # Define model constant

# --- Define the individual LLM agents ---

story\_generator = LlmAgent(

name="StoryGenerator",

model=GEMINI\_2\_FLASH,

instruction="""You are a story writer. Write a short story (around 100 words) about a cat,

based on the topic provided in session state with key 'topic'""",

input\_schema=None,

output\_key="current\_story", # Key for storing output in session state

)

critic = LlmAgent(

name="Critic",

model=GEMINI\_2\_FLASH,

instruction="""You are a story critic. Review the story provided in

session state with key 'current\_story'. Provide 1-2 sentences of constructive criticism

on how to improve it. Focus on plot or character.""",

input\_schema=None,

output\_key="criticism", # Key for storing criticism in session state

)

reviser = LlmAgent(

name="Reviser",

model=GEMINI\_2\_FLASH,

instruction="""You are a story reviser. Revise the story provided in

session state with key 'current\_story', based on the criticism in

session state with key 'criticism'. Output only the revised story.""",

input\_schema=None,

output\_key="current\_story", # Overwrites the original story

)

grammar\_check = LlmAgent(

name="GrammarCheck",

model=GEMINI\_2\_FLASH,

instruction="""You are a grammar checker. Check the grammar of the story

provided in session state with key 'current\_story'. Output only the suggested

corrections as a list, or output 'Grammar is good!' if there are no errors.""",

input\_schema=None,

output\_key="grammar\_suggestions",

)

tone\_check = LlmAgent(

name="ToneCheck",

model=GEMINI\_2\_FLASH,

instruction="""You are a tone analyzer. Analyze the tone of the story

provided in session state with key 'current\_story'. Output only one word: 'positive' if

the tone is generally positive, 'negative' if the tone is generally negative, or 'neutral'

otherwise.""",

input\_schema=None,

output\_key="tone\_check\_result", # This agent's output determines the conditional flow

)

### Part 4: Instantiating and Running the custom agent[¶](https://google.github.io/adk-docs/agents/custom-agents/#part-4-instantiating-and-running-the-custom-agent)

Finally, you instantiate your StoryFlowAgent and use the Runner as usual.

# --- Create the custom agent instance ---

story\_flow\_agent = StoryFlowAgent(

name="StoryFlowAgent",

story\_generator=story\_generator,

critic=critic,

reviser=reviser,

grammar\_check=grammar\_check,

tone\_check=tone\_check,

)

# --- Setup Runner and Session ---

session\_service = InMemorySessionService()

initial\_state = {"topic": "a brave kitten exploring a haunted house"}

session = session\_service.create\_session(

app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID,

state=initial\_state # Pass initial state here

)

logger.info(f"Initial session state: {session.state}")

runner = Runner(

agent=story\_flow\_agent, # Pass the custom orchestrator agent

app\_name=APP\_NAME,

session\_service=session\_service

)

# --- Function to Interact with the Agent ---

def call\_agent(user\_input\_topic: str):

"""

Sends a new topic to the agent (overwriting the initial one if needed)

and runs the workflow.

"""

current\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

if not current\_session:

logger.error("Session not found!")

return

current\_session.state["topic"] = user\_input\_topic

logger.info(f"Updated session state topic to: {user\_input\_topic}")

content = types.Content(role='user', parts=[types.Part(text=f"Generate a story about: {user\_input\_topic}")])

events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)

final\_response = "No final response captured."

for event in events:

if event.is\_final\_response() and event.content and event.content.parts:

logger.info(f"Potential final response from [{event.author}]: {event.content.parts[0].text}")

final\_response = event.content.parts[0].text

print("\n--- Agent Interaction Result ---")

print("Agent Final Response: ", final\_response)

final\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

print("Final Session State:")

import json

print(json.dumps(final\_session.state, indent=2))

print("-------------------------------\n")

# --- Run the Agent ---

call\_agent("a lonely robot finding a friend in a junkyard")

*(Note: The full runnable code, including imports and execution logic, can be found linked below.)*

## Full Code Example[¶](https://google.github.io/adk-docs/agents/custom-agents/#full-code-example)

**Storyflow Agent**

# Full runnable code for the StoryFlowAgent example

import logging

from typing import AsyncGenerator

from typing\_extensions import override

from google.adk.agents import LlmAgent, BaseAgent, LoopAgent, SequentialAgent

from google.adk.agents.invocation\_context import InvocationContext

from google.genai import types

from google.adk.sessions import InMemorySessionService

from google.adk.runners import Runner

from google.adk.events import Event

from pydantic import BaseModel, Field

# --- Constants ---

APP\_NAME = "story\_app"

USER\_ID = "12345"

SESSION\_ID = "123344"

GEMINI\_2\_FLASH = "gemini-2.0-flash"

# --- Configure Logging ---

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

# --- Custom Orchestrator Agent ---

class StoryFlowAgent(BaseAgent):

"""

Custom agent for a story generation and refinement workflow.

This agent orchestrates a sequence of LLM agents to generate a story,

critique it, revise it, check grammar and tone, and potentially

regenerate the story if the tone is negative.

"""

# --- Field Declarations for Pydantic ---

# Declare the agents passed during initialization as class attributes with type hints

story\_generator: LlmAgent

critic: LlmAgent

reviser: LlmAgent

grammar\_check: LlmAgent

tone\_check: LlmAgent

loop\_agent: LoopAgent

sequential\_agent: SequentialAgent

# model\_config allows setting Pydantic configurations if needed, e.g., arbitrary\_types\_allowed

model\_config = {"arbitrary\_types\_allowed": True}

def \_\_init\_\_(

self,

name: str,

story\_generator: LlmAgent,

critic: LlmAgent,

reviser: LlmAgent,

grammar\_check: LlmAgent,

tone\_check: LlmAgent,

):

"""

Initializes the StoryFlowAgent.

Args:

name: The name of the agent.

story\_generator: An LlmAgent to generate the initial story.

critic: An LlmAgent to critique the story.

reviser: An LlmAgent to revise the story based on criticism.

grammar\_check: An LlmAgent to check the grammar.

tone\_check: An LlmAgent to analyze the tone.

"""

# Create internal agents \*before\* calling super().\_\_init\_\_

loop\_agent = LoopAgent(

name="CriticReviserLoop", sub\_agents=[critic, reviser], max\_iterations=2

)

sequential\_agent = SequentialAgent(

name="PostProcessing", sub\_agents=[grammar\_check, tone\_check]

)

# Define the sub\_agents list for the framework

sub\_agents\_list = [

story\_generator,

loop\_agent,

sequential\_agent,

]

# Pydantic will validate and assign them based on the class annotations.

super().\_\_init\_\_(

name=name,

story\_generator=story\_generator,

critic=critic,

reviser=reviser,

grammar\_check=grammar\_check,

tone\_check=tone\_check,

loop\_agent=loop\_agent,

sequential\_agent=sequential\_agent,

sub\_agents=sub\_agents\_list, # Pass the sub\_agents list directly

)

@override

async def \_run\_async\_impl(

self, ctx: InvocationContext

) -> AsyncGenerator[Event, None]:

"""

Implements the custom orchestration logic for the story workflow.

Uses the instance attributes assigned by Pydantic (e.g., self.story\_generator).

"""

logger.info(f"[{self.name}] Starting story generation workflow.")

# 1. Initial Story Generation

logger.info(f"[{self.name}] Running StoryGenerator...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# Check if story was generated before proceeding

if "current\_story" not in ctx.session.state or not ctx.session.state["current\_story"]:

logger.error(f"[{self.name}] Failed to generate initial story. Aborting workflow.")

return # Stop processing if initial story failed

logger.info(f"[{self.name}] Story state after generator: {ctx.session.state.get('current\_story')}")

# 2. Critic-Reviser Loop

logger.info(f"[{self.name}] Running CriticReviserLoop...")

# Use the loop\_agent instance attribute assigned during init

async for event in self.loop\_agent.run\_async(ctx):

logger.info(f"[{self.name}] Event from CriticReviserLoop: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

logger.info(f"[{self.name}] Story state after loop: {ctx.session.state.get('current\_story')}")

# 3. Sequential Post-Processing (Grammar and Tone Check)

logger.info(f"[{self.name}] Running PostProcessing...")

# Use the sequential\_agent instance attribute assigned during init

async for event in self.sequential\_agent.run\_async(ctx):

logger.info(f"[{self.name}] Event from PostProcessing: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# 4. Tone-Based Conditional Logic

tone\_check\_result = ctx.session.state.get("tone\_check\_result")

logger.info(f"[{self.name}] Tone check result: {tone\_check\_result}")

if tone\_check\_result == "negative":

logger.info(f"[{self.name}] Tone is negative. Regenerating story...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator (Regen): {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

else:

logger.info(f"[{self.name}] Tone is not negative. Keeping current story.")

pass

logger.info(f"[{self.name}] Workflow finished.")

# --- Define the individual LLM agents ---

story\_generator = LlmAgent(

name="StoryGenerator",

model=GEMINI\_2\_FLASH,

instruction="""You are a story writer. Write a short story (around 100 words) about a cat,

based on the topic provided in session state with key 'topic'""",

input\_schema=None,

output\_key="current\_story", # Key for storing output in session state

)

critic = LlmAgent(

name="Critic",

model=GEMINI\_2\_FLASH,

instruction="""You are a story critic. Review the story provided in

session state with key 'current\_story'. Provide 1-2 sentences of constructive criticism

on how to improve it. Focus on plot or character.""",

input\_schema=None,

output\_key="criticism", # Key for storing criticism in session state

)

reviser = LlmAgent(

name="Reviser",

model=GEMINI\_2\_FLASH,

instruction="""You are a story reviser. Revise the story provided in

session state with key 'current\_story', based on the criticism in

session state with key 'criticism'. Output only the revised story.""",

input\_schema=None,

output\_key="current\_story", # Overwrites the original story

)

grammar\_check = LlmAgent(

name="GrammarCheck",

model=GEMINI\_2\_FLASH,

instruction="""You are a grammar checker. Check the grammar of the story

provided in session state with key 'current\_story'. Output only the suggested

corrections as a list, or output 'Grammar is good!' if there are no errors.""",

input\_schema=None,

output\_key="grammar\_suggestions",

)

tone\_check = LlmAgent(

name="ToneCheck",

model=GEMINI\_2\_FLASH,

instruction="""You are a tone analyzer. Analyze the tone of the story

provided in session state with key 'current\_story'. Output only one word: 'positive' if

the tone is generally positive, 'negative' if the tone is generally negative, or 'neutral'

otherwise.""",

input\_schema=None,

output\_key="tone\_check\_result", # This agent's output determines the conditional flow

)

# --- Create the custom agent instance ---

story\_flow\_agent = StoryFlowAgent(

name="StoryFlowAgent",

story\_generator=story\_generator,

critic=critic,

reviser=reviser,

grammar\_check=grammar\_check,

tone\_check=tone\_check,

)

# --- Setup Runner and Session ---

session\_service = InMemorySessionService()

initial\_state = {"topic": "a brave kitten exploring a haunted house"}

session = session\_service.create\_session(

app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID,

state=initial\_state # Pass initial state here

)

logger.info(f"Initial session state: {session.state}")

runner = Runner(

agent=story\_flow\_agent, # Pass the custom orchestrator agent

app\_name=APP\_NAME,

session\_service=session\_service

)

# --- Function to Interact with the Agent ---

def call\_agent(user\_input\_topic: str):

"""

Sends a new topic to the agent (overwriting the initial one if needed)

and runs the workflow.

"""

current\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

if not current\_session:

logger.error("Session not found!")

return

current\_session.state["topic"] = user\_input\_topic

logger.info(f"Updated session state topic to: {user\_input\_topic}")

content = types.Content(role='user', parts=[types.Part(text=f"Generate a story about: {user\_input\_topic}")])

events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)

final\_response = "No final response captured."

for event in events:

if event.is\_final\_response() and event.content and event.content.parts:

logger.info(f"Potential final response from [{event.author}]: {event.content.parts[0].text}")

final\_response = event.content.parts[0].text

print("\n--- Agent Interaction Result ---")

print("Agent Final Response: ", final\_response)

final\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

print("Final Session State:")

import json

print(json.dumps(final\_session.state, indent=2))

print("-------------------------------\n")

# --- Run the Agent ---

call\_agent("a lonely robot finding a friend in a junkyard")

Back to top

[Previous](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)

[Parallel agents](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)

[Next](https://google.github.io/adk-docs/agents/multi-agents/)

[Multi-agent systems](https://google.github.io/adk-docs/agents/multi-agents/)

Copyright Google 2025

Made with [Material for MkDocs](https://squidfunk.github.io/mkdocs-material/)

# Multi-Agent Systems in ADK[¶](https://google.github.io/adk-docs/agents/multi-agents/#multi-agent-systems-in-adk)

As agentic applications grow in complexity, structuring them as a single, monolithic agent can become challenging to develop, maintain, and reason about. The Agent Development Kit (ADK) supports building sophisticated applications by composing multiple, distinct BaseAgent instances into a **Multi-Agent System (MAS)**.

In ADK, a multi-agent system is an application where different agents, often forming a hierarchy, collaborate or coordinate to achieve a larger goal. Structuring your application this way offers significant advantages, including enhanced modularity, specialization, reusability, maintainability, and the ability to define structured control flows using dedicated workflow agents.

You can compose various types of agents derived from BaseAgent to build these systems:

* **LLM Agents:** Agents powered by large language models. (See [LLM Agents](https://google.github.io/adk-docs/agents/llm-agents/))
* **Workflow Agents:** Specialized agents (SequentialAgent, ParallelAgent, LoopAgent) designed to manage the execution flow of their sub-agents. (See [Workflow Agents](https://google.github.io/adk-docs/agents/workflow-agents/))
* **Custom agents:** Your own agents inheriting from BaseAgent with specialized, non-LLM logic. (See [Custom Agents](https://google.github.io/adk-docs/agents/custom-agents/))

The following sections detail the core ADK primitives—such as agent hierarchy, workflow agents, and interaction mechanisms—that enable you to construct and manage these multi-agent systems effectively.

## 2. ADK Primitives for Agent Composition[¶](https://google.github.io/adk-docs/agents/multi-agents/#2-adk-primitives-for-agent-composition)

ADK provides core building blocks—primitives—that enable you to structure and manage interactions within your multi-agent system.

### 2.1. Agent Hierarchy (parent\_agent, sub\_agents)[¶](https://google.github.io/adk-docs/agents/multi-agents/#21-agent-hierarchy-parent_agent-sub_agents)

The foundation for structuring multi-agent systems is the parent-child relationship defined in BaseAgent.

* **Establishing Hierarchy:** You create a tree structure by passing a list of agent instances to the sub\_agents argument when initializing a parent agent. ADK automatically sets the parent\_agent attribute on each child agent during initialization (google.adk.agents.base\_agent.py - model\_post\_init).
* **Single Parent Rule:** An agent instance can only be added as a sub-agent once. Attempting to assign a second parent will result in a ValueError.
* **Importance:** This hierarchy defines the scope for [Workflow Agents](https://google.github.io/adk-docs/agents/multi-agents/#22-workflow-agents-as-orchestrators) and influences the potential targets for LLM-Driven Delegation. You can navigate the hierarchy using agent.parent\_agent or find descendants using agent.find\_agent(name).

# Conceptual Example: Defining Hierarchy

from google.adk.agents import LlmAgent, BaseAgent

# Define individual agents

greeter = LlmAgent(name="Greeter", model="gemini-2.0-flash")

task\_doer = BaseAgent(name="TaskExecutor") # Custom non-LLM agent

# Create parent agent and assign children via sub\_agents

coordinator = LlmAgent(

name="Coordinator",

model="gemini-2.0-flash",

description="I coordinate greetings and tasks.",

sub\_agents=[ # Assign sub\_agents here

greeter,

task\_doer

]

)

# Framework automatically sets:

# assert greeter.parent\_agent == coordinator

# assert task\_doer.parent\_agent == coordinator

### 2.2. Workflow Agents as Orchestrators[¶](https://google.github.io/adk-docs/agents/multi-agents/#22-workflow-agents-as-orchestrators)

ADK includes specialized agents derived from BaseAgent that don't perform tasks themselves but orchestrate the execution flow of their sub\_agents.

* [**SequentialAgent**](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/)**:** Executes its sub\_agents one after another in the order they are listed.
  + **Context:** Passes the *same* [InvocationContext](https://google.github.io/adk-docs/runtime/) sequentially, allowing agents to easily pass results via shared state.

# Conceptual Example: Sequential Pipeline

from google.adk.agents import SequentialAgent, LlmAgent

step1 = LlmAgent(name="Step1\_Fetch", output\_key="data") # Saves output to state['data']

step2 = LlmAgent(name="Step2\_Process", instruction="Process data from state key 'data'.")

pipeline = SequentialAgent(name="MyPipeline", sub\_agents=[step1, step2])

# When pipeline runs, Step2 can access the state['data'] set by Step1.

* [**ParallelAgent**](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/)**:** Executes its sub\_agents in parallel. Events from sub-agents may be interleaved.
  + **Context:** Modifies the InvocationContext.branch for each child agent (e.g., ParentBranch.ChildName), providing a distinct contextual path which can be useful for isolating history in some memory implementations.
  + **State:** Despite different branches, all parallel children access the *same shared* session.state, enabling them to read initial state and write results (use distinct keys to avoid race conditions).

# Conceptual Example: Parallel Execution

from google.adk.agents import ParallelAgent, LlmAgent

fetch\_weather = LlmAgent(name="WeatherFetcher", output\_key="weather")

fetch\_news = LlmAgent(name="NewsFetcher", output\_key="news")

gatherer = ParallelAgent(name="InfoGatherer", sub\_agents=[fetch\_weather, fetch\_news])

# When gatherer runs, WeatherFetcher and NewsFetcher run concurrently.

# A subsequent agent could read state['weather'] and state['news'].

* [**LoopAgent**](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/)**:** Executes its sub\_agents sequentially in a loop.
  + **Termination:** The loop stops if the optional max\_iterations is reached, or if any sub-agent yields an [Event](https://google.github.io/adk-docs/events/) with actions.escalate=True.
  + **Context & State:** Passes the *same* InvocationContext in each iteration, allowing state changes (e.g., counters, flags) to persist across loops.

# Conceptual Example: Loop with Condition

from google.adk.agents import LoopAgent, LlmAgent, BaseAgent

from google.adk.events import Event, EventActions

from google.adk.agents.invocation\_context import InvocationContext

from typing import AsyncGenerator

class CheckCondition(BaseAgent): # Custom agent to check state

async def \_run\_async\_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:

status = ctx.session.state.get("status", "pending")

is\_done = (status == "completed")

yield Event(author=self.name, actions=EventActions(escalate=is\_done)) # Escalate if done

process\_step = LlmAgent(name="ProcessingStep") # Agent that might update state['status']

poller = LoopAgent(

name="StatusPoller",

max\_iterations=10,

sub\_agents=[process\_step, CheckCondition(name="Checker")]

)

# When poller runs, it executes process\_step then Checker repeatedly

# until Checker escalates (state['status'] == 'completed') or 10 iterations pass.

### 2.3. Interaction & Communication Mechanisms[¶](https://google.github.io/adk-docs/agents/multi-agents/#23-interaction-communication-mechanisms)

Agents within a system often need to exchange data or trigger actions in one another. ADK facilitates this through:

#### **a) Shared Session State (session.state)**[**¶**](https://google.github.io/adk-docs/agents/multi-agents/#a-shared-session-state-sessionstate)

The most fundamental way for agents operating within the same invocation (and thus sharing the same [Session](https://google.github.io/adk-docs/sessions/session/) object via the InvocationContext) to communicate passively.

* **Mechanism:** One agent (or its tool/callback) writes a value (context.state['data\_key'] = processed\_data), and a subsequent agent reads it (data = context.state.get('data\_key')). State changes are tracked via [CallbackContext](https://google.github.io/adk-docs/callbacks/).
* **Convenience:** The output\_key property on [LlmAgent](https://google.github.io/adk-docs/agents/llm-agents/) automatically saves the agent's final response text (or structured output) to the specified state key.
* **Nature:** Asynchronous, passive communication. Ideal for pipelines orchestrated by SequentialAgent or passing data across LoopAgent iterations.
* **See Also:** [State Management](https://google.github.io/adk-docs/sessions/state/)

# Conceptual Example: Using output\_key and reading state

from google.adk.agents import LlmAgent, SequentialAgent

agent\_A = LlmAgent(name="AgentA", instruction="Find the capital of France.", output\_key="capital\_city")

agent\_B = LlmAgent(name="AgentB", instruction="Tell me about the city stored in state key 'capital\_city'.")

pipeline = SequentialAgent(name="CityInfo", sub\_agents=[agent\_A, agent\_B])

# AgentA runs, saves "Paris" to state['capital\_city'].

# AgentB runs, its instruction processor reads state['capital\_city'] to get "Paris".

#### **b) LLM-Driven Delegation (Agent Transfer)**[**¶**](https://google.github.io/adk-docs/agents/multi-agents/#b-llm-driven-delegation-agent-transfer)

Leverages an [LlmAgent](https://google.github.io/adk-docs/agents/llm-agents/)'s understanding to dynamically route tasks to other suitable agents within the hierarchy.

* **Mechanism:** The agent's LLM generates a specific function call: transfer\_to\_agent(agent\_name='target\_agent\_name').
* **Handling:** The AutoFlow, used by default when sub-agents are present or transfer isn't disallowed, intercepts this call. It identifies the target agent using root\_agent.find\_agent() and updates the InvocationContext to switch execution focus.
* **Requires:** The calling LlmAgent needs clear instructions on when to transfer, and potential target agents need distinct descriptions for the LLM to make informed decisions. Transfer scope (parent, sub-agent, siblings) can be configured on the LlmAgent.
* **Nature:** Dynamic, flexible routing based on LLM interpretation.

# Conceptual Setup: LLM Transfer

from google.adk.agents import LlmAgent

booking\_agent = LlmAgent(name="Booker", description="Handles flight and hotel bookings.")

info\_agent = LlmAgent(name="Info", description="Provides general information and answers questions.")

coordinator = LlmAgent(

name="Coordinator",

model="gemini-2.0-flash",

instruction="You are an assistant. Delegate booking tasks to Booker and info requests to Info.",

description="Main coordinator.",

# AutoFlow is typically used implicitly here

sub\_agents=[booking\_agent, info\_agent]

)

# If coordinator receives "Book a flight", its LLM should generate:

# FunctionCall(name='transfer\_to\_agent', args={'agent\_name': 'Booker'})

# ADK framework then routes execution to booking\_agent.

#### **c) Explicit Invocation (AgentTool)**[**¶**](https://google.github.io/adk-docs/agents/multi-agents/#c-explicit-invocation-agenttool)

Allows an [LlmAgent](https://google.github.io/adk-docs/agents/llm-agents/) to treat another BaseAgent instance as a callable function or [Tool](https://google.github.io/adk-docs/tools/).

* **Mechanism:** Wrap the target agent instance in AgentTool and include it in the parent LlmAgent's tools list. AgentTool generates a corresponding function declaration for the LLM.
* **Handling:** When the parent LLM generates a function call targeting the AgentTool, the framework executes AgentTool.run\_async. This method runs the target agent, captures its final response, forwards any state/artifact changes back to the parent's context, and returns the response as the tool's result.
* **Nature:** Synchronous (within the parent's flow), explicit, controlled invocation like any other tool.
* **(Note:** AgentTool needs to be imported and used explicitly).

# Conceptual Setup: Agent as a Tool

from google.adk.agents import LlmAgent, BaseAgent

from google.adk.tools import agent\_tool

from pydantic import BaseModel

# Define a target agent (could be LlmAgent or custom BaseAgent)

class ImageGeneratorAgent(BaseAgent): # Example custom agent

name: str = "ImageGen"

description: str = "Generates an image based on a prompt."

# ... internal logic ...

async def \_run\_async\_impl(self, ctx): # Simplified run logic

prompt = ctx.session.state.get("image\_prompt", "default prompt")

# ... generate image bytes ...

image\_bytes = b"..."

yield Event(author=self.name, content=types.Content(parts=[types.Part.from\_bytes(image\_bytes, "image/png")]))

image\_agent = ImageGeneratorAgent()

image\_tool = agent\_tool.AgentTool(agent=image\_agent) # Wrap the agent

# Parent agent uses the AgentTool

artist\_agent = LlmAgent(

name="Artist",

model="gemini-2.0-flash",

instruction="Create a prompt and use the ImageGen tool to generate the image.",

tools=[image\_tool] # Include the AgentTool

)

# Artist LLM generates a prompt, then calls:

# FunctionCall(name='ImageGen', args={'image\_prompt': 'a cat wearing a hat'})

# Framework calls image\_tool.run\_async(...), which runs ImageGeneratorAgent.

# The resulting image Part is returned to the Artist agent as the tool result.

These primitives provide the flexibility to design multi-agent interactions ranging from tightly coupled sequential workflows to dynamic, LLM-driven delegation networks.

## 3. Common Multi-Agent Patterns using ADK Primitives[¶](https://google.github.io/adk-docs/agents/multi-agents/#3-common-multi-agent-patterns-using-adk-primitives)

By combining ADK's composition primitives, you can implement various established patterns for multi-agent collaboration.

### Coordinator/Dispatcher Pattern[¶](https://google.github.io/adk-docs/agents/multi-agents/#coordinatordispatcher-pattern)

* **Structure:** A central [LlmAgent](https://google.github.io/adk-docs/agents/llm-agents/) (Coordinator) manages several specialized sub\_agents.
* **Goal:** Route incoming requests to the appropriate specialist agent.
* **ADK Primitives Used:**
  + **Hierarchy:** Coordinator has specialists listed in sub\_agents.
  + **Interaction:** Primarily uses **LLM-Driven Delegation** (requires clear descriptions on sub-agents and appropriate instruction on Coordinator) or **Explicit Invocation (AgentTool)** (Coordinator includes AgentTool-wrapped specialists in its tools).

# Conceptual Code: Coordinator using LLM Transfer

from google.adk.agents import LlmAgent

billing\_agent = LlmAgent(name="Billing", description="Handles billing inquiries.")

support\_agent = LlmAgent(name="Support", description="Handles technical support requests.")

coordinator = LlmAgent(

name="HelpDeskCoordinator",

model="gemini-2.0-flash",

instruction="Route user requests: Use Billing agent for payment issues, Support agent for technical problems.",

description="Main help desk router.",

# allow\_transfer=True is often implicit with sub\_agents in AutoFlow

sub\_agents=[billing\_agent, support\_agent]

)

# User asks "My payment failed" -> Coordinator's LLM should call transfer\_to\_agent(agent\_name='Billing')

# User asks "I can't log in" -> Coordinator's LLM should call transfer\_to\_agent(agent\_name='Support')

### Sequential Pipeline Pattern[¶](https://google.github.io/adk-docs/agents/multi-agents/#sequential-pipeline-pattern)

* **Structure:** A [SequentialAgent](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/) contains sub\_agents executed in a fixed order.
* **Goal:** Implement a multi-step process where the output of one step feeds into the next.
* **ADK Primitives Used:**
  + **Workflow:** SequentialAgent defines the order.
  + **Communication:** Primarily uses **Shared Session State**. Earlier agents write results (often via output\_key), later agents read those results from context.state.

# Conceptual Code: Sequential Data Pipeline

from google.adk.agents import SequentialAgent, LlmAgent

validator = LlmAgent(name="ValidateInput", instruction="Validate the input.", output\_key="validation\_status")

processor = LlmAgent(name="ProcessData", instruction="Process data if state key 'validation\_status' is 'valid'.", output\_key="result")

reporter = LlmAgent(name="ReportResult", instruction="Report the result from state key 'result'.")

data\_pipeline = SequentialAgent(

name="DataPipeline",

sub\_agents=[validator, processor, reporter]

)

# validator runs -> saves to state['validation\_status']

# processor runs -> reads state['validation\_status'], saves to state['result']

# reporter runs -> reads state['result']

### Parallel Fan-Out/Gather Pattern[¶](https://google.github.io/adk-docs/agents/multi-agents/#parallel-fan-outgather-pattern)

* **Structure:** A [ParallelAgent](https://google.github.io/adk-docs/agents/workflow-agents/parallel-agents/) runs multiple sub\_agents concurrently, often followed by a later agent (in a SequentialAgent) that aggregates results.
* **Goal:** Execute independent tasks simultaneously to reduce latency, then combine their outputs.
* **ADK Primitives Used:**
  + **Workflow:** ParallelAgent for concurrent execution (Fan-Out). Often nested within a SequentialAgent to handle the subsequent aggregation step (Gather).
  + **Communication:** Sub-agents write results to distinct keys in **Shared Session State**. The subsequent "Gather" agent reads multiple state keys.

# Conceptual Code: Parallel Information Gathering

from google.adk.agents import SequentialAgent, ParallelAgent, LlmAgent

fetch\_api1 = LlmAgent(name="API1Fetcher", instruction="Fetch data from API 1.", output\_key="api1\_data")

fetch\_api2 = LlmAgent(name="API2Fetcher", instruction="Fetch data from API 2.", output\_key="api2\_data")

gather\_concurrently = ParallelAgent(

name="ConcurrentFetch",

sub\_agents=[fetch\_api1, fetch\_api2]

)

synthesizer = LlmAgent(

name="Synthesizer",

instruction="Combine results from state keys 'api1\_data' and 'api2\_data'."

)

overall\_workflow = SequentialAgent(

name="FetchAndSynthesize",

sub\_agents=[gather\_concurrently, synthesizer] # Run parallel fetch, then synthesize

)

# fetch\_api1 and fetch\_api2 run concurrently, saving to state.

# synthesizer runs afterwards, reading state['api1\_data'] and state['api2\_data'].

### Hierarchical Task Decomposition[¶](https://google.github.io/adk-docs/agents/multi-agents/#hierarchical-task-decomposition)

* **Structure:** A multi-level tree of agents where higher-level agents break down complex goals and delegate sub-tasks to lower-level agents.
* **Goal:** Solve complex problems by recursively breaking them down into simpler, executable steps.
* **ADK Primitives Used:**
  + **Hierarchy:** Multi-level parent\_agent/sub\_agents structure.
  + **Interaction:** Primarily **LLM-Driven Delegation** or **Explicit Invocation (AgentTool)** used by parent agents to assign tasks to children. Results are returned up the hierarchy (via tool responses or state).

# Conceptual Code: Hierarchical Research Task

from google.adk.agents import LlmAgent

from google.adk.tools import agent\_tool

# Low-level tool-like agents

web\_searcher = LlmAgent(name="WebSearch", description="Performs web searches for facts.")

summarizer = LlmAgent(name="Summarizer", description="Summarizes text.")

# Mid-level agent combining tools

research\_assistant = LlmAgent(

name="ResearchAssistant",

model="gemini-2.0-flash",

description="Finds and summarizes information on a topic.",

tools=[agent\_tool.AgentTool(agent=web\_searcher), agent\_tool.AgentTool(agent=summarizer)]

)

# High-level agent delegating research

report\_writer = LlmAgent(

name="ReportWriter",

model="gemini-2.0-flash",

instruction="Write a report on topic X. Use the ResearchAssistant to gather information.",

tools=[agent\_tool.AgentTool(agent=research\_assistant)]

# Alternatively, could use LLM Transfer if research\_assistant is a sub\_agent

)

# User interacts with ReportWriter.

# ReportWriter calls ResearchAssistant tool.

# ResearchAssistant calls WebSearch and Summarizer tools.

# Results flow back up.

### Review/Critique Pattern (Generator-Critic)[¶](https://google.github.io/adk-docs/agents/multi-agents/#reviewcritique-pattern-generator-critic)

* **Structure:** Typically involves two agents within a [SequentialAgent](https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/): a Generator and a Critic/Reviewer.
* **Goal:** Improve the quality or validity of generated output by having a dedicated agent review it.
* **ADK Primitives Used:**
  + **Workflow:** SequentialAgent ensures generation happens before review.
  + **Communication:** **Shared Session State** (Generator uses output\_key to save output; Reviewer reads that state key). The Reviewer might save its feedback to another state key for subsequent steps.

# Conceptual Code: Generator-Critic

from google.adk.agents import SequentialAgent, LlmAgent

generator = LlmAgent(

name="DraftWriter",

instruction="Write a short paragraph about subject X.",

output\_key="draft\_text"

)

reviewer = LlmAgent(

name="FactChecker",

instruction="Review the text in state key 'draft\_text' for factual accuracy. Output 'valid' or 'invalid' with reasons.",

output\_key="review\_status"

)

# Optional: Further steps based on review\_status

review\_pipeline = SequentialAgent(

name="WriteAndReview",

sub\_agents=[generator, reviewer]

)

# generator runs -> saves draft to state['draft\_text']

# reviewer runs -> reads state['draft\_text'], saves status to state['review\_status']

### Iterative Refinement Pattern[¶](https://google.github.io/adk-docs/agents/multi-agents/#iterative-refinement-pattern)

* **Structure:** Uses a [LoopAgent](https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/) containing one or more agents that work on a task over multiple iterations.
* **Goal:** Progressively improve a result (e.g., code, text, plan) stored in the session state until a quality threshold is met or a maximum number of iterations is reached.
* **ADK Primitives Used:**
  + **Workflow:** LoopAgent manages the repetition.
  + **Communication:** **Shared Session State** is essential for agents to read the previous iteration's output and save the refined version.
  + **Termination:** The loop typically ends based on max\_iterations or a dedicated checking agent setting actions.escalate=True when the result is satisfactory.

# Conceptual Code: Iterative Code Refinement

from google.adk.agents import LoopAgent, LlmAgent, BaseAgent

from google.adk.events import Event, EventActions

from google.adk.agents.invocation\_context import InvocationContext

from typing import AsyncGenerator

# Agent to generate/refine code based on state['current\_code'] and state['requirements']

code\_refiner = LlmAgent(

name="CodeRefiner",

instruction="Read state['current\_code'] (if exists) and state['requirements']. Generate/refine Python code to meet requirements. Save to state['current\_code'].",

output\_key="current\_code" # Overwrites previous code in state

)

# Agent to check if the code meets quality standards

quality\_checker = LlmAgent(

name="QualityChecker",

instruction="Evaluate the code in state['current\_code'] against state['requirements']. Output 'pass' or 'fail'.",

output\_key="quality\_status"

)

# Custom agent to check the status and escalate if 'pass'

class CheckStatusAndEscalate(BaseAgent):

async def \_run\_async\_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:

status = ctx.session.state.get("quality\_status", "fail")

should\_stop = (status == "pass")

yield Event(author=self.name, actions=EventActions(escalate=should\_stop))

refinement\_loop = LoopAgent(

name="CodeRefinementLoop",

max\_iterations=5,

sub\_agents=[code\_refiner, quality\_checker, CheckStatusAndEscalate(name="StopChecker")]

)

# Loop runs: Refiner -> Checker -> StopChecker

# State['current\_code'] is updated each iteration.

# Loop stops if QualityChecker outputs 'pass' (leading to StopChecker escalating) or after 5 iterations.

### Human-in-the-Loop Pattern[¶](https://google.github.io/adk-docs/agents/multi-agents/#human-in-the-loop-pattern)

* **Structure:** Integrates human intervention points within an agent workflow.
* **Goal:** Allow for human oversight, approval, correction, or tasks that AI cannot perform.
* **ADK Primitives Used (Conceptual):**
  + **Interaction:** Can be implemented using a custom **Tool** that pauses execution and sends a request to an external system (e.g., a UI, ticketing system) waiting for human input. The tool then returns the human's response to the agent.
  + **Workflow:** Could use **LLM-Driven Delegation** (transfer\_to\_agent) targeting a conceptual "Human Agent" that triggers the external workflow, or use the custom tool within an LlmAgent.
  + **State/Callbacks:** State can hold task details for the human; callbacks can manage the interaction flow.
  + **Note:** ADK doesn't have a built-in "Human Agent" type, so this requires custom integration.

# Conceptual Code: Using a Tool for Human Approval

from google.adk.agents import LlmAgent, SequentialAgent

from google.adk.tools import FunctionTool

# --- Assume external\_approval\_tool exists ---

# This tool would:

# 1. Take details (e.g., request\_id, amount, reason).

# 2. Send these details to a human review system (e.g., via API).

# 3. Poll or wait for the human response (approved/rejected).

# 4. Return the human's decision.

# async def external\_approval\_tool(amount: float, reason: str) -> str: ...

approval\_tool = FunctionTool(func=external\_approval\_tool)

# Agent that prepares the request

prepare\_request = LlmAgent(

name="PrepareApproval",

instruction="Prepare the approval request details based on user input. Store amount and reason in state.",

# ... likely sets state['approval\_amount'] and state['approval\_reason'] ...

)

# Agent that calls the human approval tool

request\_approval = LlmAgent(

name="RequestHumanApproval",

instruction="Use the external\_approval\_tool with amount from state['approval\_amount'] and reason from state['approval\_reason'].",

tools=[approval\_tool],

output\_key="human\_decision"

)

# Agent that proceeds based on human decision

process\_decision = LlmAgent(

name="ProcessDecision",

instruction="Check state key 'human\_decision'. If 'approved', proceed. If 'rejected', inform user."

)

approval\_workflow = SequentialAgent(

name="HumanApprovalWorkflow",

sub\_agents=[prepare\_request, request\_approval, process\_decision]

)

These patterns provide starting points for structuring your multi-agent systems. You can mix and match them as needed to create the most effective architecture for your specific application.

# Using Different Models with ADK[¶](https://google.github.io/adk-docs/agents/models/#using-different-models-with-adk)

The Agent Development Kit (ADK) is designed for flexibility, allowing you to integrate various Large Language Models (LLMs) into your agents. While the setup for Google Gemini models is covered in the [Setup Foundation Models](https://google.github.io/adk-docs/get-started/installation/) guide, this page details how to leverage Gemini effectively and integrate other popular models, including those hosted externally or running locally.

ADK primarily uses two mechanisms for model integration:

1. **Direct String / Registry:** For models tightly integrated with Google Cloud (like Gemini models accessed via Google AI Studio or Vertex AI) or models hosted on Vertex AI endpoints. You typically provide the model name or endpoint resource string directly to the LlmAgent. ADK's internal registry resolves this string to the appropriate backend client, often utilizing the google-genai library.
2. **Wrapper Classes:** For broader compatibility, especially with models outside the Google ecosystem or those requiring specific client configurations (like models accessed via LiteLLM). You instantiate a specific wrapper class (e.g., LiteLlm) and pass this object as the model parameter to your LlmAgent.

The following sections guide you through using these methods based on your needs.

## Using Google Gemini Models[¶](https://google.github.io/adk-docs/agents/models/#using-google-gemini-models)

This is the most direct way to use Google's flagship models within ADK.

**Integration Method:** Pass the model's identifier string directly to the model parameter of LlmAgent (or its alias, Agent).

**Backend Options & Setup:**

The google-genai library, used internally by ADK for Gemini, can connect through either Google AI Studio or Vertex AI.

**Model support for voice/video streaming**

In order to use voice/video streaming in ADK, you will need to use Gemini models that support the Live API. You can find the **model ID(s)** that support the Gemini Live API in the documentation:

* [Google AI Studio: Gemini Live API](https://ai.google.dev/gemini-api/docs/models#live-api)
* [Vertex AI: Gemini Live API](https://cloud.google.com/vertex-ai/generative-ai/docs/live-api)

### Google AI Studio[¶](https://google.github.io/adk-docs/agents/models/#google-ai-studio)

* **Use Case:** Google AI Studio is the easiest way to get started with Gemini. All you need is the [API key](https://aistudio.google.com/app/apikey). Best for rapid prototyping and development.
* **Setup:** Typically requires an API key set as an environment variable:

export GOOGLE\_API\_KEY="YOUR\_GOOGLE\_API\_KEY"

export GOOGLE\_GENAI\_USE\_VERTEXAI=FALSE

* **Models:** Find all available models on the [Google AI for Developers site](https://ai.google.dev/gemini-api/docs/models).

### Vertex AI[¶](https://google.github.io/adk-docs/agents/models/#vertex-ai)

* **Use Case:** Recommended for production applications, leveraging Google Cloud infrastructure. Gemini on Vertex AI supports enterprise-grade features, security, and compliance controls.
* **Setup:**
  + Authenticate using Application Default Credentials (ADC):

gcloud auth application-default login

* + Set your Google Cloud project and location:

export GOOGLE\_CLOUD\_PROJECT="YOUR\_PROJECT\_ID"

export GOOGLE\_CLOUD\_LOCATION="YOUR\_VERTEX\_AI\_LOCATION" # e.g., us-central1

* + Explicitly tell the library to use Vertex AI:

export GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE

* **Models:** Find available model IDs in the [Vertex AI documentation](https://cloud.google.com/vertex-ai/generative-ai/docs/learn/models).

**Example:**

from google.adk.agents import LlmAgent

# --- Example using a stable Gemini Flash model ---

agent\_gemini\_flash = LlmAgent(

# Use the latest stable Flash model identifier

model="gemini-2.0-flash",

name="gemini\_flash\_agent",

instruction="You are a fast and helpful Gemini assistant.",

# ... other agent parameters

)

# --- Example using a powerful Gemini Pro model ---

# Note: Always check the official Gemini documentation for the latest model names,

# including specific preview versions if needed. Preview models might have

# different availability or quota limitations.

agent\_gemini\_pro = LlmAgent(

# Use the latest generally available Pro model identifier

model="gemini-2.5-pro-preview-03-25",

name="gemini\_pro\_agent",

instruction="You are a powerful and knowledgeable Gemini assistant.",

# ... other agent parameters

)

## Using Cloud & Proprietary Models via LiteLLM[¶](https://google.github.io/adk-docs/agents/models/#using-cloud-proprietary-models-via-litellm)

To access a vast range of LLMs from providers like OpenAI, Anthropic (non-Vertex AI), Cohere, and many others, ADK offers integration through the LiteLLM library.

**Integration Method:** Instantiate the LiteLlm wrapper class and pass it to the model parameter of LlmAgent.

**LiteLLM Overview:** [LiteLLM](https://docs.litellm.ai/) acts as a translation layer, providing a standardized, OpenAI-compatible interface to over 100+ LLMs.

**Setup:**

1. **Install LiteLLM:**

pip install litellm

1. **Set Provider API Keys:** Configure API keys as environment variables for the specific providers you intend to use.
   * *Example for OpenAI:*

export OPENAI\_API\_KEY="YOUR\_OPENAI\_API\_KEY"

* + *Example for Anthropic (non-Vertex AI):*

export ANTHROPIC\_API\_KEY="YOUR\_ANTHROPIC\_API\_KEY"

* + *Consult the* [*LiteLLM Providers Documentation*](https://docs.litellm.ai/docs/providers) *for the correct environment variable names for other providers.***Example:**

from google.adk.agents import LlmAgent

from google.adk.models.lite\_llm import LiteLlm

# --- Example Agent using OpenAI's GPT-4o ---

# (Requires OPENAI\_API\_KEY)

agent\_openai = LlmAgent(

model=LiteLlm(model="openai/gpt-4o"), # LiteLLM model string format

name="openai\_agent",

instruction="You are a helpful assistant powered by GPT-4o.",

# ... other agent parameters

)

# --- Example Agent using Anthropic's Claude Haiku (non-Vertex) ---

# (Requires ANTHROPIC\_API\_KEY)

agent\_claude\_direct = LlmAgent(

model=LiteLlm(model="anthropic/claude-3-haiku-20240307"),

name="claude\_direct\_agent",

instruction="You are an assistant powered by Claude Haiku.",

# ... other agent parameters

)

## Using Open & Local Models via LiteLLM[¶](https://google.github.io/adk-docs/agents/models/#using-open-local-models-via-litellm)

For maximum control, cost savings, privacy, or offline use cases, you can run open-source models locally or self-host them and integrate them using LiteLLM.

**Integration Method:** Instantiate the LiteLlm wrapper class, configured to point to your local model server.

### Ollama Integration[¶](https://google.github.io/adk-docs/agents/models/#ollama-integration)

[Ollama](https://ollama.com/) allows you to easily run open-source models locally.

#### **Model choice**[**¶**](https://google.github.io/adk-docs/agents/models/#model-choice)

If your agent is relying on tools, please make sure that you select a model with tool support from [Ollama website](https://ollama.com/search?c=tools).

For reliable results, we recommend using a decent-sized model with tool support.

The tool support for the model can be checked with the following command:

ollama show mistral-small3.1

Model

architecture mistral3

parameters 24.0B

context length 131072

embedding length 5120

quantization Q4\_K\_M

Capabilities

completion

vision

tools

You are supposed to see tools listed under capabilities.

You can also look at the template the model is using and tweak it based on your needs.

ollama show --modelfile llama3.2 > model\_file\_to\_modify

For instance, the default template for the above model inherently suggests that the model shall call a function all the time. This may result in an infinite loop of function calls.

Given the following functions, please respond with a JSON for a function call

with its proper arguments that best answers the given prompt.

Respond in the format {"name": function name, "parameters": dictionary of

argument name and its value}. Do not use variables.

You can swap such prompts with a more descriptive one to prevent infinite tool call loops.

For instance:

Review the user's prompt and the available functions listed below.

First, determine if calling one of these functions is the most appropriate way to respond. A function call is likely needed if the prompt asks for a specific action, requires external data lookup, or involves calculations handled by the functions. If the prompt is a general question or can be answered directly, a function call is likely NOT needed.

If you determine a function call IS required: Respond ONLY with a JSON object in the format {"name": "function\_name", "parameters": {"argument\_name": "value"}}. Ensure parameter values are concrete, not variables.

If you determine a function call IS NOT required: Respond directly to the user's prompt in plain text, providing the answer or information requested. Do not output any JSON.

Then you can create a new model with the following command:

ollama create llama3.2-modified -f model\_file\_to\_modify

#### **Using ollama\_chat provider**[**¶**](https://google.github.io/adk-docs/agents/models/#using-ollama_chat-provider)

Our LiteLLM wrapper can be used to create agents with Ollama models.

root\_agent = Agent(

model=LiteLlm(model="ollama\_chat/mistral-small3.1"),

name="dice\_agent",

description=(

"hello world agent that can roll a dice of 8 sides and check prime"

" numbers."

),

instruction="""

You roll dice and answer questions about the outcome of the dice rolls.

""",

tools=[

roll\_die,

check\_prime,

],

)

**It is important to set the provider ollama\_chat instead of ollama. Using ollama will result in unexpected behaviors such as infinite tool call loops and ignoring previous context.**

While api\_base can be provided inside LiteLLM for generation, LiteLLM library is calling other APIs relying on the env variable instead as of v1.65.5 after completion. So at this time, we recommend setting the env variable OLLAMA\_API\_BASE to point to the ollama server.

export OLLAMA\_API\_BASE="http://localhost:11434"

adk web

#### **Using openai provider**[**¶**](https://google.github.io/adk-docs/agents/models/#using-openai-provider)

Alternatively, openai can be used as the provider name. But this will also require setting the OPENAI\_API\_BASE=http://localhost:11434/v1 and OPENAI\_API\_KEY=anything env variables instead of OLLAMA\_API\_BASE. **Please note that api base now has /v1 at the end.**

root\_agent = Agent(

model=LiteLlm(model="openai/mistral-small3.1"),

name="dice\_agent",

description=(

"hello world agent that can roll a dice of 8 sides and check prime"

" numbers."

),

instruction="""

You roll dice and answer questions about the outcome of the dice rolls.

""",

tools=[

roll\_die,

check\_prime,

],

)

export OPENAI\_API\_BASE=http://localhost:11434/v1

export OPENAI\_API\_KEY=anything

adk web

#### **Debugging**[**¶**](https://google.github.io/adk-docs/agents/models/#debugging)

You can see the request sent to the Ollama server by adding the following in your agent code just after imports.

import litellm

litellm.\_turn\_on\_debug()

Look for a line like the following:

Request Sent from LiteLLM:

curl -X POST \

http://localhost:11434/api/chat \

-d '{'model': 'mistral-small3.1', 'messages': [{'role': 'system', 'content': ...

### Self-Hosted Endpoint (e.g., vLLM)[¶](https://google.github.io/adk-docs/agents/models/#self-hosted-endpoint-eg-vllm)

Tools such as [vLLM](https://github.com/vllm-project/vllm) allow you to host models efficiently and often expose an OpenAI-compatible API endpoint.

**Setup:**

1. **Deploy Model:** Deploy your chosen model using vLLM (or a similar tool). Note the API base URL (e.g., https://your-vllm-endpoint.run.app/v1).
   * *Important for ADK Tools:* When deploying, ensure the serving tool supports and enables OpenAI-compatible tool/function calling. For vLLM, this might involve flags like --enable-auto-tool-choice and potentially a specific --tool-call-parser, depending on the model. Refer to the vLLM documentation on Tool Use.
2. **Authentication:** Determine how your endpoint handles authentication (e.g., API key, bearer token).  
   **Integration Example:**

import subprocess

from google.adk.agents import LlmAgent

from google.adk.models.lite\_llm import LiteLlm

# --- Example Agent using a model hosted on a vLLM endpoint ---

# Endpoint URL provided by your vLLM deployment

api\_base\_url = "https://your-vllm-endpoint.run.app/v1"

# Model name as recognized by \*your\* vLLM endpoint configuration

model\_name\_at\_endpoint = "hosted\_vllm/google/gemma-3-4b-it" # Example from vllm\_test.py

# Authentication (Example: using gcloud identity token for a Cloud Run deployment)

# Adapt this based on your endpoint's security

try:

gcloud\_token = subprocess.check\_output(

["gcloud", "auth", "print-identity-token", "-q"]

).decode().strip()

auth\_headers = {"Authorization": f"Bearer {gcloud\_token}"}

except Exception as e:

print(f"Warning: Could not get gcloud token - {e}. Endpoint might be unsecured or require different auth.")

auth\_headers = None # Or handle error appropriately

agent\_vllm = LlmAgent(

model=LiteLlm(

model=model\_name\_at\_endpoint,

api\_base=api\_base\_url,

# Pass authentication headers if needed

extra\_headers=auth\_headers

# Alternatively, if endpoint uses an API key:

# api\_key="YOUR\_ENDPOINT\_API\_KEY"

),

name="vllm\_agent",

instruction="You are a helpful assistant running on a self-hosted vLLM endpoint.",

# ... other agent parameters

)

## Using Hosted & Tuned Models on Vertex AI[¶](https://google.github.io/adk-docs/agents/models/#using-hosted-tuned-models-on-vertex-ai)

For enterprise-grade scalability, reliability, and integration with Google Cloud's MLOps ecosystem, you can use models deployed to Vertex AI Endpoints. This includes models from Model Garden or your own fine-tuned models.

**Integration Method:** Pass the full Vertex AI Endpoint resource string (projects/PROJECT\_ID/locations/LOCATION/endpoints/ENDPOINT\_ID) directly to the model parameter of LlmAgent.

**Vertex AI Setup (Consolidated):**

Ensure your environment is configured for Vertex AI:

1. **Authentication:** Use Application Default Credentials (ADC):

gcloud auth application-default login

1. **Environment Variables:** Set your project and location:

export GOOGLE\_CLOUD\_PROJECT="YOUR\_PROJECT\_ID"

export GOOGLE\_CLOUD\_LOCATION="YOUR\_VERTEX\_AI\_LOCATION" # e.g., us-central1

1. **Enable Vertex Backend:** Crucially, ensure the google-genai library targets Vertex AI:

export GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE

### Model Garden Deployments[¶](https://google.github.io/adk-docs/agents/models/#model-garden-deployments)

You can deploy various open and proprietary models from the [Vertex AI Model Garden](https://console.cloud.google.com/vertex-ai/model-garden) to an endpoint.

**Example:**

from google.adk.agents import LlmAgent

from google.genai import types # For config objects

# --- Example Agent using a Llama 3 model deployed from Model Garden ---

# Replace with your actual Vertex AI Endpoint resource name

llama3\_endpoint = "projects/YOUR\_PROJECT\_ID/locations/us-central1/endpoints/YOUR\_LLAMA3\_ENDPOINT\_ID"

agent\_llama3\_vertex = LlmAgent(

model=llama3\_endpoint,

name="llama3\_vertex\_agent",

instruction="You are a helpful assistant based on Llama 3, hosted on Vertex AI.",

generate\_content\_config=types.GenerateContentConfig(max\_output\_tokens=2048),

# ... other agent parameters

)

### Fine-tuned Model Endpoints[¶](https://google.github.io/adk-docs/agents/models/#fine-tuned-model-endpoints)

Deploying your fine-tuned models (whether based on Gemini or other architectures supported by Vertex AI) results in an endpoint that can be used directly.

**Example:**

from google.adk.agents import LlmAgent

# --- Example Agent using a fine-tuned Gemini model endpoint ---

# Replace with your fine-tuned model's endpoint resource name

finetuned\_gemini\_endpoint = "projects/YOUR\_PROJECT\_ID/locations/us-central1/endpoints/YOUR\_FINETUNED\_ENDPOINT\_ID"

agent\_finetuned\_gemini = LlmAgent(

model=finetuned\_gemini\_endpoint,

name="finetuned\_gemini\_agent",

instruction="You are a specialized assistant trained on specific data.",

# ... other agent parameters

)

### Third-Party Models on Vertex AI (e.g., Anthropic Claude)[¶](https://google.github.io/adk-docs/agents/models/#third-party-models-on-vertex-ai-eg-anthropic-claude)

Some providers, like Anthropic, make their models available directly through Vertex AI.

**Integration Method:** Uses the direct model string (e.g., "claude-3-sonnet@20240229"), *but requires manual registration* within ADK.

**Why Registration?** ADK's registry automatically recognizes gemini-\* strings and standard Vertex AI endpoint strings (projects/.../endpoints/...) and routes them via the google-genai library. For other model types used directly via Vertex AI (like Claude), you must explicitly tell the ADK registry which specific wrapper class (Claude in this case) knows how to handle that model identifier string with the Vertex AI backend.

**Setup:**

1. **Vertex AI Environment:** Ensure the consolidated Vertex AI setup (ADC, Env Vars, GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE) is complete.
2. **Install Provider Library:** Install the necessary client library configured for Vertex AI.

pip install "anthropic[vertex]"

1. **Register Model Class:** Add this code near the start of your application, *before* creating an agent using the Claude model string:

# Required for using Claude model strings directly via Vertex AI with LlmAgent

from google.adk.models.anthropic\_llm import Claude

from google.adk.models.registry import LLMRegistry

LLMRegistry.register(Claude)

1. **Example:**

from google.adk.agents import LlmAgent

from google.adk.models.anthropic\_llm import Claude # Import needed for registration

from google.adk.models.registry import LLMRegistry # Import needed for registration

from google.genai import types

# --- Register Claude class (do this once at startup) ---

LLMRegistry.register(Claude)

# --- Example Agent using Claude 3 Sonnet on Vertex AI ---

# Standard model name for Claude 3 Sonnet on Vertex AI

claude\_model\_vertexai = "claude-3-sonnet@20240229"

agent\_claude\_vertexai = LlmAgent(

model=claude\_model\_vertexai, # Pass the direct string after registration

name="claude\_vertexai\_agent",

instruction="You are an assistant powered by Claude 3 Sonnet on Vertex AI.",

generate\_content\_config=types.GenerateContentConfig(max\_output\_tokens=4096),

# ... other agent parameters

1. )

https://google.github.io/adk-docs/