

What is Agent Development Kit?

Agent Development Kit (ADK) is a flexible and modular framework for **developing and deploying AI agents**. While optimized for Gemini and the Google ecosystem, ADK is **model-agnostic, deployment-agnostic**, and is built for **compatibility with other frameworks**. ADK was designed to make agent development feel more like software development, to make it easier for developers to create, deploy, and orchestrate agentic architectures that range from simple tasks to complex workflows.

Get started:

[Python](#)

[Java](#)

```
pip install google-adk
```

[Quickstart](#) [Tutorials](#) [Sample Agents](#) [API Reference](#) [Contribute](#) 

[Learn more](#)

Watch "Introducing Agent Development Kit"!

- **Flexible Orchestration**

- Define workflows using workflow agents (Sequential, Parallel, Loop) for predictable pipelines, or leverage LLM-driven dynamic routing (LlmAgent transfer) for adaptive behavior.

[Learn about agents](#)

- **Multi-Agent Architecture**

- Build modular and scalable applications by composing multiple specialized agents in a hierarchy. Enable complex coordination and delegation.

[Explore multi-agent systems](#)

- **Rich Tool Ecosystem**

- Equip agents with diverse capabilities: use pre-built tools (Search, Code Exec), create custom functions, integrate 3rd-party libraries (LangChain, CrewAI), or even use other agents as tools.

[Browse tools](#)

- **Deployment Ready**

- Containerize and deploy your agents anywhere – run locally, scale with Vertex AI Agent Engine, or integrate into custom infrastructure using Cloud Run or Docker.

[Deploy agents](#)

- **Built-in Evaluation**

- Systematically assess agent performance by evaluating both the final response quality and the step-by-step execution trajectory against predefined test cases.

[Evaluate agents](#)

- **Building Safe and Secure Agents**

- Learn how to building powerful and trustworthy agents by implementing security and safety patterns and best practices into your agent's design.
Safety and Security

Get Started

Agent Development Kit (ADK) is designed to empower developers to build, manage, evaluate and deploy AI-powered agents. It provides a robust and flexible environment for creating both conversational and non-conversational agents, capable of handling complex tasks and workflows.

- **Installation**

- Install [google-adk](#) for Python or Java and get up and running in minutes.
[More information](#)
- **Quickstart**

- Create your first ADK agent with tools in minutes.
[More information](#)
- **Quickstart (streaming)**

- Create your first streaming ADK agent.
[More information](#)
- **Tutorial**

- Create your first ADK multi-agent.
[More information](#)
- **Discover sample agents**

- Discover sample agents for retail, travel, customer service, and more!
[Discover adk-samples](#)
- **About**

- Learn about the key components of building and deploying ADK agents.
[More information](#)

Installing ADK

[Python](#)

[Java](#)

Create & activate virtual environment

We recommend creating a virtual Python environment using [venv](#):

```
python -m venv .venv
```

Now, you can activate the virtual environment using the appropriate command for your operating system and environment:

```
# Mac / Linux
```

```
source .venv/bin/activate
```

```
# Windows CMD:
```

```
.venv\Scripts\activate.bat
```

```
# Windows PowerShell:
```

```
.venv\Scripts\Activate.ps1
```

Install ADK

```
pip install google-adk
```

(Optional) Verify your installation:

```
pip show google-adk
```

Next steps

- Try creating your first agent with the [Quickstart](#)

Quickstart

This quickstart guides you through installing the Agent Development Kit (ADK), setting up a basic agent with multiple tools, and running it locally either in the terminal or in the interactive, browser-based dev UI.

This quickstart assumes a local IDE (VS Code, PyCharm, IntelliJ IDEA, etc.) with Python 3.9+ or Java 17+ and terminal access. This method runs the application entirely on your machine and is recommended for internal development.

1. Set up Environment & Install ADK

[Python](#)

[Java](#)

Create & Activate Virtual Environment (Recommended):

```
# Create
```

```
python -m venv .venv
```

```
# Activate (each new terminal)
```

```
# macOS/Linux: source .venv/bin/activate
```

```
# Windows CMD: .venv\Scripts\activate.bat
```

```
# Windows PowerShell: .venv\Scripts\Activate.ps1
```

Install ADK:

```
pip install google-adk
```

2. Create Agent Project

Project structure

[Python](#)

[Java](#)

You will need to create the following project structure:

```
parent_folder/
```

```
    multi_tool_agent/
```

```
        __init__.py
```

```
        agent.py
```

```
    .env
```

Create the folder `multi_tool_agent`:

```
mkdir multi_tool_agent/
```

Note for Windows users

When using ADK on Windows for the next few steps, we recommend creating Python files using File Explorer or an IDE because the following commands (`mkdir`, `echo`) typically generate files with null bytes and/or incorrect encoding.

`__init__.py`

Now create an `__init__.py` file in the folder:

```
echo "from . import agent" > multi_tool_agent/__init__.py
```

Your `__init__.py` should now look like this:

```
multi_tool_agent/__init__.py  
from . import agent
```

`agent.py`

Create an `agent.py` file in the same folder:

```
touch multi_tool_agent/agent.py
```

Copy and paste the following code into `agent.py`:

```
multi_tool_agent/agent.py
```

```
import datetime

from zoneinfo import ZoneInfo

from google.adk.agents import Agent

def get_weather(city: str) -> dict:

    """Retrieves the current weather report for a specified city.

    Args:
        city (str): The name of the city for which to retrieve the weather
                    report.
    """
    # Implementation details (e.g., API calls, data processing) go here.
```

Returns:

```
dict: status and result or error msg.
```

```
"""
```

```
if city.lower() == "new york":
```

```
    return {
```

```
        "status": "success",
```

```
        "report": (
```

```
            "The weather in New York is sunny with a temperature of 25  
degrees"
```

```
            " Celsius (77 degrees Fahrenheit)."
```

```
        ),  
  
    }  
  
    else:  
  
        return {  
  
            "status": "error",  
  
            "error_message": f"Weather information for '{city}' is not  
available.",  
  
        }  
  
def get_current_time(city: str) -> dict:
```

```
    """Returns the current time in a specified city.
```

```
Args:
```

```
    city (str): The name of the city for which to retrieve the current  
time.
```

```
Returns:
```

```
    dict: status and result or error msg.
```

```
    """
```

```
if city.lower() == "new york":  
  
    tz_identifier = "America/New_York"  
  
else:  
  
    return {  
  
        "status": "error",  
  
        "error_message": (  
  
            f"Sorry, I don't have timezone information for {city}."  
  
        ),  
  
    }  

```

```
tz = ZoneInfo(tz_identifier)

now = datetime.datetime.now(tz)

report = (
    f'The current time in {city} is {now.strftime("%Y-%m-%d %H:%M:%S"
%Z%z")}'
)

return {"status": "success", "report": report}

root_agent = Agent(
```

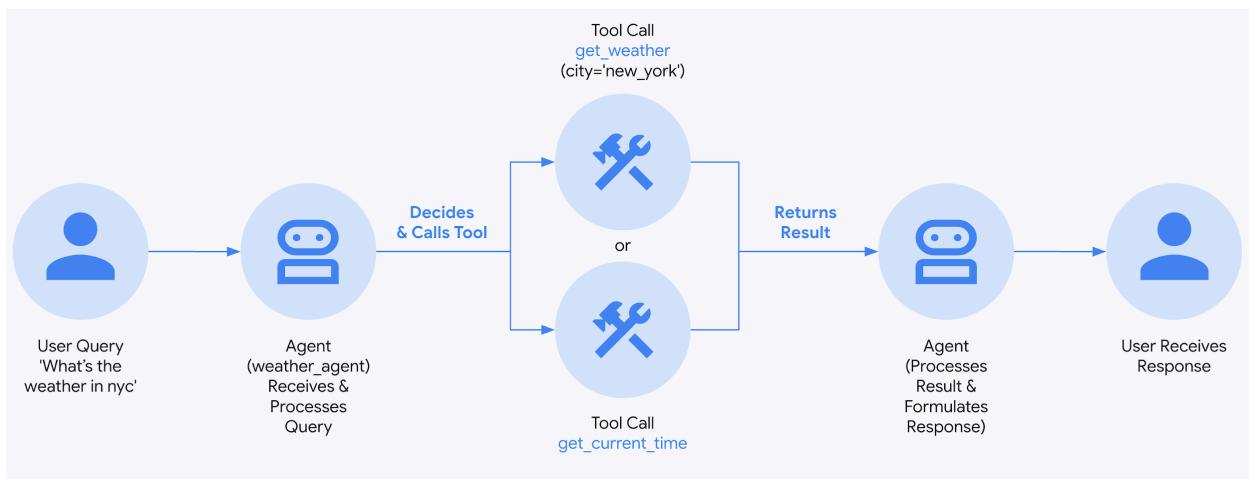
```
    name="weather_time_agent",  
  
    model="gemini-2.0-flash",  
  
    description=(  
        "Agent to answer questions about the time and weather in a city."  
    ),  
  
    instruction=(  
        "You are a helpful agent who can answer user questions about the time  
        and weather in a city."  
    ),  
  
    tools=[get_weather, get_current_time],  
  
)
```

.env

Create a `.env` file in the same folder:

```
touch multi_tool_agent/.env
```

More instructions about this file are described in the next section on [Set up the model](#).



3. Set up the model

Your agent's ability to understand user requests and generate responses is powered by a Large Language Model (LLM). Your agent needs to make secure calls to this external LLM service, which requires authentication credentials. Without valid authentication, the LLM service will deny the agent's requests, and the agent will be unable to function.

[Gemini - Google AI Studio](#)

Gemini - Google Cloud Vertex AI

1. Get an API key from [Google AI Studio](#).
2. When using Python, open the `.env` file located inside (`multi_tool_agent/`) and copy-paste the following code.
3. `multi_tool_agent/.env`

```
GOOGLE_GENAI_USE_VERTEXAI=FALSE
```

- 4.

```
GOOGLE_API_KEY=PASTE_YOUR_ACTUAL_API_KEY_HERE
```

- 5.
6. When using Java, define environment variables:
7. `terminal`

```
export GOOGLE_GENAI_USE_VERTEXAI=FALSE
```

- 8.

```
export GOOGLE_API_KEY=PASTE_YOUR_ACTUAL_API_KEY_HERE
```

- 9.
10. Replace `PASTE_YOUR_ACTUAL_API_KEY_HERE` with your actual API KEY.

4. Run Your Agent

Python

Java

Using the terminal, navigate to the parent directory of your agent project (e.g. using `cd ..`):

```
parent_folder/      <-- navigate to this directory
```

```
multi_tool_agent/
```

```
--init__.py
```

```
agent.py
```

```
.env
```

There are multiple ways to interact with your agent:

[Dev UI \(adk web\)](#)

[Terminal \(adk run\)](#)

[API Server \(adk api_server\)](#)

Run the following command to launch the **dev UI**.

```
adk web
```

Note for Windows users

When hitting the `_make_subprocess_transport` `NotImplementedError`, consider using `adk web --no-reload` instead.

Step 1: Open the URL provided (usually `http://localhost:8000` or `http://127.0.0.1:8000`) directly in your browser.

Step 2. In the top-left corner of the UI, you can select your agent in the dropdown. Select "multi_tool_agent".

Troubleshooting

If you do not see "multi_tool_agent" in the dropdown menu, make sure you are running `adk web` in the **parent folder** of your agent folder (i.e. the parent folder of `multi_tool_agent`).

Step 3. Now you can chat with your agent using the textbox:

Step 4. By using the `Events` tab at the left, you can inspect individual function calls, responses and model responses by clicking on the actions:

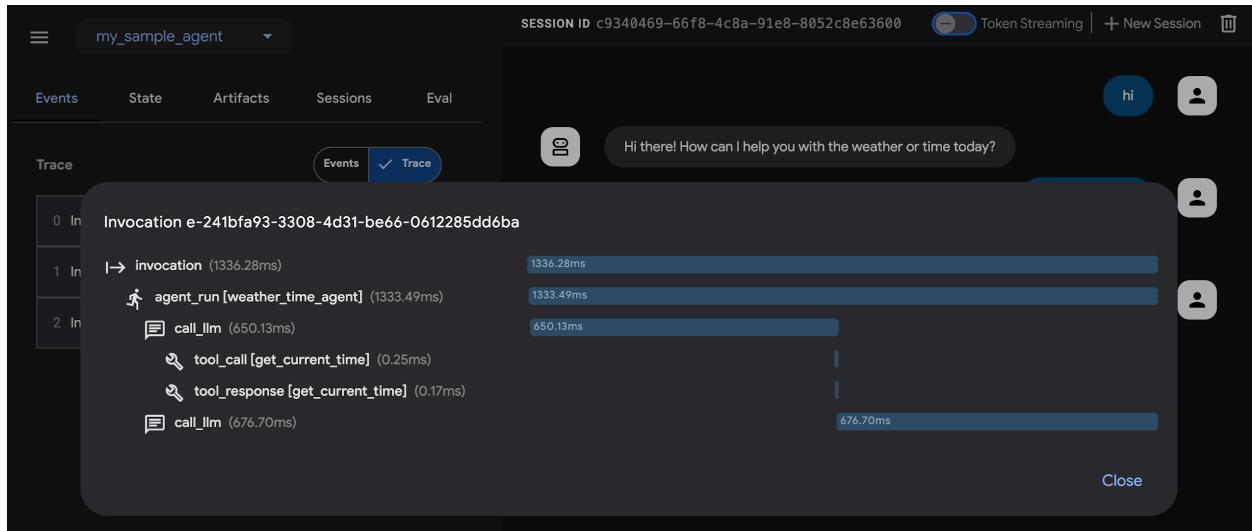
```

Event 3 of 5 < > X
SESSION ID 1e3d41f4-9442-41af-8dc0-5c2e705a13ff
Token Streaming | + New Session | [session icon]

Event Request Response
weather_time_agent get_weather
weather_time_agent get_current_time

content:
parts:
  0:
    functionCall:
      id: "adk-561d0294-24df-433f-9dc4-b85e2458a58b"
      args:
        city: "New York"
        name: "get_current_time"
role: "model"
invocation_id: "e-f04f9e47-56c0-444a-80e0-54efd7d0cf9d"
author: "weather_time_agent"
actions:
  state_delta:
  artifact_delta:
  requested_auth_configs:
long_running_tool_ids:
id: "4gVgAsPm"
timestamp: 1744205181.188365
  
```

On the `Events` tab, you can also click the `Trace` button to see the trace logs for each event that shows the latency of each function calls:



Step 5. You can also enable your microphone and talk to your agent:

Model support for voice/video streaming

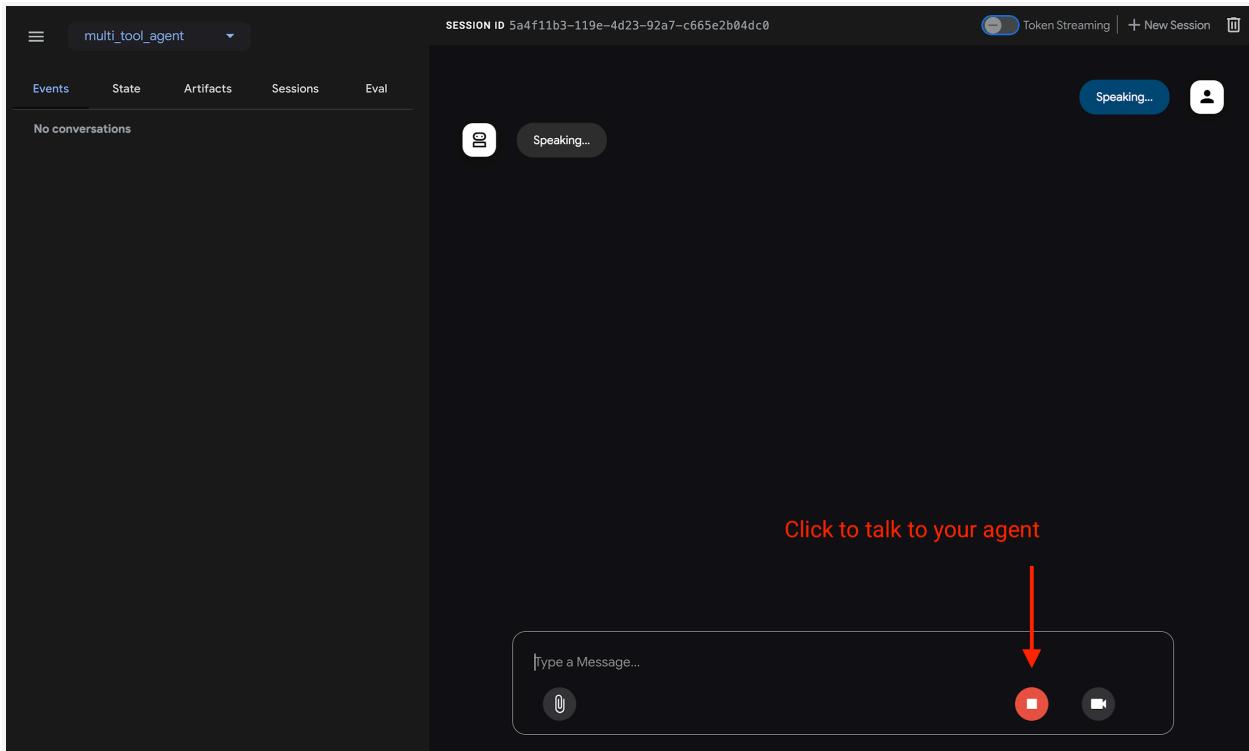
In order to use voice/video streaming in ADK, you will need to use Gemini models that support the Live API. You can find the **model ID(s)** that supports the Gemini Live API in the documentation:

- Google AI Studio: Gemini Live API
- Vertex AI: Gemini Live API

You can then replace the `model` string in `root_agent` in the `agent.py` file you created earlier ([jump to section](#)). Your code should look something like:

```
root_agent = Agent(
```

```
    name="weather_time_agent",  
  
    model="replace-me-with-model-id", #e.g. gemini-2.0-flash-live-001  
  
    ...
```



Example prompts to try

- What is the weather in New York?
- What is the time in New York?
- What is the weather in Paris?
- What is the time in Paris?

Congratulations!

You've successfully created and interacted with your first agent using ADK!

Next steps

- **Go to the tutorial:** Learn how to add memory, session, state to your agent: [tutorial](#).
- **Delve into advanced configuration:** Explore the [setup](#) section for deeper dives into project structure, configuration, and other interfaces.
- **Understand Core Concepts:** Learn about [agents concepts](#).

Testing your Agents

Before you deploy your agent, you should test it to ensure that it is working as intended. The easiest way to test your agent in your development environment is to use the ADK web UI with the following commands.

Python

Java

```
adk api_server
```

This command will launch a local web server, where you can run cURL commands or send API requests to test your agent.

Local testing

Local testing involves launching a local web server, creating a session, and sending queries to your agent. First, ensure you are in the correct working directory:

```
parent_folder/
```

```
└── my_sample_agent/
```

```
    └── agent.py (or Agent.java)
```

Launch the Local Server

Next, launch the local server using the commands listed above.

The output should appear similar to:

Python

Java

```
INFO:     Started server process [12345]
```

```
INFO:     Waiting for application startup.
```

```
INFO:     Application startup complete.
```

```
INFO:     Uvicorn running on http://localhost:8000 (Press CTRL+C to quit)
```

Your server is now running locally. Ensure you use the correct **port number** in all the subsequent commands.

Create a new session

With the API server still running, open a new terminal window or tab and create a new session with the agent using:

```
curl -X POST  
http://localhost:8000/apps/my_sample_agent/users/u_123/sessions/s_123 \
```

```
-H "Content-Type: application/json" \
```

```
-d '{"state": {"key1": "value1", "key2": 42}}'
```

Let's break down what's happening:

- `http://localhost:8000/apps/my_sample_agent/users/u_123/sessions/s_123`: This creates a new session for your agent `my_sample_agent`, which is the name of the agent folder, for a user ID (`u_123`) and for a session ID (`s_123`). You can replace `my_sample_agent` with the name of your agent folder. You can replace `u_123` with a specific user ID, and `s_123` with a specific session ID.
- `{"state": {"key1": "value1", "key2": 42}}`: This is optional. You can use this to customize the agent's pre-existing state (dict) when creating the session.

This should return the session information if it was created successfully. The output should appear similar to:

```
{"id": "s_123", "appName": "my_sample_agent", "userId": "u_123", "state": {"state": {"key1": "value1", "key2": 42}}, "events": [], "lastUpdateTime": 1743711430.022186}
```

Info

You cannot create multiple sessions with exactly the same user ID and session ID. If you try to, you may see a response, like: `{"detail": "Session already exists: s_123"}`. To fix this, you can either delete that session (e.g., `s_123`), or choose a different session ID.

Send a query

There are two ways to send queries via POST to your agent, via the `/run` or `/run_sse` routes.

- `POST http://localhost:8000/run`: collects all events as a list and returns the list all at once. Suitable for most users (if you are unsure, we recommend using this one).
- `POST http://localhost:8000/run_sse`: returns as Server-Sent-Events, which is a stream of event objects. Suitable for those who want to be notified as soon as the event is available. With `/run_sse`, you can also set `streaming` to `true` to enable token-level streaming.

Using `/run`

```
curl -X POST http://localhost:8000/run \
-H "Content-Type: application/json" \
-d '{
  "appName": "my_sample_agent",
  "userId": "u_123",
  "sessionId": "s_123",
  "newMessage": {
    "role": "user",
    "parts": [
      "text": "Hey whats the weather in new york today"
    ]
  }
}'
```

```
}'
```

If using `/run`, you will see the full output of events at the same time, as a list, which should appear similar to:

```
[{"content": {"parts": [{"functionCall": {"id": "af-e75e946d-c02a-4aad-931e-49e4ab859838", "args": {"city": "new  
york"}, "name": "get_weather"}]}, "role": "model", "invocationId": "e-71353f1e-aea1-4821-aa4b-46874a766853", "author": "weather_time_agent", "actions": {"stateDelta": {}, "artifactDelta": {}, "requestedAuthConfigs": {}}, "longRunningToolIds": [], "id": "2Btee6zW", "timestamp": 1743712220.385936}, {"content": {"parts": [{"functionResponse": {"id": "af-e75e946d-c02a-4aad-931e-49e4ab859838", "name": "get_weather", "response": {"status": "success", "report": "The weather in New York is sunny with a temperature of 25 degrees Celsius (41 degrees Farenheit)."}}}, {"role": "user", "invocationId": "e-71353f1e-aea1-4821-aa4b-46874a766853", "author": "weather_time_agent", "actions": {"stateDelta": {}, "artifactDelta": {}, "requestedAuthConfigs": {}}, "id": "PmWibL2m", "timestamp": 1743712221.895042}, {"content": {"parts": [{"text": "OK. The weather in New York is sunny with a temperature of 25 degrees Celsius (41 degrees Farenheit).\n"}]}, {"role": "model", "invocationId": "e-71353f1e-aea1-4821-aa4b-46874a766853", "author": "weather_time_agent", "actions": {"stateDelta": {}, "artifactDelta": {}, "requestedAuthConfigs": {}}, "id": "sYT42eVC", "timestamp": 1743712221.899018}]]
```

Using `/run_sse`

```
curl -X POST http://localhost:8000/run_sse \  
  
-H "Content-Type: application/json" \  
  
-d '{
```

```
"appName": "my_sample_agent",  
  
"userId": "u_123",  
  
"sessionId": "s_123",  
  
"newMessage": {  
    "role": "user",  
  
    "parts": [{  
        "text": "Hey whats the weather in new york today"  
    }]  
},  
  
"streaming": false  
}'
```

You can set `streaming` to `true` to enable token-level streaming, which means the response will be returned to you in multiple chunks and the output should appear similar to:

```
data:  
{"content": {"parts": [{"functionCall": {"id": "af-f83f8af9-f732-46b6-8cb5-7b5b73b  
bf13d", "args": {"city": "new  
york"}, "name": "get_weather"}]}, "role": "model", "invocationId": "e-3f6d7765-5287  
-419e-9991-5ffa1a75565", "author": "weather_time_agent", "actions": {"stateDelta": {}},  
"artifactDelta": {}, "requestedAuthConfigs": {}}, "longRunningToolIds": [], "id":  
"ptcjaZBa", "timestamp": 1743712255.313043}
```

```
data:  
{"content": {"parts": [{"functionResponse": {"id": "af-f83f8af9-f732-46b6-8cb5-7b5  
b73bbf13d", "name": "get_weather", "response": {"status": "success", "report": "The  
weather in New York is sunny with a temperature of 25 degrees Celsius (41  
degrees  
Fahrenheit)."}}], "role": "user", "invocationId": "e-3f6d7765-5287-419e-9991-5ff  
fa1a75565", "author": "weather_time_agent", "actions": {"stateDelta": {}}, "artifactD  
elta": {}, "requestedAuthConfigs": {}}, "id": "5aocxjaq", "timestamp": 1743712257.387  
306}
```

```
data: {"content": {"parts": [{"text": "OK. The weather in New York is sunny with  
a temperature of 25 degrees Celsius (41 degrees  
Fahrenheit).\n"}]}, "role": "model", "invocationId": "e-3f6d7765-5287-419e-9991-5f  
ffa1a75565", "author": "weather_time_agent", "actions": {"stateDelta": {}}, "artifact  
Delta": {}, "requestedAuthConfigs": {}}, "id": "rAnWGSiV", "timestamp": 1743712257.39  
1317}
```

Info

If you are using `/run_sse`, you should see each event as soon as it becomes available.

Integrations

ADK uses [Callbacks](#) to integrate with third-party observability tools. These integrations capture detailed traces of agent calls and interactions, which are crucial for understanding behavior, debugging issues, and evaluating performance.

- [Comet Opik](#) is an open-source LLM observability and evaluation platform that natively supports ADK.

Deploying your agent

Now that you've verified the local operation of your agent, you're ready to move on to deploying your agent! Here are some ways you can deploy your agent:

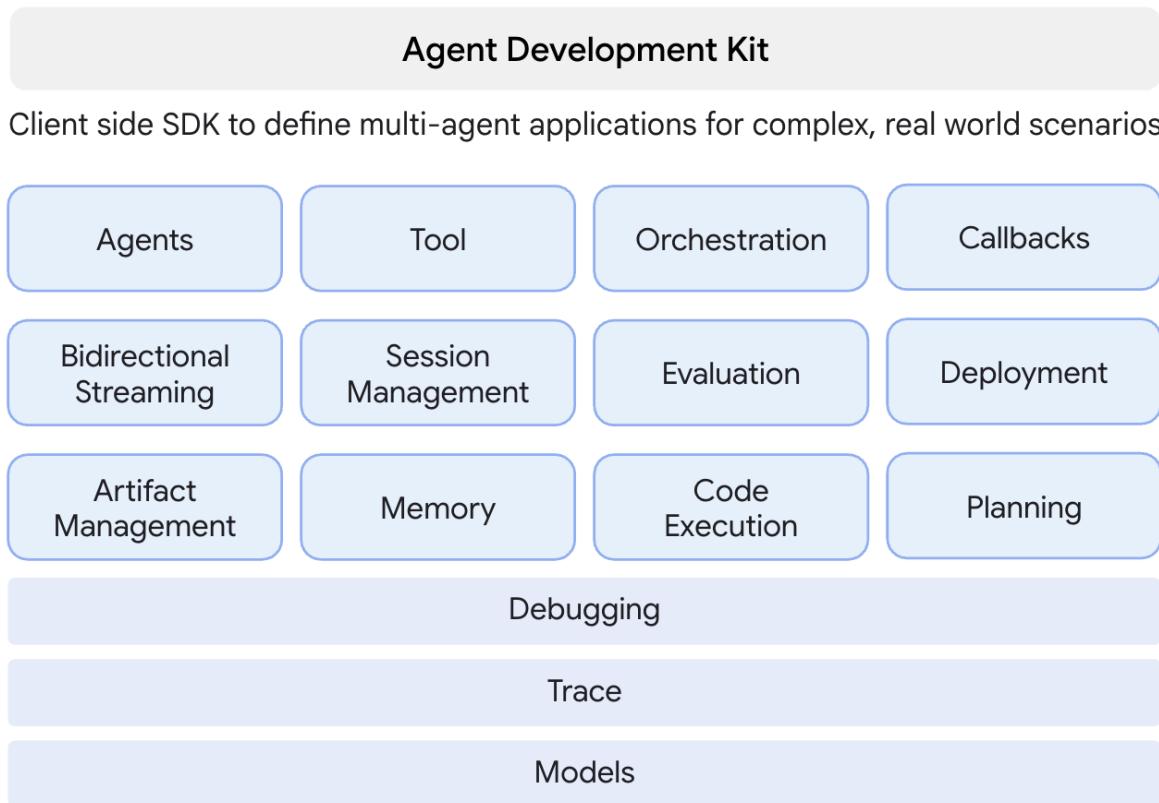
- Deploy to [Agent Engine](#), the easiest way to deploy your ADK agents to a managed service in Vertex AI on Google Cloud.
- Deploy to [Cloud Run](#) and have full control over how you scale and manage your agents using serverless architecture on Google Cloud.

Agent Development Kit (ADK)

Build, Evaluate and Deploy agents, seamlessly!

ADK is designed to empower developers to build, manage, evaluate and deploy AI-powered agents. It provides a robust and flexible environment for creating both conversational and non-conversational agents, capable of handling complex tasks and workflows.

Discover, build and deploy agents



Core Concepts

ADK is built around a few key primitives and concepts that make it powerful and flexible. Here are the essentials:

- **Agent:** The fundamental worker unit designed for specific tasks. Agents can use language models (`LlmAgent`) for complex reasoning, or act as deterministic controllers of the execution, which are called "[workflow agents](#)" (`SequentialAgent`, `ParallelAgent`, `LoopAgent`).
- **Tool:** Gives agents abilities beyond conversation, letting them interact with external APIs, search information, run code, or call other services.
- **Callbacks:** Custom code snippets you provide to run at specific points in the agent's process, allowing for checks, logging, or behavior modifications.

- **Session Management (Session & State):** Handles the context of a single conversation (`Session`), including its history (`Events`) and the agent's working memory for that conversation (`State`).
- **Memory:** Enables agents to recall information about a user across *multiple* sessions, providing long-term context (distinct from short-term session `State`).
- **Artifact Management (Artifact):** Allows agents to save, load, and manage files or binary data (like images, PDFs) associated with a session or user.
- **Code Execution:** The ability for agents (usually via Tools) to generate and execute code to perform complex calculations or actions.
- **Planning:** An advanced capability where agents can break down complex goals into smaller steps and plan how to achieve them like a ReAct planner.
- **Models:** The underlying LLM that powers `LlmAgents`, enabling their reasoning and language understanding abilities.
- **Event:** The basic unit of communication representing things that happen during a session (user message, agent reply, tool use), forming the conversation history.
- **Runner:** The engine that manages the execution flow, orchestrates agent interactions based on Events, and coordinates with backend services.

Note: Features like *Multimodal Streaming, Evaluation, Deployment, Debugging, and Trace* are also part of the broader ADK ecosystem, supporting real-time interaction and the development lifecycle.

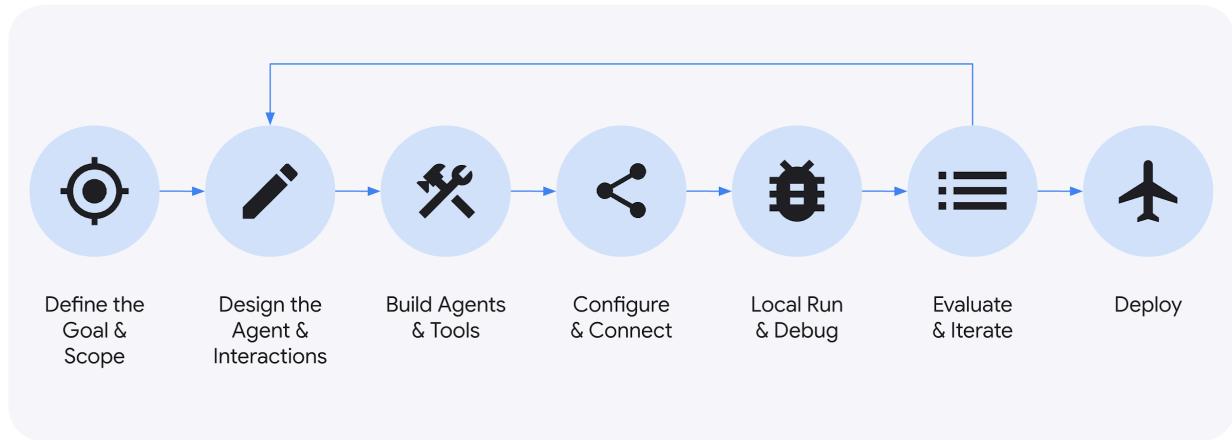
Key Capabilities

ADK offers several key advantages for developers building agentic applications:

1. **Multi-Agent System Design:** Easily build applications composed of multiple, specialized agents arranged hierarchically. Agents can coordinate complex tasks, delegate sub-tasks using LLM-driven transfer or explicit `AgentTool` invocation, enabling modular and scalable solutions.
2. **Rich Tool Ecosystem:** Equip agents with diverse capabilities. ADK supports integrating custom functions (`FunctionTool`), using other agents as tools (`AgentTool`), leveraging built-in functionalities like code execution, and interacting with external data sources and APIs (e.g., Search, Databases).

Support for long-running tools allows handling asynchronous operations effectively.

3. **Flexible Orchestration:** Define complex agent workflows using built-in workflow agents (`SequentialAgent`, `ParallelAgent`, `LoopAgent`) alongside LLM-driven dynamic routing. This allows for both predictable pipelines and adaptive agent behavior.
4. **Integrated Developer Tooling:** Develop and iterate locally with ease. ADK includes tools like a command-line interface (CLI) and a Developer UI for running agents, inspecting execution steps (events, state changes), debugging interactions, and visualizing agent definitions.
5. **Native Streaming Support:** Build real-time, interactive experiences with native support for bidirectional streaming (text and audio). This integrates seamlessly with underlying capabilities like the [Multimodal Live API](#) for the [Gemini Developer API](#) (or for [Vertex AI](#)), often enabled with simple configuration changes.
6. **Built-in Agent Evaluation:** Assess agent performance systematically. The framework includes tools to create multi-turn evaluation datasets and run evaluations locally (via CLI or the dev UI) to measure quality and guide improvements.
7. **Broad LLM Support:** While optimized for Google's Gemini models, the framework is designed for flexibility, allowing integration with various LLMs (potentially including open-source or fine-tuned models) through its `BaseLlm` interface.
8. **Artifact Management:** Enable agents to handle files and binary data. The framework provides mechanisms (`ArtifactService`, context methods) for agents to save, load, and manage versioned artifacts like images, documents, or generated reports during their execution.
9. **Extensibility and Interoperability:** ADK promotes an open ecosystem. While providing core tools, it allows developers to easily integrate and reuse tools from other popular agent frameworks including LangChain and CrewAI.
10. **State and Memory Management:** Automatically handles short-term conversational memory (`State` within a `Session`) managed by the `SessionService`. Provides integration points for longer-term `Memory` services, allowing agents to recall user information across multiple sessions.



Get Started

- Ready to build your first agent? [Try the quickstart](#)

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



[Open in Colab](#)



Share to:



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

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

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

Why a Weather Bot Team?

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

What is ADK Again?

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

In this advanced tutorial, you will master:

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

block requests/tool usage based on predefined rules, enhancing application safety and control.

End State Expectation:

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

Prerequisites:

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

Note on Execution Environment:

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

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

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

If you prefer a setup that handles the runner and session management automatically using ADK's standard tools, you can find the equivalent code structured for that purpose [here](#). That version is designed to be run directly with commands like `adk web` (for a web

UI), `adk run` (for CLI interaction), or `adk api_server` (to expose an API). Please follow the `README.md` instructions provided in that alternative resource.

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

Note: This tutorial works with adk version 1.0.0 and above

```
# @title Step 0: Setup and Installation

# Install ADK and LiteLLM for multi-model support

!pip install google-adk -q

!pip install litellm -q

print("Installation complete.")

# @title Import necessary libraries

import os
```

```
import asyncio

from google.adk.agents import Agent

from google.adk.models.lite_llm import LiteLlm # For multi-model support

from google.adk.sessions import InMemorySessionService

from google.adk.runners import Runner

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

import warnings

# Ignore all warnings

warnings.filterwarnings("ignore")
```

```
import logging

logging.basicConfig(level=logging.ERROR)

print("Libraries imported.")

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

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

# Gemini API Key (Get from Google AI Studio:
# https://aistudio.google.com/app/apikey)

os.environ["GOOGLE_API_KEY"] = "YOUR_GOOGLE_API_KEY" # <--- REPLACE
```

```
# [Optional]

# OpenAI API Key (Get from OpenAI Platform:
# https://platform.openai.com/api-keys)

os.environ['OPENAI_API_KEY'] = 'YOUR_OPENAI_API_KEY' # <--- REPLACE

# [Optional]

# Anthropic API Key (Get from Anthropic Console:
# https://console.anthropic.com/settings/keys)

os.environ['ANTHROPIC_API_KEY'] = 'YOUR_ANTHROPIC_API_KEY' # <--- REPLACE

# --- Verify Keys (Optional Check) ---

print("API Keys Set:")
```

```
print(f"Google API Key set: {'Yes' if os.environ.get('GOOGLE_API_KEY') and  
os.environ['GOOGLE_API_KEY'] != 'YOUR_GOOGLE_API_KEY' else 'No (REPLACE  
PLACEHOLDER!)'}")
```

```
print(f"OpenAI API Key set: {'Yes' if os.environ.get('OPENAI_API_KEY') and  
os.environ['OPENAI_API_KEY'] != 'YOUR_OPENAI_API_KEY' else 'No (REPLACE  
PLACEHOLDER!)'}")
```

```
print(f"Anthropic API Key set: {'Yes' if os.environ.get('ANTHROPIC_API_KEY')  
and os.environ['ANTHROPIC_API_KEY'] != 'YOUR_ANTHROPIC_API_KEY' else 'No  
(REPLACE PLACEHOLDER!)'}")
```

```
# Configure ADK to use API keys directly (not Vertex AI for this multi-model  
setup)
```

```
os.environ[ "GOOGLE_GENAI_USE_VERTEXAI" ] = "False"
```

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

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

```
# More supported models can be referenced here:  
https://ai.google.dev/gemini-api/docs/models#model-variations
```

```
MODEL_GEMINI_2_0_FLASH = "gemini-2.0-flash"
```

```
# More supported models can be referenced here:  
https://docs.litellm.ai/docs/providers/openai#openai-chat-completion-models
```

```
MODEL_GPT_40 = "openai/gpt-4.1" # You can also try: gpt-4.1-mini, gpt-4o etc.
```

```
# More supported models can be referenced here:  
https://docs.litellm.ai/docs/providers/anthropic
```

```
MODEL_CLAUDE SONNET = "anthropic/clause-sonnet-4-20250514" # You can also try:  
clause-opus-4-20250514 , clause-3-7-sonnet-20250219 etc
```

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

Step 1: Your First Agent - Basic Weather Lookup

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

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

1. Define the Tool (`get_weather`)

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

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

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

- *What* the tool does.

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

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

```
# @title Define the get_weather Tool
```

```
def get_weather(city: str) -> dict:
```

```
    """Retrieves the current weather report for a specified city.
```

Args:

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

Returns:

```
    dict: A dictionary containing the weather information.
```

```
    Includes a 'status' key ('success' or 'error').
```

```
If 'success', includes a 'report' key with weather details.
```

```
If 'error', includes an 'error_message' key.
```

```
"""
```

```
    print(f"--- Tool: get_weather called for city: {city} ---") # Log tool
execution
```

```
    city_normalized = city.lower().replace(" ", "") # Basic normalization
```

```
# Mock weather data
```

```
mock_weather_db = {
```

```
    "newyork": {"status": "success", "report": "The weather in New York is
sunny with a temperature of 25°C."},
```

```
    "london": {"status": "success", "report": "It's cloudy in London with a
temperature of 15°C."},
```

```
        "tokyo": {"status": "success", "report": "Tokyo is experiencing light  
rain and a temperature of 18°C."},
```

```
}
```

```
    if city_normalized in mock_weather_db:
```

```
        return mock_weather_db[city_normalized]
```

```
    else:
```

```
        return {"status": "error", "error_message": f"Sorry, I don't have  
weather information for '{city}'."}
```

```
# Example tool usage (optional test)
```

```
print(get_weather("New York"))
```

```
print(get_weather("Paris"))
```

2. Define the Agent (`weather_agent`)

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

We configure it with several key parameters:

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

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

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

```
# @title Define the Weather Agent

# Use one of the model constants defined earlier

AGENT_MODEL = MODEL_GEMINI_2_0_FLASH # Starting with Gemini

weather_agent = Agent(
```

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

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

3. Setup Runner and Session Service

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

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

```
# @title Setup Session Service and Runner
```

```
# --- Session Management ---
```

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

```
# InMemorySessionService is simple, non-persistent storage for this tutorial.
```

```
session_service = InMemorySessionService()
```

```
# Define constants for identifying the interaction context

APP_NAME = "weather_tutorial_app"

USER_ID = "user_1"

SESSION_ID = "session_001" # Using a fixed ID for simplicity

# Create the specific session where the conversation will happen

session = await session_service.create_session(
    app_name=APP_NAME,
    user_id=USER_ID,
    session_id=SESSION_ID
```

```
)
```

```
print(f"Session created: App='{APP_NAME}', User='{USER_ID}',  
Session='{SESSION_ID}'")
```

```
# --- Runner ---
```

```
# Key Concept: Runner orchestrates the agent execution loop.
```

```
runner = Runner(
```

```
    agent=weather_agent, # The agent we want to run
```

```
    app_name=APP_NAME, # Associates runs with our app
```

```
    session_service=session_service # Uses our session manager
```

```
)
```

```
print(f"Runner created for agent '{runner.agent.name}'.")
```

4. Interact with the Agent

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

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

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

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

```
# @title Define Agent Interaction Function
```

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

```
async def call_agent_async(query: str, runner, user_id, session_id):
```

```
    """Sends a query to the agent and prints the final response."""
```

```
print(f"\n>>> User Query: {query}")

# Prepare the user's message in ADK format

content = types.Content(role='user', parts=[types.Part(text=query)])

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

# Key Concept: run_async executes the agent logic and yields Events.

# We iterate through events to find the final answer.

async for event in runner.run_async(user_id=user_id, session_id=session_id,
new_message=content):

    # You can uncomment the line below to see *all* events during execution
```

```
# print(f" [Event] Author: {event.author}, Type: {type(event).__name__},  
Final: {event.is_final_response()}, Content: {event.content}")  
  
# Key Concept: is_final_response() marks the concluding message for the  
turn.  
  
if event.is_final_response():  
  
    if event.content and event.content.parts:  
  
        # Assuming text response in the first part  
  
        final_response_text = event.content.parts[0].text  
  
    elif event.actions and event.actions.escalate: # Handle potential  
errors/escalations  
  
        final_response_text = f"Agent escalated: {event.error_message or  
'No specific message.'}"  
  
    # Add more checks here if needed (e.g., specific error codes)
```

```
break # Stop processing events once the final response is found

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

5. Run the Conversation

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

Watch the output:

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

```
# @title Run the Initial Conversation

# We need an async function to await our interaction helper

async def run_conversation():
```

```
    await call_agent_async("What is the weather like in London?",  
                          runner=runner,  
  
                          user_id=USER_ID,  
  
                          session_id=SESSION_ID)  
  
    await call_agent_async("How about Paris?",  
                          runner=runner,  
  
                          user_id=USER_ID,  
  
                          session_id=SESSION_ID) # Expecting the  
tool's error message  
  
    await call_agent_async("Tell me the weather in New York",
```

```
    runner=runner,  
  
    user_id=USER_ID,  
  
    session_id=SESSION_ID)  
  
# Execute the conversation using await in an async context (like  
# Colab/Jupyter)  
  
await run_conversation()  
  
# --- OR ---  
  
# Uncomment the following lines if running as a standard Python script (.py  
file):  
  
# import asyncio
```

```
# if __name__ == "__main__":
#     try:
#         asyncio.run(run_conversation())
#
#     except Exception as e:
#         print(f"An error occurred: {e}")

```

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

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

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

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

- **Performance:** Some models excel at specific tasks (e.g., coding, reasoning, creative writing).

- **Cost:** Different models have varying price points.
- **Capabilities:** Models offer diverse features, context window sizes, and fine-tuning options.
- **Availability/Redundancy:** Having alternatives ensures your application remains functional even if one provider experiences issues.

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

In this step, we will:

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

1. Import `LiteLlm`

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

```
# @title 1. Import LiteLlm

from google.adk.models.lite_llm import LiteLlm
```

2. Define and Test Multi-Model Agents

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

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

Make sure you have configured the necessary API keys for OpenAI and Anthropic in Step 0. We'll use the `call_agent_async` function (defined earlier, which now accepts

`runner`, `user_id`, and `session_id`) to interact with each agent immediately after its setup.

Each block below will:

- Define the agent using a specific LiteLLM model (`MODEL_GPT_40` 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_40`, `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_40),
        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_40}' .")
```

```
# InMemorySessionService is simple, non-persistent storage for this  
tutorial.
```

```
session_service_gpt = InMemorySessionService() # Create a dedicated service
```

```
# Define constants for identifying the interaction context
```

```
APP_NAME_GPT = "weather_tutorial_app_gpt" # Unique app name for this test
```

```
USER_ID_GPT = "user_1_gpt"
```

```
SESSION_ID_GPT = "session_001_gpt" # Using a fixed ID for simplicity

# Create the specific session where the conversation will happen

    session_gpt = await session_service_gpt.create_session(
        app_name=APP_NAME_GPT,
        user_id=USER_ID_GPT,
        session_id=SESSION_ID_GPT
    )

    print(f"Session created: App={APP_NAME_GPT}, User={USER_ID_GPT}, Session={SESSION_ID_GPT}")

# Create a runner specific to this agent and its session service
```

```
runner_gpt = Runner(  
  
    agent=weather_agent_gpt,  
  
    app_name=APP_NAME_GPT,          # Use the specific app name  
  
    session_service=session_service_gpt # Use the specific session service  
  
)  
  
print(f"Runner created for agent '{runner_gpt.agent.name}'.")  
  
# --- Test the GPT Agent ---  
  
print("\n--- Testing GPT Agent ---")  
  
# Ensure call_agent_async uses the correct runner, user_id, session_id  
  
await call_agent_async(query = "What's the weather in Tokyo?",
```



```
#             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_40}'. Check API  
Key and model name. Error: {e}")
```

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

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

```
# --- Agent using Claude Sonnet ---

weather_agent_claude = None # Initialize to None

runner_claude = None      # Initialize runner to None

try:

    weather_agent_claude = Agent(
        name="weather_agent_claude",
        # Key change: Wrap the LiteLLM model identifier
        model=LiteLlm(model=MODEL_CLAUDE SONNET),
        description="Provides weather information (using Claude Sonnet).",
```

```
instruction="You are a helpful weather assistant powered by Claude
Sonnet. "
    "Use the 'get_weather' tool for city weather requests. "
    "Analyze the tool's dictionary output ('status',
'report'/'error_message'). "
    "Clearly present successful reports or polite error
messages.",
tools=[get_weather], # Re-use the same tool
)
print(f"Agent '{weather_agent_claude.name}' created using model
'{MODEL_CLAUDE SONNET}' .")
# InMemorySessionService is simple, non-persistent storage for this
tutorial.
session_service_claude = InMemorySessionService() # Create a dedicated
service
```

```
# Define constants for identifying the interaction context

APP_NAME_CLAUDE = "weather_tutorial_app_claude" # Unique app name

USER_ID_CLAUDE = "user_1_claude"

SESSION_ID_CLAUDE = "session_001_claude" # Using a fixed ID for simplicity

# Create the specific session where the conversation will happen

session_claude = await session_service_claude.create_session(
    app_name=APP_NAME_CLAUDE,
    user_id=USER_ID_CLAUDE,
    session_id=SESSION_ID_CLAUDE
```

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

```
# --- Test the Claude Agent ---  
  
print("\n--- Testing Claude Agent ---")  
  
# Ensure call_agent_async uses the correct runner, user_id, session_id  
  
await call_agent_async(query = "Weather in London please.",  
                       runner=runner_claude,  
                       user_id=USER_ID_CLAUDE,  
                       session_id=SESSION_ID_CLAUDE)  
  
# --- OR ---  
  
# Uncomment the following lines if running as a standard Python script (.py  
file):
```

```
# import asyncio

# if __name__ == "__main__":
#     try:
#         asyncio.run(call_agent_async(query = "Weather in London please.",
#                                       runner=runner_claude,
#                                       user_id=USER_ID_CLAUDE,
#                                       session_id=SESSION_ID_CLAUDE))

#     except Exception as e:
#         print(f"An error occurred: {e}")

except Exception as e:
```

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

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

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

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

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

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

In Steps 1 and 2, we built and experimented with a single agent focused solely on weather lookups. While effective for its specific task, real-world applications often involve handling a wider variety of user interactions. We *could* keep adding more tools

and complex instructions to our single weather agent, but this can quickly become unmanageable and less efficient.

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

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

Why build an Agent Team?

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

In this step, we will:

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

1. Define Tools for Sub-Agents

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

```
# @title Define Tools for Greeting and Farewell Agents
```

```
from typing import Optional # Make sure to import Optional

# Ensure 'get_weather' from Step 1 is available if running this step
# independently.

# def get_weather(city: str) -> dict: ... (from Step 1)

def say_hello(name: Optional[str] = None) -> str:

    """Provides a simple greeting. If a name is provided, it will be used.

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

```
Returns:
```

```
    str: A friendly greeting message.
```

```
    """
```

```
if name:
```

```
    greeting = f"Hello, {name}!"
```

```
    print(f"--- Tool: say_hello called with name: {name} ---")
```

```
else:
```

```
    greeting = "Hello there!" # Default greeting if name is None or not explicitly passed
```

```
    print(f"--- Tool: say_hello called without a specific name (name_arg_value: {name}) ---")
```

```
return greeting
```

```
def say_goodbye() -> str:  
  
    """Provides a simple farewell message to conclude the conversation."""  
  
    print(f"--- Tool: say_goodbye called ---")  
  
    return "Goodbye! Have a great day."  
  
print("Greeting and Farewell tools defined.")  
  
# Optional self-test  
  
print(say_hello("Alice"))  
  
print(say_hello()) # Test with no argument (should use default "Hello there!")  
  
print(say_hello(name=None)) # Test with name explicitly as None (should use  
default "Hello there!")
```

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

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

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

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

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

```
# If you want to use models other than Gemini, Ensure LiteLlm is imported and
API keys are set (from Step 0/2)
```

```
# from google.adk.models.lite_llm import LiteLlm
```

```
# MODEL_GPT_40, 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_40), # 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_40), # If you would like to experiment  
with other models  
  
    name="farewell_agent",  
  
    instruction="You are the Farewell Agent. Your ONLY task is to provide a  
polite goodbye message."  
  
        "Use the 'say_goodbye' tool when the user indicates they  
are leaving or ending the conversation "  
  
        "(e.g., using words like 'bye', 'goodbye', 'thanks bye',  
'see you'). "  
  
    "Do not perform any other actions.",  
  
    description="Handles simple farewells and goodbyes using the  
'say_goodbye' tool.", # Crucial for delegation
```

```

        tools=[say_goodbye],

    )

    print(f"✓ Agent '{farewell_agent.name}' created using model
'{farewell_agent.model}'.")
except Exception as e:

    print(f"✗ Could not create Farewell agent. Check API Key
({farewell_agent.model}). Error: {e}")

```

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

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

- Adding the `sub_agents` parameter: We pass a list containing the `greeting_agent` and `farewell_agent` instances we just created.
- Updating the `instruction`: We explicitly tell the root agent *about* its sub-agents and *when* it should delegate tasks to them.

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

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

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

# Ensure sub-agents were created successfully before defining the root agent.

# Also ensure the original 'get_weather' tool is defined.

root_agent = None

runner_root = None # Initialize runner

if greeting_agent and farewell_agent and 'get_weather' in globals():

    # Let's use a capable Gemini model for the root agent to handle
    # orchestration

    root_agent_model = MODEL_GEMINI_2_0_FLASH
```

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

```
        "Analyze the user's query. If it's a greeting, delegate to  
'greeting_agent'. If it's a farewell, delegate to 'farewell_agent'. "
```

```
        "If it's a weather request, handle it yourself using  
'get_weather'. "
```

```
        "For anything else, respond appropriately or state you  
cannot handle it.",
```

```
    tools=[get_weather], # Root agent still needs the weather tool for its  
core task
```

```
# Key change: Link the sub-agents here!
```

```
    sub_agents=[greeting_agent, farewell_agent]
```

```
)
```

```
print(f"✓ Root Agent '{weather_agent_team.name}' created using model  
'{root_agent_model}' with sub-agents: {[sa.name for sa in  
weather_agent_team.sub_agents]}")
```

```
else:
```

```
    print("✖ Cannot create root agent because one or more sub-agents failed  
to initialize or 'get_weather' tool is missing.")
```

```
if not greeting_agent: print(" - Greeting Agent is missing.")
```

```
if not farewell_agent: print(" - Farewell Agent is missing.")
```

```
if 'get_weather' not in globals(): print(" - get_weather function is  
missing.")
```

4. Interact with the Agent Team

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

The following code block will:

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

We expect the following flow:

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

```
# @title Interact with the Agent Team
```

```
import asyncio # Ensure asyncio is imported
```

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

```
# Ensure the call_agent_async function is defined.
```

```
# Check if the root agent variable exists before defining the conversation function
```

```
root_agent_var_name = 'root_agent' # Default name from Step 3 guide
```

```
if 'weather_agent_team' in globals(): # Check if user used this name instead

    root_agent_var_name = 'weather_agent_team'

elif 'root_agent' not in globals():

    print("⚠ Root agent ('root_agent' or 'weather_agent_team') not found.
Cannot define run_team_conversation.")

# Assign a dummy value to prevent NameError later if the code block runs
anyway

root_agent = None # Or set a flag to prevent execution

# Only define and run if the root agent exists

if root_agent_var_name in globals() and globals()[root_agent_var_name]:

    # Define the main async function for the conversation logic.

    # The 'await' keywords INSIDE this function are necessary for async
operations.
```

```
async def run_team_conversation():

    print("\n--- Testing Agent Team Delegation ---")

    session_service = InMemorySessionService()

    APP_NAME = "weather_tutorial_agent_team"

    USER_ID = "user_1_agent_team"

    SESSION_ID = "session_001_agent_team"

    session = await session_service.create_session(
        app_name=APP_NAME, user_id=USER_ID, session_id=SESSION_ID

    )

    print(f"Session created: App='{APP_NAME}', User='{USER_ID}', Session='{SESSION_ID}'")
```

```
actual_root_agent = globals()[root_agent_var_name]

runner_agent_team = Runner( # Or use InMemoryRunner

    agent=actual_root_agent,

    app_name=APP_NAME,

    session_service=session_service

)

print(f"Runner created for agent '{actual_root_agent.name}'.")

# --- Interactions using await (correct within async def) ---

await call_agent_async(query = "Hello there!",

    runner=runner_agent_team,
```

```
        user_id=USER_ID,  
  
        session_id=SESSION_ID)  
  
    await call_agent_async(query = "What is the weather in New York?",  
  
                          runner=runner_agent_team,  
  
                          user_id=USER_ID,  
  
                          session_id=SESSION_ID)  
  
    await call_agent_async(query = "Thanks, bye!",  
  
                          runner=runner_agent_team,  
  
                          user_id=USER_ID,  
  
                          session_id=SESSION_ID)  
  
# --- Execute the `run_team_conversation` async function ---
```

```
# Choose ONE of the methods below based on your environment.
```

```
# Note: This may require API keys for the models used!
```

```
# METHOD 1: Direct await (Default for Notebooks/Async REPLs)
```

```
# If your environment supports top-level await (like Colab/Jupyter notebooks),
```

```
# it means an event loop is already running, so you can directly await the function.
```

```
print("Attempting execution using 'await' (default for notebooks)...")
```

```
await run_team_conversation()
```

```
# METHOD 2: asyncio.run (For Standard Python Scripts [.py])
```

```
# If running this code as a standard Python script from your terminal,
```

```
# the script context is synchronous. `asyncio.run()` is needed to
```

```
# create and manage an event loop to execute your async function.
```

```
# To use this method:
```

```
# 1. Comment out the `await run_team_conversation()` line above.
```

```
# 2. Uncomment the following block:
```

```
"""
```

```
import asyncio
```

```
if __name__ == "__main__": # Ensures this runs only when script is executed  
directly
```

```
    print("Executing using 'asyncio.run()' (for standard Python  
scripts)...")
```

```
try:
```

```
# This creates an event loop, runs your async function, and closes
the loop.

asyncio.run(run_team_conversation())

except Exception as e:

    print(f"An error occurred: {e}")

    ...

else:

    # This message prints if the root agent variable wasn't found earlier

    print("\n⚠️ Skipping agent team conversation execution as the root agent
was not successfully defined in a previous step.")
```

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

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

This confirms successful **automatic delegation!** The root agent, guided by its instructions and the `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

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 = await session_service_stateful.create_session(
    app_name=APP_NAME, # Use the consistent app name

    user_id=USER_ID_STATEFUL,

    session_id=SESSION_ID_STATEFUL,

    state=initial_state # <<< Initialize state during creation

)

print(f"✓ Session '{SESSION_ID_STATEFUL}' created for user
'{USER_ID_STATEFUL}'")
```

```
# Verify the initial state was set correctly

retrieved_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,

session_id =
SESSION_ID_STATEFUL)

print("\n--- Initial Session State ---")

if retrieved_session:

    print(retrieved_session.state)

else:

    print("Error: Could not retrieve session.")
```

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

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

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

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

```
def get_weather_stateful(city: str, tool_context: ToolContext) -> dict:
```

""Retrieves weather, converts temp unit based on session state.""""

```
print(f"--- Tool: get_weather_stateful called for {city} ---")
```

```
# --- Read preference from state ---
```

```
    preferred_unit = tool_context.state.get("user_preference_temperature_unit",  
"Celsius") # Default to Celsius
```

```
    print(f"--- Tool: Reading state 'user_preference_temperature_unit':\n{preferred_unit} ---")\n\n\n    city_normalized = city.lower().replace(" ", "")\n\n\n    # Mock weather data (always stored in Celsius internally)\n\n\n    mock_weather_db = {\n\n        "newyork": {"temp_c": 25, "condition": "sunny"},\n\n        "london": {"temp_c": 15, "condition": "cloudy"},\n\n        "tokyo": {"temp_c": 18, "condition": "light rain"},\n\n    }\n\n
```

```
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_40, 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:

- 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.
- 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.
- 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`.
- 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.
- 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_STATE
FUL]

stored_session.state["user_preference_temperature_unit"] =
"Fahrenheit"

# Optional: You might want to update the timestamp as well if any
logic depends on it

# import time

# stored_session.last_update_time = time.time()

print(f"--- Stored session state updated. Current
'user_preference_temperature_unit':
{stored_session.state.get('user_preference_temperature_unit', 'Not Set')}
---") # Added .get for safety
```

```
except KeyError:

    print(f"--- Error: Could not retrieve session
'{SESSION_ID_STATEFUL}' from internal storage for user '{USER_ID_STATEFUL}' in
app '{APP_NAME}' to update state. Check IDs and if session was created. ---")

except Exception as e:

    print(f"--- Error updating internal session state: {e} ---")

# 3. Check weather again (Tool should now use Fahrenheit)

# This will also update 'last_weather_report' via output_key

print("\n--- Turn 2: Requesting weather in New York (expect Fahrenheit)
---")

await call_agent_async(query= "Tell me the weather in New York.",

runner=runner_root_stateful,

user_id=USER_ID_STATEFUL,
```

```
    session_id=SESSION_ID_STATEFUL
```

```
)
```

```
# 4. Test basic delegation (should still work)
```

```
# This will update 'last_weather_report' again, overwriting the NY
weather report
```

```
    print("\n--- Turn 3: Sending a greeting ---")
```

```
    await call_agent_async(query= "Hi!",
```

```
        runner=runner_root_stateful,
```

```
        user_id=USER_ID_STATEFUL,
```

```
        session_id=SESSION_ID_STATEFUL
```

```
)
```

```
# --- Execute the `run_stateful_conversation` async function ---
```

```
# Choose ONE of the methods below based on your environment.
```

```
# METHOD 1: Direct await (Default for Notebooks/Async REPLs)
```

```
# If your environment supports top-level await (like Colab/Jupyter notebooks),
```

```
# it means an event loop is already running, so you can directly await the function.
```

```
print("Attempting execution using 'await' (default for notebooks)...")
```

```
await run_stateful_conversation()
```

```
# METHOD 2: asyncio.run (For Standard Python Scripts [.py])

# If running this code as a standard Python script from your terminal,
# the script context is synchronous. `asyncio.run()` is needed to
# create and manage an event loop to execute your async function.

# To use this method:

# 1. Comment out the `await run_stateful_conversation()` line above.

# 2. Uncomment the following block:

"""

import asyncio

if __name__ == "__main__": # Ensures this runs only when script is executed
directly

    print("Executing using 'asyncio.run()' (for standard Python
scripts)...")
```

```
try:

    # This creates an event loop, runs your async function, and closes
    the loop.

    asyncio.run(run_stateful_conversation())

except Exception as e:

    print(f"An error occurred: {e}")

    ...

# --- Inspect final session state after the conversation ---


# This block runs after either execution method completes.

print("\n--- Inspecting Final Session State ---")

final_session = await
session_service_stateful.get_session(app_name=APP_NAME,
```

```
        user_id=
USER_ID_STATEFUL,

session_id=SESSION_ID_STATEFUL)

if final_session:

    # Use .get() for safer access to potentially missing keys

    print(f"Final Preference:
{final_session.state.get('user_preference_temperature_unit', 'Not Set')}")

    print(f"Final Last Weather Report (from output_key):
{final_session.state.get('last_weather_report', 'Not Set')}")

    print(f"Final Last City Checked (by tool):
{final_session.state.get('last_city_checked_stateful', 'Not Set')}")

# Print full state for detailed view

# print(f"Full State Dict: {final_session.state}") # For detailed view

else:
```

```
    print("\n⚠ Error: Could not retrieve final session state.")
```

```
else:
```

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

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

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

You've now successfully integrated session state to personalize agent behavior using `ToolContext`, manually manipulated state for testing `InMemorySessionService`, and observed how `output_key` provides a simple mechanism for saving the agent's last

response to state. This foundational understanding of state management is key as we proceed to implement safety guardrails using callbacks in the next steps.

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

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

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

What is `before_model_callback`?

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

Common Use Cases:

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

How it Works:

1. Define a function accepting `callback_context: CallbackContext` and `llm_request: LlmRequest`.
 - `callback_context`: Provides access to agent info, session state (`callback_context.state`), etc.
 - `llm_request`: Contains the full payload intended for the LLM (`contents, config`).
2. Inside the function:
 - **Inspect:** Examine `llm_request.contents` (especially the last user message).
 - **Modify (Use Caution):** You *can* change parts of `llm_request`.
 - **Block (Guardrail):** Return an `LlmResponse` object. ADK will send this response back immediately, *skipping* the LLM call for that turn.
 - **Allow:** Return `None`. ADK proceeds to call the LLM with the (potentially modified) request.

In this step, we will:

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

1. Define the Guardrail Callback Function

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

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

```
# Ensure necessary imports are available

from google.adk.agents.callback_context import CallbackContext

from google.adk.models.llm_request import LlmRequest

from google.adk.models.llm_response import LlmResponse

from google.genai import types # For creating response content

from typing import Optional

def block_keyword_guardrail(
    callback_context: CallbackContext, llm_request: LlmRequest
) -> Optional[LlmResponse]:
    """
    Inspects the latest user message for 'BLOCK'. If found, blocks the LLM call
    """

    ...
```

```
    and returns a predefined LlmResponse. Otherwise, returns None to proceed.
```

```
    """
```

```
    agent_name = callback_context.agent_name # Get the name of the agent whose
model call is being intercepted
```

```
    print(f"--- Callback: block_keyword_guardrail running for agent:
{agent_name} ---")
```

```
# Extract the text from the latest user message in the request history
```

```
last_user_message_text = ""
```

```
if llm_request.contents:
```

```
    # Find the most recent message with role 'user'
```

```
    for content in reversed(llm_request.contents):
```

```
if content.role == 'user' and content.parts:  
  
    # Assuming text is in the first part for simplicity  
  
    if content.parts[0].text:  
  
        last_user_message_text = content.parts[0].text  
  
        break # Found the last user message text  
  
print(f"--- Callback: Inspecting last user message:  
'{last_user_message_text[:100]}...'" ---") # Log first 100 chars  
  
# --- Guardrail Logic ---  
  
keyword_to_block = "BLOCK"  
  
if keyword_to_block in last_user_message_text.upper(): # Case-insensitive  
check
```

```
    print(f"--- Callback: Found '{keyword_to_block}'. Blocking LLM call!
---")

# Optionally, set a flag in state to record the block event

callback_context.state["guardrail_block_keyword_triggered"] = True

print(f"--- Callback: Set state 'guardrail_block_keyword_triggered': True ---")

# Construct and return an LlmResponse to stop the flow and send this
back instead

return LlmResponse(
    content=types.Content(
        role="model", # Mimic a response from the agent's perspective
        parts=[types.Part(text=f"I cannot process this request because
it contains the blocked keyword '{keyword_to_block}' .")],
    )
)
```

```
    )

# Note: You could also set an error_message field here if needed

)

else:

    # Keyword not found, allow the request to proceed to the LLM

    print(f"--- Callback: Keyword not found. Allowing LLM call for
{agent_name}. ---")

    return None # Returning None signals ADK to continue normally

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

2. Update Root Agent to Use the Callback

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

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

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

```
# --- Redefine Sub-Agents (Ensures they exist in this context) ---
```

```
greeting_agent = None
```

```
try:
```

```
    # Use a defined model constant
```

```
    greeting_agent = Agent(
```

```
        model=MODEL_GEMINI_2_0_FLASH,
```

```
        name="greeting_agent", # Keep original name for consistency
```

```
    instruction="You are the Greeting Agent. Your ONLY task is to provide a
friendly greeting using the 'say_hello' tool. Do nothing else.",

description="Handles simple greetings and hellos using the 'say_hello'
tool.",

tools=[say_hello],

)

print(f"✅ Sub-Agent '{greeting_agent.name}' redefined.")

except Exception as e:

    print(f"❌ Could not redefine Greeting agent. Check Model/API Key
({greeting_agent.model}). Error: {e}")

farewell_agent = None

try:

    # Use a defined model constant
```

```
farewell_agent = Agent(  
  
    model=MODEL_GEMINI_2_0_FLASH,  
  
    name="farewell_agent", # Keep original name  
  
    instruction="You are the Farewell Agent. Your ONLY task is to provide a  
polite goodbye message using the 'say_goodbye' tool. Do not perform any other  
actions.",  
  
    description="Handles simple farewells and goodbyes using the  
'say_goodbye' tool.",  
  
    tools=[say_goodbye],  
  
)  
  
print(f"✓ Sub-Agent '{farewell_agent.name}' redefined.")  
  
except Exception as e:  
  
    print(f"✗ Could not redefine Farewell agent. Check Model/API Key  
({farewell_agent.model}). Error: {e}")
```

```
# --- Define the Root Agent with the Callback ---

root_agent_model_guardrail = None

runner_root_model_guardrail = None

# Check all components before proceeding

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

    # Use a defined model constant

    root_agent_model = MODEL_GEMINI_2_0_FLASH
```

```
root_agent_model_guardrail = Agent(  
  
    name="weather_agent_v5_model_guardrail", # New version name for clarity  
  
    model=root_agent_model,  
  
    description="Main agent: Handles weather, delegates  
greetings/farewells, includes input keyword guardrail.",  
  
    instruction="You are the main Weather Agent. Provide weather using  
'get_weather_stateful'. "  
  
        "Delegate simple greetings to 'greeting_agent' and  
farewells to 'farewell_agent'. "  
  
    "Handle only weather requests, greetings, and farewells.",  
  
    tools=[get_weather],  
  
    sub_agents=[greeting_agent, farewell_agent], # Reference the redefined  
sub-agents
```

```
        output_key="last_weather_report", # Keep output_key from Step 4

    before_model_callback=block_keyword_guardrail # <<< Assign the
guardrail callback

    )

print(f"✓ Root Agent '{root_agent_model_guardrail.name}' created with
before_model_callback.")

# --- Create Runner for this Agent, Using SAME Stateful Session Service ---

# Ensure session_service_stateful exists from Step 4

if 'session_service_stateful' in globals():

    runner_root_model_guardrail = Runner(
        agent=root_agent_model_guardrail,
```

```
    app_name=APP_NAME, # Use consistent APP_NAME

Step 4

    )

    print(f"✓ Runner created for guardrail agent
'{runner_root_model.guardrail.agent.name}', using stateful session service.")

else:

    print("✗ Cannot create runner. 'session_service_stateful' from Step 4
is missing.")

else:

    print("✗ Cannot create root agent with model guardrail. One or more
prerequisites are missing or failed initialization:")

if not greeting_agent: print("    - Greeting Agent")
```

```
if not farewell_agent: print(" - Farewell Agent")

if 'get_weather_stateful' not in globals(): print(" - "
'get_weather_stateful' tool)

if 'block_keyword_guardrail' not in globals(): print(" - "
'block_keyword_guardrail' callback)
```

3. Interact to Test the Guardrail

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

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

```
# @title 3. Interact to Test the Model Input Guardrail
```

```
import asyncio # Ensure asyncio is imported
```

```
# Ensure the runner for the guardrail agent is available
```

```
if 'runner_root_model_guardrail' in globals() and runner_root_model_guardrail:
```

```
# Define the main async function for the guardrail test conversation.

# The 'await' keywords INSIDE this function are necessary for async
operations.

async def run_guardrail_test_conversation():

    print("\n--- Testing Model Input Guardrail ---")

    # Use the runner for the agent with the callback and the existing
stateful session ID

    # Define a helper lambda for cleaner interaction calls

    interaction_func = lambda query: call_agent_async(query,
runner_root_model_guardrail,
                                         USER_ID_STATEFUL, #
Use existing user ID
```

```
SESSION_ID_STATEFUL #  
Use existing session ID
```

```
)
```

```
# 1. Normal request (Callback allows, should use Fahrenheit from  
previous state change)
```

```
print(" --- Turn 1: Requesting weather in London (expect allowed,  
Fahrenheit) ---")
```

```
await interaction_func("What is the weather in London?")
```

```
# 2. Request containing the blocked keyword (Callback intercepts)
```

```
print("\n --- Turn 2: Requesting with blocked keyword (expect blocked)  
---")
```

```
await interaction_func("BLOCK the request for weather in Tokyo") #  
Callback should catch "BLOCK"
```

```
# 3. Normal greeting (Callback allows root agent, delegation happens)
```

```
    print("\n--- Turn 3: Sending a greeting (expect allowed) ---")
```

```
    await interaction_func("Hello again")
```

```
# --- Execute the `run_guardrail_test_conversation` async function ---
```

```
# Choose ONE of the methods below based on your environment.
```

```
# METHOD 1: Direct await (Default for Notebooks/Async REPLs)
```

```
# If your environment supports top-level await (like Colab/Jupyter notebooks),
```

```
# it means an event loop is already running, so you can directly await the function.
```

```
    print("Attempting execution using 'await' (default for notebooks)...")
```

```
    await run_guardrail_test_conversation()

# METHOD 2: asyncio.run (For Standard Python Scripts [.py])

# If running this code as a standard Python script from your terminal,
# the script context is synchronous. `asyncio.run()` is needed to
# create and manage an event loop to execute your async function.

# To use this method:

# 1. Comment out the `await run_guardrail_test_conversation()` line above.

# 2. Uncomment the following block:

    """
import asyncio
```

```
if __name__ == "__main__": # Ensures this runs only when script is executed
    directly

        print("Executing using 'asyncio.run()' (for standard Python
scripts)...")

    try:

        # This creates an event loop, runs your async function, and closes
the loop.

        asyncio.run(run_guardrail_test_conversation())

    except Exception as e:

        print(f"An error occurred: {e}")

    """

# --- Inspect final session state after the conversation ---

# This block runs after either execution method completes.
```

```
# Optional: Check state for the trigger flag set by the callback

print("\n--- Inspecting Final Session State (After Guardrail Test) ---")

# Use the session service instance associated with this stateful session

final_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,

session_id=SESSION_ID_STATEFUL)

if final_session:

    # Use .get() for safer access

    print(f"Guardrail Triggered Flag:
{final_session.state.get('guardrail_block_keyword_triggered', 'Not Set (or
False)')}"
```

```

        print(f"Last Weather Report:
{final_session.state.get('last_weather_report', 'Not Set')}") # Should be
London weather if successful

        print(f"Temperature Unit:
{final_session.state.get('user_preference_temperature_unit', 'Not Set')}") #
Should be Fahrenheit

# print(f"Full State Dict: {final_session.state}") # For detailed view

else:

    print("\n⚠️ Error: Could not retrieve final session state.")

else:

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

```

Observe the execution flow:

1. **London Weather:** The callback runs for `weather_agent_v5_model_guardrail`, inspects the message, prints "Keyword not found. Allowing LLM call.", and returns `None`. The agent proceeds, calls

the `get_weather_stateful` tool (which uses the "Fahrenheit" preference from Step 4's state change), and returns the weather. This response updates `last_weather_report` via `output_key`.

2. **BLOCK Request:** The callback runs again for `weather_agent_v5_model_guardrail`, inspects the message, finds "BLOCK", prints "Blocking LLM call!", sets the state flag, and returns the predefined `LlmResponse`. The agent's underlying LLM is *never called* for this turn. The user sees the callback's blocking message.
3. **Hello Again:** The callback runs for `weather_agent_v5_model_guardrail`, allows the request. The root agent then delegates to `greeting_agent`. *Note: The `before_model_callback` defined on the root agent does NOT automatically apply to sub-agents.* The `greeting_agent` proceeds normally, calls its `say_hello` tool, and returns the greeting.

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

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

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

ADK provides the `before_tool_callback` for this precise purpose.

What is `before_tool_callback`?

- It's a Python function executed just *before* a specific tool function runs, after the LLM has requested its use and decided on the arguments.
- **Purpose:** Validate tool arguments, prevent tool execution based on specific inputs, modify arguments dynamically, or enforce resource usage policies.

Common Use Cases:

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

How it Works:

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

In this step, we will:

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

1. Define the Tool Guardrail Callback Function

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

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

# Ensure necessary imports are available

from google.adk.tools.base_tool import BaseTool

from google.adk.tools.tool_context import ToolContext

from typing import Optional, Dict, Any # For type hints

def block_paris_tool_guardrail(
    tool: BaseTool, args: Dict[str, Any], tool_context: ToolContext
) -> Optional[Dict]:
```

```
....
```

```
Checks if 'get_weather_stateful' is called for 'Paris'.
```

```
If so, blocks the tool execution and returns a specific error dictionary.
```

```
Otherwise, allows the tool call to proceed by returning None.
```

```
....
```

```
tool_name = tool.name
```

```
agent_name = tool_context.agent_name # Agent attempting the tool call
```

```
    print(f"--- Callback: block_paris_tool_guardrail running for tool  
'{tool_name}' in agent '{agent_name}' ---")
```

```
    print(f"--- Callback: Inspecting args: {args} ---")
```

```
# --- Guardrail Logic ---
```

```
target_tool_name = "get_weather_stateful" # Match the function name used by
FunctionTool

blocked_city = "paris"

# Check if it's the correct tool and the city argument matches the blocked
city

if tool_name == target_tool_name:

    city_argument = args.get("city", "") # Safely get the 'city' argument

    if city_argument and city_argument.lower() == blocked_city:

        print(f"--- Callback: Detected blocked city '{city_argument}'.
Blocking tool execution! ---")

    # Optionally update state

    tool_context.state["guardrail_tool_block_triggered"] = True
```

```
        print(f"--- Callback: Set state 'guardrail_tool_block_triggered' :  
True ---")  
  
    # Return a dictionary matching the tool's expected output format  
for errors  
  
    # This dictionary becomes the tool's result, skipping the actual  
tool run.  
  
    return {  
  
        "status": "error",  
  
        "error_message": f"Policy restriction: Weather checks for  
'{city_argument.capitalize()}' are currently disabled by a tool guardrail."  
  
    }  
  
else:  
  
    print(f"--- Callback: City '{city_argument}' is allowed for tool  
'{tool_name}'. ---")
```

```
    else:

        print(f"--- Callback: Tool '{tool_name}' is not the target tool.
Allowing. ---")

# If the checks above didn't return a dictionary, allow the tool to execute

        print(f"--- Callback: Allowing tool '{tool_name}' to proceed. ---")

    return None # Returning None allows the actual tool function to run

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

2. Update Root Agent to Use Both Callbacks

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

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

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

```
# --- Ensure Prerequisites are Defined ---
```

```
# (Include or ensure execution of definitions for: Agent, LiteLlm, Runner,  
ToolContext,
```

```
# MODEL constants, say_hello, say_goodbye, greeting_agent, farewell_agent,
```

```
# get_weather_stateful, block_keyword_guardrail, block_paris_tool_guardrail)
```

```
# --- Redefine Sub-Agents (Ensures they exist in this context) ---
```

```
greeting_agent = None
```

```
try:
```

```
# Use a defined model constant

greeting_agent = Agent(
    model=MODEL_GEMINI_2_0_FLASH,
    name="greeting_agent", # Keep original name for consistency

    instruction="You are the Greeting Agent. Your ONLY task is to provide a
friendly greeting using the 'say_hello' tool. Do nothing else.",

    description="Handles simple greetings and hellos using the 'say_hello'
tool.",

    tools=[say_hello],

)

print(f"✅ Sub-Agent '{greeting_agent.name}' redefined.")

except Exception as e:

    print(f"❗ Could not redefine Greeting agent. Check Model/API Key
({greeting_agent.model}). Error: {e}")
```

```
farewell_agent = None

try:

    # Use a defined model constant

    farewell_agent = Agent(

        model=MODEL_GEMINI_2_0_FLASH,

        name="farewell_agent", # Keep original name

        instruction="You are the Farewell Agent. Your ONLY task is to provide a
polite goodbye message using the 'say_goodbye' tool. Do not perform any other
actions.",

        description="Handles simple farewells and goodbyes using the
'say_goodbye' tool.",

        tools=[say_goodbye],
```

```
    )

    print(f"✓ Sub-Agent '{farewell_agent.name}' redefined.")

except Exception as e:

    print(f"✗ Could not redefine Farewell agent. Check Model/API Key
({farewell_agent.model}). Error: {e}")

# --- Define the Root Agent with Both Callbacks ---

root_agent_tool_guardrail = None

runner_root_tool_guardrail = None

if ('greeting_agent' in globals() and greeting_agent and

    'farewell_agent' in globals() and farewell_agent and
```

```
'get_weather_stateful' in globals() and

'block_keyword_guardrail' in globals() and

'block_paris_tool_guardrail' in globals())):

root_agent_model = MODEL_GEMINI_2_0_FLASH

root_agent_tool_guardrail = Agent(
    name="weather_agent_v6_tool_guardrail", # New version name

    model=root_agent_model,

    description="Main agent: Handles weather, delegates, includes input AND
tool guardrails.",

    instruction="You are the main Weather Agent. Provide weather using
'get_weather_stateful'. "
```

```
        "Delegate greetings to 'greeting_agent' and farewells to
        'farewell_agent'. "
    )

    "Handle only weather, greetings, and farewells.",

    tools=[get_weather_stateful],

    sub_agents=[greeting_agent, farewell_agent],

    output_key="last_weather_report",

    before_model_callback=block_keyword_guardrail, # Keep model guardrail

    before_tool_callback=block_paris_tool_guardrail # <<< Add tool
guardrail

)

print(f"✓ Root Agent '{root_agent_tool_guardrail.name}' created with BOTH
callbacks.")

# --- Create Runner, Using SAME Stateful Session Service ---
```

```
if 'session_service_stateful' in globals():

    runner_root_tool_guardrail = Runner(
        agent=root_agent_tool_guardrail,
        app_name=APP_NAME,
        session_service=session_service_stateful # <<< Use the service from
Step 4/5

    )

    print(f"✓ Runner created for tool guardrail agent
'{runner_root_tool_guardrail.agent.name}', using stateful session service.")

else:

    print("✗ Cannot create runner. 'session_service_stateful' from Step
4/5 is missing.")
```

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

3. Interact to Test the Tool Guardrail

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

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

```
# @title 3. Interact to Test the Tool Argument Guardrail
```

```
import asyncio # Ensure asyncio is imported  
  
# Ensure the runner for the tool guardrail agent is available  
  
if 'runner_root_tool_guardrail' in globals() and runner_root_tool_guardrail:
```

```
# Define the main async function for the tool guardrail test conversation.

# The 'await' keywords INSIDE this function are necessary for async
operations.

async def run_tool_guardrail_test():

    print("\n--- Testing Tool Argument Guardrail ('Paris' blocked) ---")

    # Use the runner for the agent with both callbacks and the existing
stateful session

    # Define a helper lambda for cleaner interaction calls

    interaction_func = lambda query: call_agent_async(query,
runner_root_tool_guardrail,
                                         USER_ID_STATEFUL, #
Use existing user ID
```

```
SESSION_ID_STATEFUL #  
Use existing session ID
```

```
)
```

```
# 1. Allowed city (Should pass both callbacks, use Fahrenheit state)
```

```
    print("--- Turn 1: Requesting weather in New York (expect allowed)  
---")
```

```
    await interaction_func("What's the weather in New York?")
```

```
# 2. Blocked city (Should pass model callback, but be blocked by tool  
callback)
```

```
    print("\n--- Turn 2: Requesting weather in Paris (expect blocked by  
tool guardrail) ---")
```

```
    await interaction_func("How about Paris?") # Tool callback should  
intercept this
```

```
# 3. Another allowed city (Should work normally again)
```

```
    print("\n--- Turn 3: Requesting weather in London (expect allowed)
```

```
---")
```

```
    await interaction_func("Tell me the weather in London.")
```

```
# --- Execute the `run_tool_guardrail_test` async function ---
```

```
# Choose ONE of the methods below based on your environment.
```

```
# METHOD 1: Direct await (Default for Notebooks/Async REPLs)
```

```
# If your environment supports top-level await (like Colab/Jupyter notebooks),
```

```
# it means an event loop is already running, so you can directly await the function.
```

```
print("Attempting execution using 'await' (default for notebooks)..." )
```

```
    await run_tool_guardrail_test()
```

```
# METHOD 2: asyncio.run (For Standard Python Scripts [.py])
```

```
# If running this code as a standard Python script from your terminal,
```

```
# the script context is synchronous. `asyncio.run()` is needed to
```

```
# create and manage an event loop to execute your async function.
```

```
# To use this method:
```

```
# 1. Comment out the `await run_tool_guardrail_test()` line above.
```

```
# 2. Uncomment the following block:
```

```
    """
```

```
import asyncio
```

```
if __name__ == "__main__": # Ensures this runs only when script is executed
    directly

        print("Executing using 'asyncio.run()' (for standard Python
scripts)...")

    try:

        # This creates an event loop, runs your async function, and closes
the loop.

        asyncio.run(run_tool_guardrail_test())

    except Exception as e:

        print(f"An error occurred: {e}")

    """
    # --- Inspect final session state after the conversation ---

    # This block runs after either execution method completes.
```

```
# Optional: Check state for the tool block trigger flag

    print("\n--- Inspecting Final Session State (After Tool Guardrail Test)
---")

# Use the session service instance associated with this stateful session

    final_session = await
session_service_stateful.get_session(app_name=APP_NAME,

user_id=USER_ID_STATEFUL,

                                         session_id=
SESSION_ID_STATEFUL)

if final_session:

    # Use .get() for safer access

        print(f"Tool Guardrail Triggered Flag:
{final_session.state.get('guardrail_tool_block_triggered', 'Not Set (or
False)')}"")
```

```

        print(f"Last Weather Report:
{final_session.state.get('last_weather_report', 'Not Set')}") # Should be
London weather if successful

        print(f"Temperature Unit:
{final_session.state.get('user_preference_temperature_unit', 'Not Set')}") #
Should be Fahrenheit

# print(f"Full State Dict: {final_session.state}") # For detailed view

else:

    print("\n⚠️ Error: Could not retrieve final session state.")

else:

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

```

Analyze the output:

- New York:** The `before_model_callback` allows the request. The LLM `requests get_weather_stateful`. The `before_tool_callback` runs, inspects the args (`{'city': 'New York'}`), sees it's not "Paris", prints "Allowing tool..."

and returns `None`. The actual `get_weather_stateful` function executes, reads "Fahrenheit" from state, and returns the weather report. The agent relays this, and it gets saved via `output_key`.

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

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

Conclusion: Your Agent Team is Ready!

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

Let's recap what you've accomplished:

- You started with a **fundamental agent** equipped with a single tool (`get_weather`).
- You explored ADK's **multi-model flexibility** using LiteLLM, running the same core logic with different LLMs like Gemini, GPT-4o, and Claude.

- You embraced **modularity** by creating specialized sub-agents (`greeting_agent`, `farewell_agent`) and enabling **automatic delegation** from a root agent.
- You gave your agents **memory** using **Session State**, allowing them to remember user preferences (`temperature_unit`) and past interactions (`output_key`).
- You implemented crucial **safety guardrails** using both `before_model_callback` (blocking specific input keywords) and `before_tool_callback` (blocking tool execution based on arguments like the city "Paris").

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

Key Takeaways:

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

Where to Go Next?

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

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

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

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

Happy building!

LLM Agent

The `LlmAgent` (often aliased simply as `Agent`) is a core component in ADK, acting as the "thinking" part of your application. It leverages the power of a Large Language Model (LLM) for reasoning, understanding natural language, making decisions, generating responses, and interacting with tools.

Unlike deterministic [Workflow Agents](#) that follow predefined execution paths, `LlmAgent` behavior is non-deterministic. It uses the LLM to interpret instructions and context, deciding dynamically how to proceed, which tools to use (if any), or whether to transfer control to another agent.

Building an effective `LlmAgent` involves defining its identity, clearly guiding its behavior through instructions, and equipping it with the necessary tools and capabilities.

Defining the Agent's Identity and Purpose

First, you need to establish what the agent *is* and what it's *for*.

- **`name` (Required)**: Every agent needs a unique string identifier. This `name` is crucial for internal operations, especially in multi-agent systems where agents need to refer to or delegate tasks to each other. Choose a descriptive name that reflects the agent's function (e.g., `customer_support_router`, `billing_inquiry_agent`). Avoid reserved names like `user`.
- **`description` (Optional, Recommended for Multi-Agent)**: Provide a concise summary of the agent's capabilities. This description is primarily used by *other* LLM agents to determine if they should route a task to this agent. Make it specific enough to differentiate it from peers (e.g., "Handles inquiries about current billing statements," not just "Billing agent").
- **`model` (Required)**: Specify the underlying LLM that will power this agent's reasoning. This is a string identifier like `"gemini-2.0-flash"`. The choice of model impacts the agent's capabilities, cost, and performance. See the [Models](#) page for available options and considerations.

Python

Java

```
# Example: Defining the basic identity
```

```
capital_agent = LlmAgent(
```

```
    model="gemini-2.0-flash",
```

```
    name="capital_agent",  
  
    description="Answers user questions about the capital city of a given  
country."  
  
    # instruction and tools will be added next  
  
)
```

Guiding the Agent: Instructions (instruction)

The `instruction` parameter is arguably the most critical for shaping an `LlmAgent`'s behavior. It's a string (or a function returning a string) that tells the agent:

- Its core task or goal.
- Its personality or persona (e.g., "You are a helpful assistant," "You are a witty pirate").
- Constraints on its behavior (e.g., "Only answer questions about X," "Never reveal Y").
- How and when to use its `tools`. You should explain the purpose of each tool and the circumstances under which it should be called, supplementing any descriptions within the tool itself.
- The desired format for its output (e.g., "Respond in JSON," "Provide a bulleted list").

Tips for Effective Instructions:

- **Be Clear and Specific:** Avoid ambiguity. Clearly state the desired actions and outcomes.

- **Use Markdown:** Improve readability for complex instructions using headings, lists, etc.
- **Provide Examples (Few-Shot):** For complex tasks or specific output formats, include examples directly in the instruction.
- **Guide Tool Use:** Don't just list tools; explain *when* and *why* the agent should use them.

State:

- The instruction is a string template, you can use the `{var}` syntax to insert dynamic values into the instruction.
- `{var}` is used to insert the value of the state variable named var.
- `{artifact.var}` is used to insert the text content of the artifact named var.
- If the state variable or artifact does not exist, the agent will raise an error. If you want to ignore the error, you can append a ? to the variable name as in `{var?}`.

Python

Java

```
# Example: Adding instructions
```

```
capital_agent = LlmAgent(
```

```
    model="gemini-2.0-flash",
```

```
    name="capital_agent",
```

```
    description="Answers user questions about the capital city of a given
country.",
```

```
    instruction="""You are an agent that provides the capital city of a
country.}}
```

```
When a user asks for the capital of a country:
```

1. Identify the country name from the user's query.
2. Use the `get_capital_city` tool to find the capital.
3. Respond clearly to the user, stating the capital city.

```
Example Query: "What's the capital of {country}?"
```

```
Example Response: "The capital of France is Paris."
```

```
    """,
```

```
    # tools will be added next
```

```
)
```

(Note: For instructions that apply to all agents in a system, consider using `global_instruction` on the root agent, detailed further in the [Multi-Agents section](#).)

Equipping the Agent: Tools (tools)

Tools give your `LlmAgent` capabilities beyond the LLM's built-in knowledge or reasoning. They allow the agent to interact with the outside world, perform calculations, fetch real-time data, or execute specific actions.

- **tools (Optional):** Provide a list of tools the agent can use. Each item in the list can be:
 - A native function or method (wrapped as a `FunctionTool`). Python ADK automatically wraps the native function into a `FunctionTool` whereas, you must explicitly wrap your Java methods using `FunctionTool.create(...)`
 - An instance of a class inheriting from `BaseTool`.
 - An instance of another agent (`AgentTool`, enabling agent-to-agent delegation - see [Multi-Agents](#)).

The LLM uses the function/tool names, descriptions (from docstrings or the `description` field), and parameter schemas to decide which tool to call based on the conversation and its instructions.

Python

Java

```
# Define a tool function

def get_capital_city(country: str) -> str:

    """Retrieves the capital city for a given country."""

# Replace with actual logic (e.g., API call, database lookup)

capitals = {"france": "Paris", "japan": "Tokyo", "canada": "Ottawa"}


return capitals.get(country.lower(), f"Sorry, I don't know the capital of {country}.")
```

```
# Add the tool to the agent

capital_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="capital_agent",
    description="Answers user questions about the capital city of a given
country.",
    instruction="""You are an agent that provides the capital city of a
country... (previous instruction text)""",
    tools=[get_capital_city] # Provide the function directly
)
```

Learn more about Tools in the [Tools](#) section.

Advanced Configuration & Control

Beyond the core parameters, `LlmAgent` offers several options for finer control:

Fine-Tuning LLM Generation (`generate_content_config`)

You can adjust how the underlying LLM generates responses using `generate_content_config`.

- **`generate_content_config` (Optional):** Pass an instance of `google.genai.types.GenerateContentConfig` to control parameters like `temperature` (randomness), `max_output_tokens` (response length), `top_p`, `top_k`, and safety settings.

Python

Java

```
from google.genai import types

agent = LlmAgent(
    # ... other params
    generate_content_config=types.GenerateContentConfig(
        temperature=0.2, # More deterministic output
    )
)
```

```
max_output_tokens=250
```

```
)
```

```
)
```

Structuring Data (`input_schema`, `output_schema`, `output_key`)

For scenarios requiring structured data exchange with an `LLM Agent`, the ADK provides mechanisms to define expected input and desired output formats using schema definitions.

- **`input_schema (Optional)`:** Define a schema representing the expected input structure. If set, the user message content passed to this agent *must* be a JSON string conforming to this schema. Your instructions should guide the user or preceding agent accordingly.
- **`output_schema (Optional)`:** Define a schema representing the desired output structure. If set, the agent's final response *must* be a JSON string conforming to this schema.
 - **Constraint:** Using `output_schema` enables controlled generation within the LLM but **disables the agent's ability to use tools or transfer control to other agents**. Your instructions must guide the LLM to produce JSON matching the schema directly.
- **`output_key (Optional)`:** Provide a string key. If set, the text content of the agent's *final* response will be automatically saved to the session's state dictionary under this key. This is useful for passing results between agents or steps in a workflow.
 - In Python, this might look like: `session.state[output_key] = agent_response_text`
 - In Java: `session.state().put(outputKey, agentResponseText)`

Python

Java

The input and output schema is typically a Pydantic BaseModel.

```
from pydantic import BaseModel, Field

class CapitalOutput(BaseModel):
    capital: str = Field(description="The capital of the country.")

structured_capital_agent = LlmAgent(
    # ... name, model, description

    instruction="""You are a Capital Information Agent. Given a country,
    respond ONLY with a JSON object containing the capital. Format: {"capital": "
    "capital_name"}""",
```

```
        output_schema=CapitalOutput, # Enforce JSON output

        output_key="found_capital" # Store result in state['found_capital']

    # Cannot use tools=[get_capital_city] effectively here

)
```

Managing Context (`include_contents`)

Control whether the agent receives the prior conversation history.

- **`include_contents` (Optional, Default: `'default'`):** Determines if the contents (history) are sent to the LLM.
 - `'default'`: The agent receives the relevant conversation history.
 - `'none'`: The agent receives no prior contents. It operates based solely on its current instruction and any input provided in the *current* turn (useful for stateless tasks or enforcing specific contexts).

Python

Java

```
stateless_agent = LlmAgent(
```

```
# ... other params  
  
    include_contents='none'  
  
)
```

Planning & Code Execution



For more complex reasoning involving multiple steps or executing code:

- **planner (Optional):** Assign a `BasePlanner` instance to enable multi-step reasoning and planning before execution. (See [Multi-Agents](#) patterns).
- **code_executor (Optional):** Provide a `BaseCodeExecutor` instance to allow the agent to execute code blocks (e.g., Python) found in the LLM's response. ([See Tools/Built-in tools](#)).

Putting It Together: Example

Code

(This example demonstrates the core concepts. More complex agents might incorporate schemas, context control, planning, etc.)

Related Concepts (Deferred Topics)

While this page covers the core configuration of `LlmAgent`, several related concepts provide more advanced control and are detailed elsewhere:

- **Callbacks:** Intercepting execution points (before/after model calls, before/after tool calls) using `before_model_callback`, `after_model_callback`, etc. See [Callbacks](#).
- **Multi-Agent Control:** Advanced strategies for agent interaction, including planning (`planner`), controlling agent transfer (`disallow_transfer_to_parent`, `disallow_transfer_to_peers`), and system-wide instructions (`global_instruction`). See [Multi-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](#), 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](#)

- **Loop Agents**

- **Repeatedly** executes its sub-agents until a specific termination condition is met.
[Learn more](#)
- **Parallel Agents**

- Executes multiple sub-agents in **parallel**.
[Learn more](#)

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 LLM Agent instances. This allows you to combine structured process control with flexible, LLM-powered task execution.

Sequential agents

The SequentialAgent

The `SequentialAgent` is a [workflow agent](#) 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

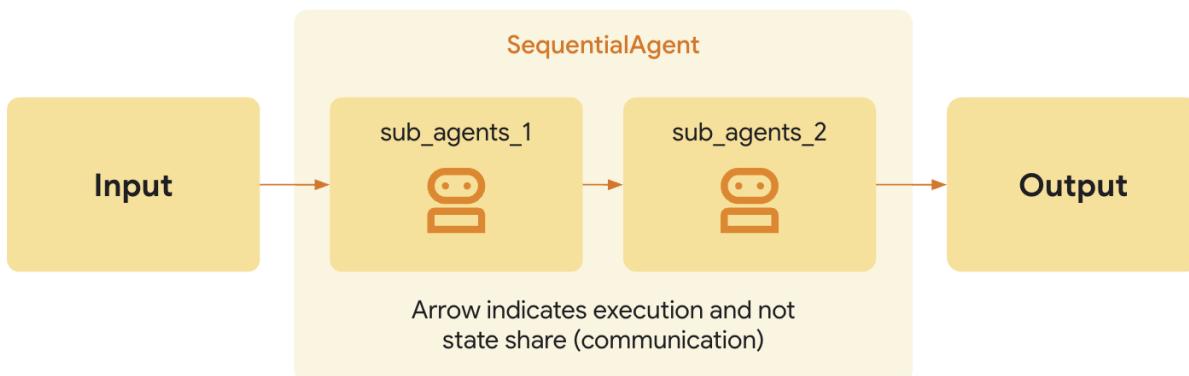
- You want to build an agent that can summarize any webpage, using two tools: `Get Page Contents` and `Summarize Page`. Because the agent must always call `Get Page Contents` before calling `Summarize Page` (you can't summarize from nothing!), you should build your agent using a `SequentialAgent`.

As with other [workflow agents](#), 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

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

Consider a simplified code development pipeline:

- **Code Writer Agent:** An LLM Agent that generates initial code based on a specification.
- **Code Reviewer Agent:** An LLM Agent 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 LLM Agent 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.**

Code

Python

Java

```
# 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,
    max_tokens=1000,
    temperature=0.0,
    top_p=1.0,
    top_k=50,
    stop_sequences=[".", "\n"],

    # 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
```

[Previous](#)
[Workflow Agents](#)

[Next](#)
[Loop agents](#)

[Material for MkDocs](#)

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

- 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`.

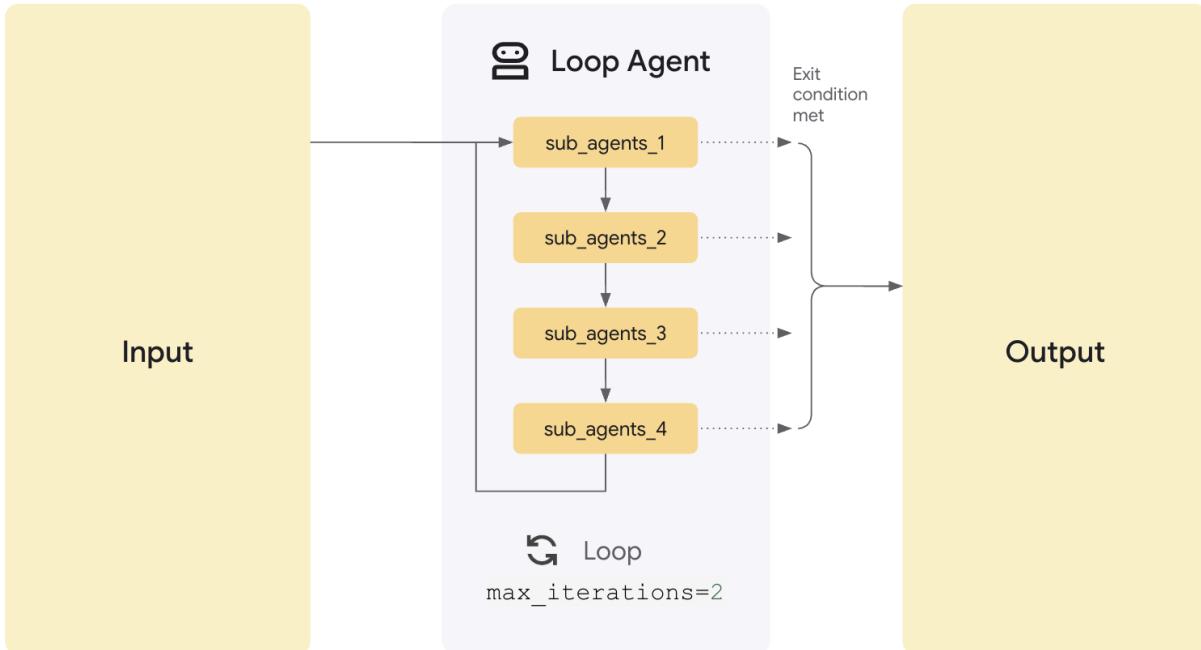
As with other [workflow agents](#), the `LoopAgent` is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are only concerned only with their execution (i.e. in a loop), and not their internal logic; the tools or sub-agents of a workflow agent may or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

How it Works

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

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

Python

Java

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

```
import asyncio
```

```
import os
```

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

```
from google.genai import types
```

```
from google.adk.runners import InMemoryRunner
```

```
from google.adk.agents.invocation_context import InvocationContext
```

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

```
from typing import AsyncGenerator, Optional
```

```
from google.adk.events import Event, EventActions
```

```
# --- 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."  
  
)
```

[Material for MkDocs](#)

Parallel agents

The `ParallelAgent` is a [workflow agent](#) 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](#), 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

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

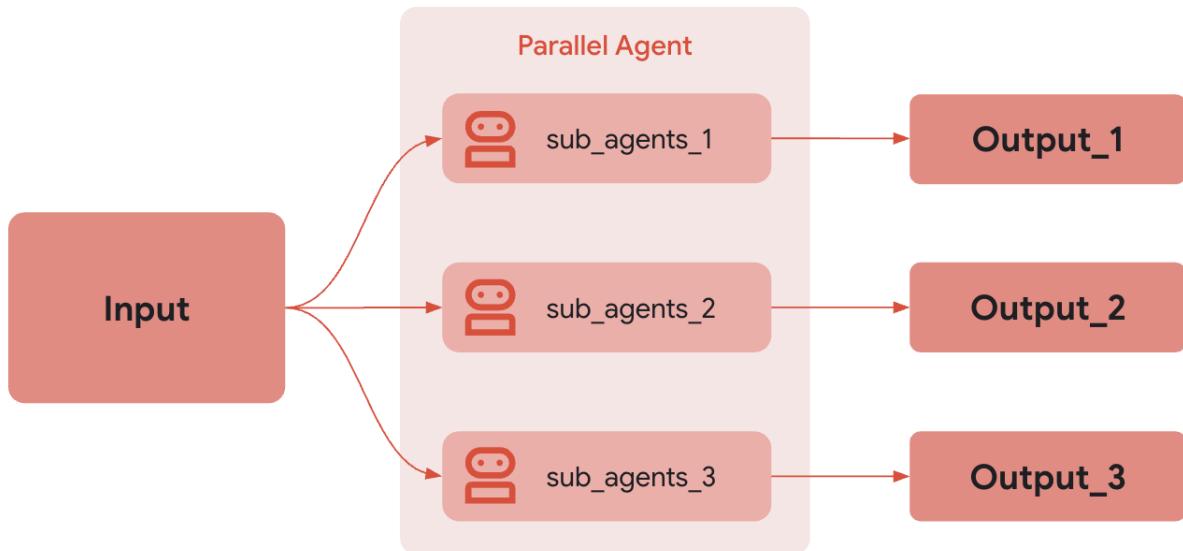
When the `ParallelAgent`'s `run_async()` method is called:

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

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

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])
```

4.

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.

Full Code

[Python](#)

[Java](#)

```
# 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(  
    description="A parallel agent that runs multiple researchers concurrently.",  
    tools=[  
        {"name": "researcher_1", "function": "researcher"},  
        {"name": "researcher_2", "function": "researcher"},  
        {"name": "researcher_3", "function": "researcher"}  
    ],  
    # Store result in state for the merger agent  
    output_key="carbon_capture_result")
```

```
        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
```

[Previous](#)
[Loop agents](#)

[Next](#)
[Custom agents](#)

[Material for MkDocs](#)

Advanced Concept

Building custom agents by directly implementing `_run_async_impl` (or its equivalent in other languages) 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

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

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.

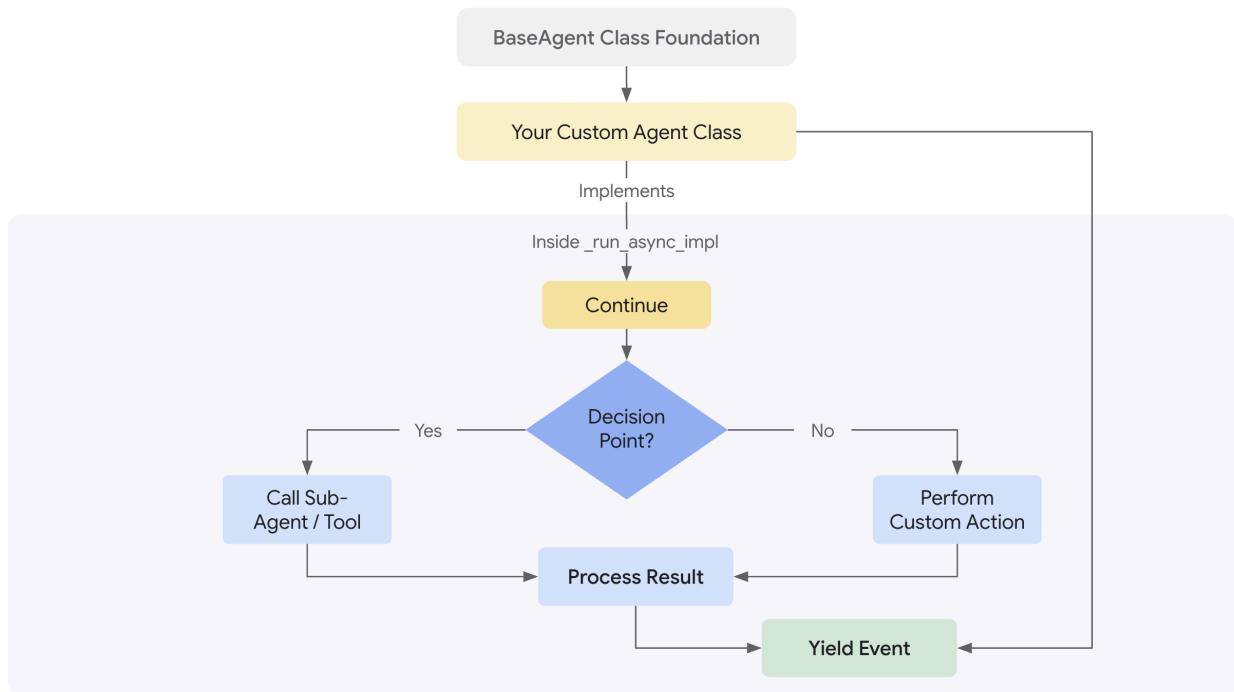
Note

The specific method name for implementing an agent's core asynchronous logic may vary slightly by SDK language (e.g., `runAsyncImpl` in Java, `_run_async_impl` in Python). Refer to the language-specific API documentation for details.

Why Use Them?

While the standard [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 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:

The core of any custom agent is the method where you define its unique asynchronous behavior. This method allows you to orchestrate sub-agents and manage the flow of execution.

Python

Java

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 the Core Asynchronous Method:

Python

Java

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):
```

2.

```
# Optionally inspect or log the event
```

3.

```
yield event # Pass the event up
```

4.

5. **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
```

6.

```
previous_result = ctx.session.state.get("some_key")
```

7.

8.

```
# Make a decision based on state
```

9.

```
if previous_result == "some_value":
```

10.

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

11.

```
else:
```

12.

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

13.

14.

```
# Store a result for a later step (often done via a sub-agent's output_key)
```

15.

```
# ctx.session.state["my_custom_result"] = "calculated_value"
```

16.

17. **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

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 constructor and store them as instance fields/attributes (e.g.,
`this.story_generator = story_generator_instance` or `self.story_generator = story_generator_instance`). This makes them accessible within the custom agent's core asynchronous execution logic (such as: `_run_async_impl` method).
- **Sub Agents List:** When initializing the `BaseAgent` using its `super()` constructor, 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 core execution logic (`_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

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 logic (`if tone == "negative": ...`) within the orchestrator.

Part 1: Simplified custom agent Initialization

[Python](#)

[Java](#)

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

Python

Java

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 `grammarSuggestions` 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

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.

Python

Java

```
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="grammarSuggestions",
```

```
)
```

```
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

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

Python

Java

```
# --- 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

Storyflow Agent

Python

Java

```
# 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="grammarSuggestions",  
  
)  
  
toneCheck = 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")
```

[Previous](#)
[Parallel agents](#)

[Next](#)
[Multi-agent systems](#)

[Material for MkDocs](#)

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](#))
- **Workflow Agents:** Specialized agents (`SequentialAgent`, `ParallelAgent`, `LoopAgent`) designed to manage the execution flow of their sub-agents. (See [Workflow Agents](#))
- **Custom agents:** Your own agents inheriting from `BaseAgent` with specialized, non-LLM logic. (See [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.

1. ADK Primitives for Agent Composition

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

Note

The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., `sub_agents` in Python, `subAgents` in Java). Refer to the language-specific API documentation for details.

1.1. 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.
- **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](#) 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)`.

Python

Java

```
# 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
```

1.2. 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**: Executes its `sub_agents` one after another in the order they are listed.
 - **Context**: Passes the *same* `InvocationContext` sequentially, allowing agents to easily pass results via shared state.

Python

Java

```
# 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**: 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).

Python

Java

```
# 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**: Executes its `sub_agents` sequentially in a loop.
- **Termination**: The loop stops if the optional `max_iterations` is reached, or if any sub-agent returns an `Event` with `escalate=True` in its Event Actions.
- **Context & State**: Passes the *same* `InvocationContext` in each iteration, allowing state changes (e.g., counters, flags) to persist across loops.

Python

Java

```
# 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.

```

1.3. 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`)

The most fundamental way for agents operating within the same invocation (and thus sharing the same `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`.

- **Convenience:** The `output_key` property on `LlmAgent` 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](#)

Python

Java

```
# 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)

Leverages an `LlmAgent`'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.

Python

Java

```
# 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`)

Allows an `LlmAgent` to treat another `BaseAgent` instance as a callable function or `Tool`.

- **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).

Python

Java

```
# 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.

2. 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

- **Structure:** A central `LlmAgent` (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`).

[Python](#)

[Java](#)

```
# 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

- **Structure:** A `SequentialAgent` 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`.

Python

Java

```
# 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

- **Structure:** A `ParallelAgent` 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.

Python

Java

```
# 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

- **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 subagents. Results are returned up the hierarchy (via tool responses or state).

Python

Java

```
# 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)

- **Structure:** Typically involves two agents within a `SequentialAgent`: 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.

[Python](#)

[Java](#)

```
# 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

- **Structure:** Uses a `LoopAgent` 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 `escalate=True` in the `Event Actions` when the result is satisfactory.

Python

Java

```
# 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

- **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.

Python

Java

```
# 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

Note

Java ADK currently supports Gemini and Anthropic models. More model support coming soon.

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](#) 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

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](#)
- [Vertex AI: Gemini Live API](#)

Google AI Studio

- **Use Case:** Google AI Studio is the easiest way to get started with Gemini. All you need is the [API key](#). Best for rapid prototyping and development.
- **Setup:** Typically requires an API key:
 - Set as an environment variable or
 - Passed during the model initialization via the `Client` (see example below)

```
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](#).

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
```

-
- Configure these variables either as environment variables or by providing them directly when initializing the Model.
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](#).

Example:

Python

Java

```
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 Anthropic models



You can integrate Anthropic's Claude models directly using their API key or from a Vertex AI backend into your Java ADK applications by using the ADK's `Claude` wrapper class.

For Vertex AI backend, see the [Third-Party Models on Vertex AI](#) section.

Prerequisites:

1. Dependencies:

- **Anthropic SDK Classes (Transitive):** The Java ADK's `com.google.adk.models.Claude` wrapper relies on classes from Anthropic's official Java SDK. These are typically included as **transitive dependencies**.

2. Anthropic API Key:

- Obtain an API key from Anthropic. Securely manage this key using a secret manager.

Integration:

Instantiate `com.google.adk.models.Claude`, providing the desired Claude model name and an `AnthropicOkHttpClient` configured with your API key. Then, pass this `Claude` instance to your `LlmAgent`.

Example:

```
import com.anthropic.client.AnthropicClient;
```

```
import com.google.adk.agents.LlmAgent;
```

```
import com.google.adk.models.Claude;
```

```
import com.anthropic.client.okhttp.AnthropicOkHttpClient; // From Anthropic's
SDK
```

```
public class DirectAnthropicAgent {
```

```
    private static final String CLAUDE_MODEL_ID = "claude-3-7-sonnet-latest"; //  
Or your preferred Claude model
```

```
public static LlmAgent createAgent() {  
  
    // It's recommended to load sensitive keys from a secure config  
  
    AnthropicClient anthropicClient = AnthropicOkHttpClient.builder()  
  
        .apiKey("ANTHROPIC_API_KEY")  
  
        .build();  
  
    Claude claudeModel = new Claude(  
  
        CLAUDE_MODEL_ID,  
  
        anthropicClient
```

```
    );  
  
    return LlmAgent.builder()  
  
        .name("claude_direct_agent")  
  
        .model(claudeModel)  
  
        .instruction("You are a helpful AI assistant powered by Anthropic  
Claude.")  
  
        // ... other LlmAgent configurations  
  
    .build();  
  
}  
  
public static void main(String[] args) {
```

```
try {

    LlmAgent agent = createAgent();

    System.out.println("Successfully created direct Anthropic agent: " +
agent.name());

} catch (IllegalStateException e) {

    System.err.println("Error creating agent: " + e.getMessage());

}

}
```

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` acts as a translation layer, providing a standardized, OpenAI-compatible interface to over 100+ LLMs.

Setup:

1. Install LiteLLM:

```
pip install litellm
```

2.

3. 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](#) 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(  
    # ... other agent parameters  
)
```

```
    model=LiteLlm(model="anthropic/clause-3-haiku-20240307"),  
    •  
  
    name="clause_direct_agent",  
    •  
  
    instruction="You are an assistant powered by Claude Haiku.",  
    •  
  
    # ... other agent parameters  
    •  
  
)
```

Note for Windows users

Avoiding LiteLLM UnicodeDecodeError on Windows

When using ADK agents with LiteLlm on Windows, users might encounter the following error:

```
UnicodeDecodeError: 'charmap' codec can't decode byte...
```

This issue occurs because `litellm` (used by LiteLlm) reads cached files (e.g., model pricing information) using the default Windows encoding (`cp1252`) instead of UTF-8. Windows users can prevent this issue by setting the `PYTHONUTF8` environment variable to `1`. This forces Python to use UTF-8 globally. **Example (PowerShell):**

```
# Set for current session
```

```
$env:PYTHONUTF8 = "1"
```

```
# Set persistently for the user
```

```
[System.Environment]::SetEnvironmentVariable('PYTHONUTF8', '1',
[System.EnvironmentVariableTarget]::User)
```

Applying this setting ensures that Python reads cached files using UTF-8, avoiding the decoding error.

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

[Ollama](#) allows you to easily run open-source models locally.

Model choice

If your agent is relying on tools, please make sure that you select a model with tool support from [Ollama website](#).

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

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

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

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)



Tools such as [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
```

3.

```
from google.adk.agents import LlmAgent
```

4.

```
from google.adk.models.lite_llm import LiteLlm
```

5.

6.

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

7.

8.

```
# Endpoint URL provided by your vLLM deployment
```

9.

```
api_base_url = "https://your-vllm-endpoint.run.app/v1"
```

10.

11.

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

12.

```
model_name_at_endpoint = "hosted_vllm/google/gemma-3-4b-bit" # Example from  
vllm_test.py
```

13.

14.

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

15.

```
# Adapt this based on your endpoint's security
```

16.

```
try:
```

17.

```
    gcloud_token = subprocess.check_output(
```

18.

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

19.

```
    ).decode().strip()
```

20.

```
    auth_headers = {"Authorization": f"Bearer {gcloud_token}"}
```

21.

```
except Exception as e:
```

22.

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

23.

```
auth_headers = None # Or handle error appropriately
```

24.

25.

```
agent_vllm = LlmAgent(
```

26.

```
    model=LiteLlm(
```

27.

```
        model=model_name_at_endpoint,
```

28.

```
        api_base=api_base_url,
```

29.

```
        # Pass authentication headers if needed
```

30.

```
        extra_headers=auth_headers
```

31.

```
# Alternatively, if endpoint uses an API key:
```

32.

```
# api_key="YOUR_ENDPOINT_API_KEY"
```

33.

```
),
```

34.

```
name="vllm_agent",
```

35.

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

36.

```
# ... other agent parameters
```

37.

```
)
```

38.

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
```

- 2.

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

```
export GOOGLE_CLOUD_PROJECT="YOUR_PROJECT_ID"
```

- 4.

```
export GOOGLE_CLOUD_LOCATION="YOUR_VERTEX_AI_LOCATION" # e.g., us-central1
```

- 5.

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

```
export GOOGLE_GENAI_USE_VERTEXAI=TRUE
```

7.

Model Garden Deployments



You can deploy various open and proprietary models from the [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



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)

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

[Python](#)

[Java](#)

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]"
```

- 3.
4. **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
```

- 5.

```
from google.adk.models.anthropic_llm import Claude
```

- 6.

```
from google.adk.models.registry import LLMRegistry
```

- 7.

- 8.

```
LLMRegistry.register(Claude)
```

- 9.

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

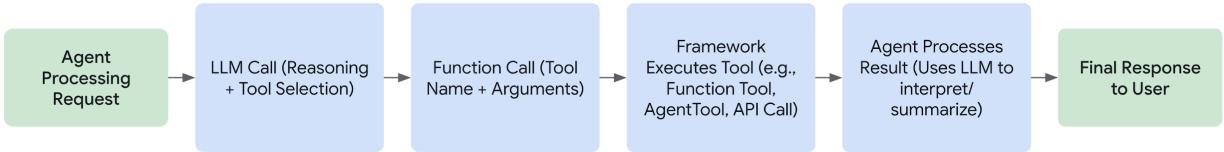
)
```

Tools

What is a Tool?

In the context of ADK, a Tool represents a specific capability provided to an AI agent, enabling it to perform actions and interact with the world beyond its core text generation and reasoning abilities. What distinguishes capable agents from basic language models is often their effective use of tools.

Technically, a tool is typically a modular code component—**like a Python/ Java function**, a class method, or even another specialized agent—designed to execute a distinct, predefined task. These tasks often involve interacting with external systems or data.



Key Characteristics

Action-Oriented: Tools perform specific actions, such as:

- Querying databases
- Making API requests (e.g., fetching weather data, booking systems)
- Searching the web
- Executing code snippets
- Retrieving information from documents (RAG)
- Interacting with other software or services

Extends Agent capabilities: They empower agents to access real-time information, affect external systems, and overcome the knowledge limitations inherent in their training data.

Execute predefined logic: Crucially, tools execute specific, developer-defined logic. They do not possess their own independent reasoning capabilities like the agent's core Large Language Model (LLM). The LLM reasons about which tool to use, when, and with what inputs, but the tool itself just executes its designated function.

How Agents Use Tools

Agents leverage tools dynamically through mechanisms often involving function calling. The process generally follows these steps:

1. **Reasoning:** The agent's LLM analyzes its system instruction, conversation history, and user request.
2. **Selection:** Based on the analysis, the LLM decides on which tool, if any, to execute, based on the tools available to the agent and the docstrings that describes each tool.

3. **Invocation:** The LLM generates the required arguments (inputs) for the selected tool and triggers its execution.
4. **Observation:** The agent receives the output (result) returned by the tool.
5. **Finalization:** The agent incorporates the tool's output into its ongoing reasoning process to formulate the next response, decide the subsequent step, or determine if the goal has been achieved.

Think of the tools as a specialized toolkit that the agent's intelligent core (the LLM) can access and utilize as needed to accomplish complex tasks.

Tool Types in ADK

ADK offers flexibility by supporting several types of tools:

1. **Function Tools:** Tools created by you, tailored to your specific application's needs.
 - **Functions/Methods:** Define standard synchronous functions or methods in your code (e.g., Python def).
 - **Agents-as-Tools:** Use another, potentially specialized, agent as a tool for a parent agent.
 - **Long Running Function Tools:** Support for tools that perform asynchronous operations or take significant time to complete.
2. **Built-in Tools:** Ready-to-use tools provided by the framework for common tasks. Examples: Google Search, Code Execution, Retrieval-Augmented Generation (RAG).
3. **Third-Party Tools:** Integrate tools seamlessly from popular external libraries. Examples: LangChain Tools, CrewAI Tools.

Navigate to the respective documentation pages linked above for detailed information and examples for each tool type.

Referencing Tool in Agent's Instructions

Within an agent's instructions, you can directly reference a tool by using its **function name**. If the tool's **function name** and **docstring** are sufficiently descriptive, your instructions can primarily focus on **when the Large Language Model (LLM) should utilize the tool**. This promotes clarity and helps the model understand the intended use of each tool.

It is **crucial to clearly instruct the agent on how to handle different return values** that a tool might produce. For example, if a tool returns an error message, your instructions should specify whether the agent should retry the operation, give up on the task, or request additional information from the user.

Furthermore, ADK supports the sequential use of tools, where the output of one tool can serve as the input for another. When implementing such workflows, it's important to **describe the intended sequence of tool usage** within the agent's instructions to guide the model through the necessary steps.

Example

The following example showcases how an agent can use tools by **referencing their function names in its instructions**. It also demonstrates how to guide the agent to **handle different return values from tools**, such as success or error messages, and how to orchestrate the **sequential use of multiple tools** to accomplish a task.

Python

Java

```
from google.adk.agents import Agent

from google.adk.tools import FunctionTool

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService
```

```
from google.genai import types
```

```
APP_NAME="weather_sentiment_agent"
```

```
USER_ID="user1234"
```

```
SESSION_ID="1234"
```

```
MODEL_ID="gemini-2.0-flash"
```

```
# Tool 1
```

```
def get_weather_report(city: str) -> dict:
```

```
    """Retrieves the current weather report for a specified city.
```

```
Returns:
```

```
    dict: A dictionary containing the weather information with a 'status' key ('success' or 'error') and a 'report' key with the weather details if successful, or an 'error_message' if an error occurred.
```

```
"""
```

```
if city.lower() == "london":
```

```
    return {"status": "success", "report": "The current weather in London is cloudy with a temperature of 18 degrees Celsius and a chance of rain."}
```

```
elif city.lower() == "paris":
```

```
    return {"status": "success", "report": "The weather in Paris is sunny with a temperature of 25 degrees Celsius."}
```

```
else:
```

```
    return {"status": "error", "error_message": f"Weather information for '{city}' is not available."}
```

```
weather_tool = FunctionTool(func=get_weather_report)
```

```
# Tool 2
```

```
def analyze_sentiment(text: str) -> dict:
```

```
    """Analyzes the sentiment of the given text.
```

```
Returns:
```

```
    dict: A dictionary with 'sentiment' ('positive', 'negative', or  
'neutral') and a 'confidence' score.
```

```
    """
```

```
    if "good" in text.lower() or "sunny" in text.lower():
```

```
        return {"sentiment": "positive", "confidence": 0.8}
```

```
    elif "rain" in text.lower() or "bad" in text.lower():
```

```
        return {"sentiment": "negative", "confidence": 0.7}
```

```
    else:
```

```
        return {"sentiment": "neutral", "confidence": 0.6}
```

```
sentiment_tool = FunctionTool(func=analyze_sentiment)
```

```
# Agent
```

```
weather_sentiment_agent = Agent(
```

```
    model=MODEL_ID,
```

```
    name='weather_sentiment_agent',
```

```
instruction="""You are a helpful assistant that provides weather information and analyzes the sentiment of user feedback.
```

```
**If the user asks about the weather in a specific city, use the 'get_weather_report' tool to retrieve the weather details.**
```

```
**If the 'get_weather_report' tool returns a 'success' status, provide the weather report to the user.**
```

```
**If the 'get_weather_report' tool returns an 'error' status, inform the user that the weather information for the specified city is not available and ask if they have another city in mind.**
```

```
**After providing a weather report, if the user gives feedback on the weather (e.g., 'That's good' or 'I don't like rain'), use the 'analyze_sentiment' tool to understand their sentiment.** Then, briefly acknowledge their sentiment.
```

```
You can handle these tasks sequentially if needed."",
```

```
    tools=[weather_tool, sentiment_tool]
```

```
)
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()

session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
                                         session_id=SESSION_ID)

runner = Runner(agent=weather_sentiment_agent, app_name=APP_NAME,
                session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID,
                         new_message=content)

    for event in events:
```

```
if event.is_final_response():

    final_response = event.content.parts[0].text

    print("Agent Response: ", final_response)

call_agent("weather in london?")
```

Tool Context

For more advanced scenarios, ADK allows you to access additional contextual information within your tool function by including the special parameter `tool_context: ToolContext`. By including this in the function signature, ADK will **automatically** provide an **instance of the ToolContext class** when your tool is called during agent execution.

The **ToolContext** provides access to several key pieces of information and control levers:

- `state: State`: Read and modify the current session's state. Changes made here are tracked and persisted.
- `actions: EventActions`: Influence the agent's subsequent actions after the tool runs (e.g., skip summarization, transfer to another agent).
- `function_call_id: str`: The unique identifier assigned by the framework to this specific invocation of the tool. Useful for tracking and correlating with authentication responses. This can also be helpful when multiple tools are called within a single model response.

- `function_call_event_id`: str: This attribute provides the unique identifier of the **event** that triggered the current tool call. This can be useful for tracking and logging purposes.
- `auth_response`: Any: Contains the authentication response/credentials if an authentication flow was completed before this tool call.
- Access to Services: Methods to interact with configured services like Artifacts and Memory.

Note that you shouldn't include the `tool_context` parameter in the tool function docstring. Since `ToolContext` is automatically injected by the ADK framework *after* the LLM decides to call the tool function, it is not relevant for the LLM's decision-making and including it can confuse the LLM.

State Management

The `tool_context.state` attribute provides direct read and write access to the state associated with the current session. It behaves like a dictionary but ensures that any modifications are tracked as deltas and persisted by the session service. This enables tools to maintain and share information across different interactions and agent steps.

- **Reading State:** Use standard dictionary access (`tool_context.state['my_key']`) or the `.get()` method (`tool_context.state.get('my_key', default_value)`).
- **Writing State:** Assign values directly (`tool_context.state['new_key'] = 'new_value'`). These changes are recorded in the `state_delta` of the resulting event.
- **State Prefixes:** Remember the standard state prefixes:
 - `app :*`: Shared across all users of the application.
 - `user :*`: Specific to the current user across all their sessions.
 - (No prefix): Specific to the current session.
 - `temp :*`: Temporary, not persisted across invocations (useful for passing data within a single run call but generally less useful inside a tool context which operates between LLM calls).

Python

Java

```
from google.adk.tools import ToolContext, FunctionTool
```

```
def update_user_preference(preference: str, value: str, tool_context: ToolContext):

    """Updates a user-specific preference."""

    user_prefs_key = "user:preferences"

    # Get current preferences or initialize if none exist

    preferences = tool_context.state.get(user_prefs_key, {})

    preferences[preference] = value

    # Write the updated dictionary back to the state

    tool_context.state[user_prefs_key] = preferences

    print(f"Tool: Updated user preference '{preference}' to '{value}'")

    return {"status": "success", "updated_preference": preference}
```

```
pref_tool = FunctionTool(func=update_user_preference)

# In an Agent:

# my_agent = Agent(..., tools=[pref_tool])

# When the LLM calls update_user_preference(preference='theme', value='dark',
...):

# The tool_context.state will be updated, and the change will be part of the

# resulting tool response event's actions.state_delta.
```

Controlling Agent Flow

The `tool_context.actions` attribute (`ToolContext.actions()` in Java) holds an **EventActions** object. Modifying attributes on this object allows your tool to influence what the agent or framework does after the tool finishes execution.

- `skip_summarization: bool`: (Default: False) If set to True, instructs the ADK to bypass the LLM call that typically summarizes the tool's output. This is useful if your tool's return value is already a user-ready message.
- `transfer_to_agent: str`: Set this to the name of another agent. The framework will halt the current agent's execution and **transfer control of the conversation to the specified agent**. This allows tools to dynamically hand off tasks to more specialized agents.
- `escalate: bool`: (Default: False) Setting this to True signals that the current agent cannot handle the request and should pass control up to its parent agent (if in a hierarchy). In a LoopAgent, setting `escalate=True` in a sub-agent's tool will terminate the loop.

Example

Python

Java

```
from google.adk.agents import Agent

from google.adk.tools import FunctionTool

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools import ToolContext

from google.genai import types
```

```
APP_NAME="customer_support_agent"
```

```
USER_ID="user1234"
```

```
SESSION_ID="1234"
```

```
def check_and_transfer(query: str, tool_context: ToolContext) -> str:  
  
    """Checks if the query requires escalation and transfers to another agent  
    if needed."""  
  
    if "urgent" in query.lower():  
  
        print("Tool: Detected urgency, transferring to the support agent.")  
  
        tool_context.actions.transfer_to_agent = "support_agent"  
  
    return "Transferring to the support agent..."  
  
else:
```

```
        return f"Processed query: '{query}'. No further action needed."  
  
escalation_tool = FunctionTool(func=check_and_transfer)  
  
main_agent = Agent(  
    model='gemini-2.0-flash',  
    name='main_agent',  
    instruction="""You are the first point of contact for customer support of  
an analytics tool. Answer general queries. If the user indicates urgency, use  
the 'check_and_transfer' tool.""" ,  
    tools=[check_and_transfer]  
)
```

```
support_agent = Agent(  
  
    model='gemini-2.0-flash',  
  
    name='support_agent',  
  
    instruction="""You are the dedicated support agent. Mentioned you are a  
support handler and please help the user with their urgent issue."""
```

```
)
```

```
main_agent.sub_agents = [support_agent]
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()
```

```
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,  
session_id=SESSION_ID)
```

```
runner = Runner(agent=main_agent, app_name=APP_NAME,
session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    for event in events:

        if event.is_final_response():

            final_response = event.content.parts[0].text
```

```
    print("Agent Response: ", final_response)
```

```
call_agent("this is urgent, i cant login")
```

Explanation

- We define two agents: `main_agent` and `support_agent`. The `main_agent` is designed to be the initial point of contact.
- The `check_and_transfer` tool, when called by `main_agent`, examines the user's query.
- If the query contains the word "urgent", the tool accesses the `tool_context`, specifically `tool_context.actions`, and sets the `transfer_to_agent` attribute to `support_agent`.
- This action signals to the framework to **transfer the control of the conversation to the agent named support_agent**.
- When the `main_agent` processes the urgent query, the `check_and_transfer` tool triggers the transfer. The subsequent response would ideally come from the `support_agent`.
- For a normal query without urgency, the tool simply processes it without triggering a transfer.

This example illustrates how a tool, through EventActions in its ToolContext, can dynamically influence the flow of the conversation by transferring control to another specialized agent.

Authentication



ToolContext provides mechanisms for tools interacting with authenticated APIs. If your tool needs to handle authentication, you might use the following:

- `auth_response`: Contains credentials (e.g., a token) if authentication was already handled by the framework before your tool was called (common with RestApiTool and OpenAPI security schemes).
- `request_credential(auth_config: dict)`: Call this method if your tool determines authentication is needed but credentials aren't available. This signals the framework to start an authentication flow based on the provided `auth_config`.
- `get_auth_response()`: Call this in a subsequent invocation (after `request_credential` was successfully handled) to retrieve the credentials the user provided.

For detailed explanations of authentication flows, configuration, and examples, please refer to the dedicated Tool Authentication documentation page.

Context-Aware Data Access Methods

These methods provide convenient ways for your tool to interact with persistent data associated with the session or user, managed by configured services.

- `list_artifacts()` (or `listArtifacts()` in Java): Returns a list of filenames (or keys) for all artifacts currently stored for the session via the `artifact_service`. Artifacts are typically files (images, documents, etc.) uploaded by the user or generated by tools/agents.
- `load_artifact(filename: str)`: Retrieves a specific artifact by its filename from the `artifact_service`. You can optionally specify a version; if omitted, the latest version is returned. Returns a `google.genai.types.Part` object containing the artifact data and mime type, or `None` if not found.
- `save_artifact(filename: str, artifact: types.Part)`: Saves a new version of an artifact to the `artifact_service`. Returns the new version number (starting from 0).



- `search_memory(query: str)`
Queries the user's long-term memory using the configured `memory_service`. This is useful for retrieving relevant information from past interactions or stored knowledge. The structure of the `SearchMemoryResponse` depends

on the specific memory service implementation but typically contains relevant text snippets or conversation excerpts.

Example

Python

Java

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
#      http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
from google.adk.tools import ToolContext, FunctionTool
```

```
from google.genai import types
```

```
def process_document(
```

```
    document_name: str, analysis_query: str, tool_context: ToolContext
```

```
) -> dict:
```

```
"""Analyzes a document using context from memory."""

# 1. Load the artifact

print(f"Tool: Attempting to load artifact: {document_name}")

document_part = tool_context.load_artifact(document_name)

if not document_part:

    return {"status": "error", "message": f"Document '{document_name}' not
found."}

document_text = document_part.text # Assuming it's text for simplicity

print(f"Tool: Loaded document '{document_name}' ({len(document_text)} chars.)")
```

```
# 2. Search memory for related context

    print(f"Tool: Searching memory for context related to: '{analysis_query}'")

    memory_response = tool_context.search_memory(
        f"Context for analyzing document about {analysis_query}"
    )

    memory_context = "\n".join([
        m.events[0].content.parts[0].text
        for m in memory_response.memories
        if m.events and m.events[0].content
    ])
```

```
    ) # Simplified extraction

    print(f"Tool: Found memory context: {memory_context[:100]}...")

# 3. Perform analysis (placeholder)

    analysis_result = f"Analysis of '{document_name}' regarding
'{analysis_query}' using memory context: [Placeholder Analysis Result]"

    print("Tool: Performed analysis.")

# 4. Save the analysis result as a new artifact

    analysis_part = types.Part.from_text(text=analysis_result)

    new_artifact_name = f"analysis_{document_name}"

    version = await tool_context.save_artifact(new_artifact_name,
analysis_part)
```

```
    print(f"Tool: Saved analysis result as '{new_artifact_name}' version  
{version}.")  
  
    return {  
  
        "status": "success",  
  
        "analysis_artifact": new_artifact_name,  
  
        "version": version,  
  
    }  
  
doc_analysis_tool = FunctionTool(func=process_document)
```

```
# In an Agent:  
  
# Assume artifact 'report.txt' was previously saved.  
  
# Assume memory service is configured and has relevant past data.  
  
# my_agent = Agent(..., tools=[doc_analysis_tool], artifact_service=...,  
memory_service=...)
```

By leveraging the **ToolContext**, developers can create more sophisticated and context-aware custom tools that seamlessly integrate with ADK's architecture and enhance the overall capabilities of their agents.

Defining Effective Tool Functions

When using a method or function as an ADK Tool, how you define it significantly impacts the agent's ability to use it correctly. The agent's Large Language Model (LLM) relies heavily on the function's **name**, **parameters (arguments)**, **type hints**, and **docstring / source code comments** to understand its purpose and generate the correct call.

Here are key guidelines for defining effective tool functions:

- **Function Name:**
 - Use descriptive, verb-noun based names that clearly indicate the action (e.g., `get_weather`, `searchDocuments`, `schedule_meeting`).
 - Avoid generic names like `run`, `process`, `handle_data`, or overly ambiguous names like `doStuff`. Even with a good description, a name like `do_stuff` might confuse the model about when to use the tool versus, for example, `cancelFlight`.

- The LLM uses the function name as a primary identifier during tool selection.
- **Parameters (Arguments):**
 - Your function can have any number of parameters.
 - Use clear and descriptive names (e.g., `city` instead of `c`, `search_query` instead of `q`).
 - **Provide type hints in Python** for all parameters (e.g., `city: str, user_id: int, items: list[str]`). This is essential for ADK to generate the correct schema for the LLM.
 - Ensure all parameter types are **JSON serializable**. All Java primitives as well as standard Python types like `str, int, float, bool, list, dict`, and their combinations are generally safe. Avoid complex custom class instances as direct parameters unless they have a clear JSON representation.
 - **Do not set default values** for parameters. E.g., `def my_func(param1: str = "default")`. Default values are not reliably supported or used by the underlying models during function call generation. All necessary information should be derived by the LLM from the context or explicitly requested if missing.
 - **`self / cls Handled Automatically`:** Implicit parameters like `self` (for instance methods) or `cls` (for class methods) are automatically handled by ADK and excluded from the schema shown to the LLM. You only need to define type hints and descriptions for the logical parameters your tool requires the LLM to provide.
- **Return Type:**
 - The function's return value **must be a dictionary (`dict`)** in Python or a **Map** in Java.
 - If your function returns a non-dictionary type (e.g., a string, number, list), the ADK framework will automatically wrap it into a dictionary/Map like `{'result': your_original_return_value}` before passing the result back to the model.
 - Design the dictionary/Map keys and values to be **descriptive and easily understood by the LLM**. Remember, the model reads this output to decide its next step.
 - Include meaningful keys. For example, instead of returning just an error code like `500`, return `{'status': 'error', 'error_message': 'Database connection failed'}`.

- It's a **highly recommended practice** to include a `status` key (e.g., `'success'`, `'error'`, `'pending'`, `'ambiguous'`) to clearly indicate the outcome of the tool execution for the model.
- **Docstring / Source Code Comments:**
 - **This is critical.** The docstring is the primary source of descriptive information for the LLM.
 - **Clearly state what the tool does.** Be specific about its purpose and limitations.
 - **Explain when the tool should be used.** Provide context or example scenarios to guide the LLM's decision-making.
 - **Describe each parameter clearly.** Explain what information the LLM needs to provide for that argument.
 - Describe the **structure and meaning of the expected dict return value**, especially the different `status` values and associated data keys.
 - **Do not describe the injected ToolContext parameter.** Avoid mentioning the optional `tool_context: ToolContext` parameter within the docstring description since it is not a parameter the LLM needs to know about. `ToolContext` is injected by ADK, *after* the LLM decides to call it.
- **Example of a good definition:**

Python

Java

```
def lookup_order_status(order_id: str) -> dict:
```

"""Fetches the current status of a customer's order using its ID.

Use this tool ONLY when a user explicitly asks for the status of

a specific order and provides the order ID. Do not use it for

```
general inquiries.
```

Args:

```
order_id: The unique identifier of the order to look up.
```

Returns:

```
A dictionary containing the order status.
```

```
Possible statuses: 'shipped', 'processing', 'pending', 'error'.
```

```
Example success: {'status': 'shipped', 'tracking_number': '1Z9...'}  
Example error: {'status': 'error', 'error_message': 'Order ID not found.'}
```

...

```

# ... function implementation to fetch status ...

if status := fetch_status_from_backend(order_id):

    return {"status": status.state, "tracking_number": status.tracking} #

Example structure

else:

    return {"status": "error", "error_message": f"Order ID {order_id} not
found."}

```

- **Simplicity and Focus:**

- **Keep Tools Focused:** Each tool should ideally perform one well-defined task.
- **Fewer Parameters are Better:** Models generally handle tools with fewer, clearly defined parameters more reliably than those with many optional or complex ones.
- **Use Simple Data Types:** Prefer basic types (`str`, `int`, `bool`, `float`, `List[str]`, or `int`, `byte`, `short`, `long`, `float`, `double`, `boolean` and `char` in **Java**) over complex custom classes or deeply nested structures as parameters when possible.
- **Decompose Complex Tasks:** Break down functions that perform multiple distinct logical steps into smaller, more focused tools. For instance, instead of a single `update_user_profile(profile: ProfileObject)` tool, consider separate tools like `update_user_name(name: str)`, `update_user_address(address: str)`, `update_user_preferences(preferences: list[str])`, etc. This makes it easier for the LLM to select and use the correct capability.

By adhering to these guidelines, you provide the LLM with the clarity and structure it needs to effectively utilize your custom function tools, leading to more capable and reliable agent behavior.

Toolsets: Grouping and Dynamically

Providing Tools



Beyond individual tools, ADK introduces the concept of a **Toolset** via the `BaseToolset` interface (defined in `google.adk.tools.base_toolset`). A toolset allows you to manage and provide a collection of `BaseTool` instances, often dynamically, to an agent.

This approach is beneficial for:

- **Organizing Related Tools:** Grouping tools that serve a common purpose (e.g., all tools for mathematical operations, or all tools interacting with a specific API).
- **Dynamic Tool Availability:** Enabling an agent to have different tools available based on the current context (e.g., user permissions, session state, or other runtime conditions). The `get_tools` method of a toolset can decide which tools to expose.
- **Integrating External Tool Providers:** Toolsets can act as adapters for tools coming from external systems, like an OpenAPI specification or an MCP server, converting them into ADK-compatible `BaseTool` objects.

The `BaseToolset` Interface

Any class acting as a toolset in ADK should implement the `BaseToolset` abstract base class. This interface primarily defines two methods:

- `async def get_tools(...) -> list[BaseTool]`: This is the core method of a toolset. When an ADK agent needs to know its available tools, it will call `get_tools()` on each `BaseToolset` instance provided in its `tools` list.

- It receives an optional `readonly_context` (an instance of `ReadonlyContext`). This context provides read-only access to information like the current session state (`readonly_context.state`), agent name, and invocation ID. The toolset can use this context to dynamically decide which tools to return.
- It **must** return a list of `BaseTool` instances (e.g., `FunctionTool`, `RestApiTool`).
- `async def close(self) -> None`: This asynchronous method is called by the ADK framework when the toolset is no longer needed, for example, when an agent server is shutting down or the `Runner` is being closed. Implement this method to perform any necessary cleanup, such as closing network connections, releasing file handles, or cleaning up other resources managed by the toolset.

Using Toolsets with Agents

You can include instances of your `BaseToolset` implementations directly in an `LlmAgent`'s `tools` list, alongside individual `BaseTool` instances.

When the agent initializes or needs to determine its available capabilities, the ADK framework will iterate through the `tools` list:

- If an item is a `BaseTool` instance, it's used directly.
- If an item is a `BaseToolset` instance, its `get_tools()` method is called (with the current `ReadonlyContext`), and the returned list of `BaseTools` is added to the agent's available tools.

Example: A Simple Math Toolset

Let's create a basic example of a toolset that provides simple arithmetic operations.

```
# 1. Define the individual tool functions
```

```
def add_numbers(a: int, b: int, tool_context: ToolContext) -> Dict[str, Any]:
```

```
    """Adds two integer numbers.
```

```
Args:
```

```
    a: The first number.
```

```
    b: The second number.
```

```
Returns:
```

```
    A dictionary with the sum, e.g., {'status': 'success', 'result': 5}
```

```
    """
```

```
    print(f"Tool: add_numbers called with a={a}, b={b}")
```

```
    result = a + b
```

```
    # Example: Storing something in tool_context state
```

```
    tool_context.state["last_math_operation"] = "addition"
```

```
    return {"status": "success", "result": result}
```

```
def subtract_numbers(a: int, b: int) -> Dict[str, Any]:
```

```
    """Subtracts the second number from the first.
```

```
Args:
```

```
    a: The first number.
```

```
    b: The second number.
```

```
Returns:
```

```
    A dictionary with the difference, e.g., {'status': 'success', 'result':  
1}
```

```
    """
```

```
    print(f"Tool: subtract_numbers called with a={a}, b={b}")
```

```
    return {"status": "success", "result": a - b}

# 2. Create the Toolset by implementing BaseToolset

class SimpleMathToolset(BaseToolset):

    def __init__(self, prefix: str = "math_"):

        self.prefix = prefix

        # Create FunctionTool instances once

        self._add_tool = FunctionTool(
            func=add_numbers,
            name=f"{self.prefix}add_numbers", # Toolset can customize names
```

```
    )

    self._subtract_tool = FunctionTool(
        func=subtract_numbers, name=f"{self.prefix}subtract_numbers"
    )

)

print(f"SimpleMathToolset initialized with prefix '{self.prefix}'")

async def get_tools(
    self, readonly_context: Optional[ReadonlyContext] = None
) -> List[BaseTool]:
    print(f"SimpleMathToolset.get_tools() called.")

    # Example of dynamic behavior:

    # Could use readonly_context.state to decide which tools to return
```

```
# For instance, if readonly_context.state.get("enable_advanced_math"):

#     return [self._add_tool, self._subtract_tool, self._multiply_tool]

# For this simple example, always return both tools

tools_to_return = [self._add_tool, self._subtract_tool]

print(f"SimpleMathToolset providing tools: {[t.name for t in
tools_to_return]}")

return tools_to_return

async def close(self) -> None:

    # No resources to clean up in this simple example

    print(f"SimpleMathToolset.close() called for prefix '{self.prefix}'.")
```

```
    await asyncio.sleep(0) # Placeholder for async cleanup if needed

# 3. Define an individual tool (not part of the toolset)

def greet_user(name: str = "User") -> Dict[str, str]:
    """Greets the user."""

    print(f"Tool: greet_user called with name={name}")

    return {"greeting": f"Hello, {name}!"}

greet_tool = FunctionTool(func=greet_user)
```

```
# 4. Instantiate the toolset

math_toolset_instance = SimpleMathToolset(prefix="calculator_")

# 5. Define an agent that uses both the individual tool and the toolset

calculator_agent = LlmAgent(
    name="CalculatorAgent",
    model="gemini-2.0-flash", # Replace with your desired model
    instruction="You are a helpful calculator and greeter. "
    "Use 'greet_user' for greetings. "
    "Use 'calculator_add_numbers' to add and 'calculator_subtract_numbers' to
    subtract. "
```

```
"Announce the state of 'last_math_operation' if it's set.",  
  
    tools=[greet_tool, math_toolset_instance], # Individual tool # Toolset  
instance  
  
)
```

In this example:

- `SimpleMathToolset` implements `BaseToolset` and its `get_tools()` method returns `FunctionTool` instances for `add_numbers` and `subtract_numbers`. It also customizes their names using a prefix.
- The `calculator_agent` is configured with both an individual `greet_tool` and an instance of `SimpleMathToolset`.
- When `calculator_agent` is run, ADK will call `math_toolset_instance.get_tools()`. The agent's LLM will then have access to `greet_user`, `calculator_add_numbers`, and `calculator_subtract_numbers` to handle user requests.
- The `add_numbers` tool demonstrates writing to `tool_context.state`, and the agent's instruction mentions reading this state.
- The `close()` method is called to ensure any resources held by the toolset are released.

Toolsets offer a powerful way to organize, manage, and dynamically provide collections of tools to your ADK agents, leading to more modular, maintainable, and adaptable agentic applications.

Function tools

What are function tools?

When out-of-the-box tools don't fully meet specific requirements, developers can create custom function tools. This allows for **tailored functionality**, such as connecting to proprietary databases or implementing unique algorithms.

For example, a function tool, "myfinancetool", might be a function that calculates a specific financial metric. ADK also supports long running functions, so if that calculation takes a while, the agent can continue working on other tasks.

ADK offers several ways to create functions tools, each suited to different levels of complexity and control:

1. Function Tool
2. Long Running Function Tool
3. Agents-as-a-Tool

1. Function Tool

Transforming a function into a tool is a straightforward way to integrate custom logic into your agents. In fact, when you assign a function to an agent's tools list, the framework will automatically wrap it as a Function Tool for you. This approach offers flexibility and quick integration.

Parameters

Define your function parameters using standard **JSON-serializable types** (e.g., string, integer, list, dictionary). It's important to avoid setting default values for parameters, as the language model (LLM) does not currently support interpreting them.

Return Type

The preferred return type for a Function Tool is a **dictionary** in Python or **Map** in Java. This allows you to structure the response with key-value pairs, providing context and clarity to the LLM. If your function returns a type other than a dictionary, the framework automatically wraps it into a dictionary with a single key named "**result**".

Strive to make your return values as descriptive as possible. *For example*, instead of returning a numeric error code, return a dictionary with an "error_message" key containing a human-readable explanation. **Remember that the LLM**, not a piece of code, needs to understand the result. As a best practice, include a "status" key in your return dictionary to indicate the overall outcome (e.g., "success", "error", "pending"), providing the LLM with a clear signal about the operation's state.

Docstring / Source code comments

The docstring (or comments above) your function serve as the tool's description and is sent to the LLM. Therefore, a well-written and comprehensive docstring is crucial for the LLM to understand how to use the tool effectively. Clearly explain the purpose of the function, the meaning of its parameters, and the expected return values.

Example

Best Practices

While you have considerable flexibility in defining your function, remember that simplicity enhances usability for the LLM. Consider these guidelines:

- **Fewer Parameters are Better:** Minimize the number of parameters to reduce complexity.
- **Simple Data Types:** Favor primitive data types like `str` and `int` over custom classes whenever possible.
- **Meaningful Names:** The function's name and parameter names significantly influence how the LLM interprets and utilizes the tool. Choose names that clearly reflect the function's purpose and the meaning of its inputs. Avoid generic names like `do_stuff()` or `beAgent()`.

2. Long Running Function Tool

Designed for tasks that require a significant amount of processing time without blocking the agent's execution. This tool is a subclass of `FunctionTool`.

When using a `LongRunningFunctionTool`, your function can initiate the long-running operation and optionally return an **initial result**** (e.g. the long-running operation id). Once a long running function tool is invoked the agent runner will pause the agent run and let the agent client to decide whether to continue or wait until the long-running operation finishes. The agent client can query the progress of the long-running operation and send back an intermediate or final response. The agent can then continue with other tasks. An example is the human-in-the-loop scenario where the agent needs human approval before proceeding with a task.

How it Works

In Python, you wrap a function with `LongRunningFunctionTool`. In Java, you pass a Method name to `LongRunningFunctionTool.create()`.

1. **Initiation:** When the LLM calls the tool, your function starts the long-running operation.
2. **Initial Updates:** Your function should optionally return an initial result (e.g. the long-running operation id). The ADK framework takes the result and sends it back to the LLM packaged within a `FunctionResponse`. This allows the LLM to inform the user (e.g., status, percentage complete, messages). And then the agent run is ended / paused.
3. **Continue or Wait:** After each agent run is completed. Agent client can query the progress of the long-running operation and decide whether to continue the agent run with an intermediate response (to update the progress) or wait until a final response is retrieved. Agent client should send the intermediate or final response back to the agent for the next run.
4. **Framework Handling:** The ADK framework manages the execution. It sends the intermediate or final `FunctionResponse` sent by agent client to the LLM to generate a user friendly message.

Creating the Tool

Define your tool function and wrap it using the `LongRunningFunctionTool` class:

Python

Java

```
# 1. Define the long running function
```

```
def ask_for_approval(  
    purpose: str, amount: float  
) -> dict[str, Any]:  
  
    """Ask for approval for the reimbursement."""  
  
    # create a ticket for the approval  
  
    # Send a notification to the approver with the link of the ticket  
  
    return {'status': 'pending', 'approver': 'Sean Zhou', 'purpose' : purpose,  
'amount': amount, 'ticket-id': 'approval-ticket-1' }  
  
def reimburse(purpose: str, amount: float) -> str:  
  
    """Reimburse the amount of money to the employee."""  
  
    # send the reimbursement request to payment vendor
```

```
    return {'status': 'ok'}
```

2. Wrap the function with LongRunningFunctionTool

```
long_running_tool = LongRunningFunctionTool(func=ask_for_approval)
```

Intermediate / Final result Updates

Agent client received an event with long running function calls and check the status of the ticket. Then Agent client can send the intermediate or final response back to update the progress. The framework packages this value (even if it's None) into the content of the `FunctionResponse` sent back to the LLM.

Applies to only Java ADK

When passing `ToolContext` with Function Tools, ensure that one of the following is true:

- The Schema is passed with the `ToolContext` parameter in the function signature, like:

```
@com.google.adk.tools.Annotations.Schema(name = "toolContext") ToolContext  
toolContext
```

-
- OR
- The following `-parameters` flag is set to the mvn compiler plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.14.0</version> <!-- or newer -->
      <configuration>
        <compilerArgs>
```

```
<arg>-parameters</arg>
```

```
</compilerArgs>
```

```
</configuration>
```

```
</plugin>
```

```
</plugins>
```

```
</build>
```

This constraint is temporary and will be removed.

Python

Java

```
# Agent Interaction

async def call_agent(query):

    def get_long_running_function_call(event: Event) -> types.FunctionCall:

        # Get the long running function call from the event

        if not event.long_running_tool_ids or not event.content or not
event.content.parts:

            return

        for part in event.content.parts:

            if (
                part
                and part.function_call
                and event.long_running_tool_ids
```

```
        and part.function_call.id in event.long_running_tool_ids
```

```
):
```

```
    return part.function_call
```

```
def get_function_response(event: Event, function_call_id: str) ->
types.FunctionResponse:
```

```
# Get the function response for the fuction call with specified id.
```

```
if not event.content or not event.content.parts:
```

```
    return
```

```
    for part in event.content.parts:
```

```
        if (
```

```
            part
```

```
        and part.function_response

        and part.function_response.id == function_call_id

    ):

        return part.function_response

content = types.Content(role='user', parts=[types.Part(text=query)])

events = runner.run_async(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

print("\nRunning agent...")

events_async = runner.run_async(
    session_id=session.id, user_id=USER_ID, new_message=content

)
```

```
    long_running_function_call, long_running_function_response, ticket_id =
None, None, None

    async for event in events_async:

        # Use helper to check for the specific auth request event

        if not long_running_function_call:

            long_running_function_call = get_long_running_function_call(event)

        else:

            long_running_function_response = get_function_response(event,
long_running_function_call.id)

        if long_running_function_response:

            ticket_id =
long_running_function_response.response['ticket-id']
```

```
if event.content and event.content.parts:

    if text := ''.join(part.text or '' for part in
event.content.parts):

        print(f'{event.author}: {text}')

if long_running_function_response:

    # query the status of the correpsonding ticket via tciket_id

    # send back an intermediate / final response

    updated_response = long_running_function_response.model_copy(deep=True)

    updated_response.response = {'status': 'approved'}

async for event in runner.run_async()
```

```

        session_id=session.id, user_id=USER_ID,
new_message=types.Content(parts=[types.Part(function_response =
updated_response)], role='user')

):

    if event.content and event.content.parts:

        if text := ''.join(part.text or '' for part in
event.content.parts):

            print(f'{event.author}: {text}')

```

Python complete example: File Processing Simulation

Key aspects of this example

- **LongRunningFunctionTool**: Wraps the supplied method/function; the framework handles sending yielded updates and the final return value as sequential FunctionResponses.
- **Agent instruction**: Directs the LLM to use the tool and understand the incoming FunctionResponse stream (progress vs. completion) for user updates.
- **Final return**: The function returns the final result dictionary, which is sent in the concluding FunctionResponse to indicate completion.

3. Agent-as-a-Tool

This powerful feature allows you to leverage the capabilities of other agents within your system by calling them as tools. The Agent-as-a-Tool enables you to invoke another

agent to perform a specific task, effectively **delegating responsibility**. This is conceptually similar to creating a Python function that calls another agent and uses the agent's response as the function's return value.

Key difference from sub-agents

It's important to distinguish an Agent-as-a-Tool from a Sub-Agent.

- **Agent-as-a-Tool:** When Agent A calls Agent B as a tool (using Agent-as-a-Tool), Agent B's answer is **passed back** to Agent A, which then summarizes the answer and generates a response to the user. Agent A retains control and continues to handle future user input.
- **Sub-agent:** When Agent A calls Agent B as a sub-agent, the responsibility of answering the user is completely **transferred to Agent B**. Agent A is effectively out of the loop. All subsequent user input will be answered by Agent B.

Usage

To use an agent as a tool, wrap the agent with the `AgentTool` class.

Python

Java

```
tools=[AgentTool(agent=agent_b)]
```

Customization

The `AgentTool` class provides the following attributes for customizing its behavior:

- **skip_summarization: bool:** If set to True, the framework will **bypass the LLM-based summarization** of the tool agent's response. This can be useful when the tool's response is already well-formatted and requires no further processing.

Example

How it works

1. When the `main_agent` receives the long text, its instruction tells it to use the 'summarize' tool for long texts.
2. The framework recognizes 'summarize' as an `AgentTool` that wraps the `summary_agent`.
3. Behind the scenes, the `main_agent` will call the `summary_agent` with the long text as input.
4. The `summary_agent` will process the text according to its instruction and generate a summary.
5. **The response from the `summary_agent` is then passed back to the `main_agent`.**
6. The `main_agent` can then take the summary and formulate its final response to the user (e.g., "Here's a summary of the text: ...")

Built-in tools

These built-in tools provide ready-to-use functionality such as Google Search or code executors that provide agents with common capabilities. For instance, an agent that needs to retrieve information from the web can directly use the `google_search` tool without any additional setup.

How to Use

1. **Import:** Import the desired tool from the tools module. This is `agents.tools` in Python or `com.google.adk.tools` in Java.
2. **Configure:** Initialize the tool, providing required parameters if any.
3. **Register:** Add the initialized tool to the `tools` list of your Agent.

Once added to an agent, the agent can decide to use the tool based on the **user prompt** and its **instructions**. The framework handles the execution of the tool when the agent calls it. Important: check the **Limitations** section of this page.

Available Built-in tools

Note: Java only supports Google Search and Code Execution tools currently.

Google Search

The `google_search` tool allows the agent to perform web searches using Google Search. The `google_search` tool is only compatible with Gemini 2 models.

Additional requirements when using the `google_search` tool

When you use grounding with Google Search, and you receive Search suggestions in your response, you must display the Search suggestions in production and in your applications. For more information on grounding with Google Search, see Grounding with Google Search documentation for [Google AI Studio](#) or [Vertex AI](#). The UI code (HTML) is returned in the Gemini response as `renderedContent`, and you will need to show the HTML in your app, in accordance with the policy.

Python

Java

```
from google.adk.agents import Agent
```

```
from google.adk.runners import Runner
```

```
from google.adk.sessions import InMemorySessionService
```

```
from google.adk.tools import google_search
```

```
from google.genai import types
```

```
APP_NAME="google_search_agent"

USER_ID="user1234"

SESSION_ID="1234"

root_agent = Agent(
    name="basic_search_agent",
    model="gemini-2.0-flash",
    description="Agent to answer questions using Google Search.",
    instruction="I can answer your questions by searching the internet. Just ask me anything!",
```

```
# google_search is a pre-built tool which allows the agent to perform
Google searches.
```

```
    tools=[google_search]
```

```
)
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()
```

```
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)
```

```
runner = Runner(agent=root_agent, app_name=APP_NAME,
session_service=session_service)
```

```
# Agent Interaction
```

```
def call_agent(query):

    """Helper function to call the agent with a query.

    """

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    for event in events:

        if event.is_final_response():

            final_response = event.content.parts[0].text

            print("Agent Response: ", final_response)
```

```
call_agent("what's the latest ai news?")
```

Code Execution

The `built_in_code_execution` tool enables the agent to execute code, specifically when using Gemini 2 models. This allows the model to perform tasks like calculations, data manipulation, or running small scripts.

Python

Java

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
#      http://www.apache.org/licenses/LICENSE-2.0
```

```
#  
  
# Unless required by applicable law or agreed to in writing, software  
  
# distributed under the License is distributed on an "AS IS" BASIS,  
  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
  
# See the License for the specific language governing permissions and  
  
# limitations under the License.  
  
import asyncio  
  
from google.adk.agents import LlmAgent  
  
from google.adk.runners import Runner  
  
from google.adk.sessions import InMemorySessionService  
  
from google.adk.code_executors import BuiltInCodeExecutor
```

```
from google.genai import types

AGENT_NAME = "calculator_agent"

APP_NAME = "calculator"

USER_ID = "user1234"

SESSION_ID = "session_code_exec_async"

GEMINI_MODEL = "gemini-2.0-flash"

# Agent Definition

code_agent = LlmAgent(
    name=AGENT_NAME,
```

```
    model=GEMINI_MODEL,  
  
    executor=[BuiltInCodeExecutor],  
  
    instruction="""You are a calculator agent.  
  
When given a mathematical expression, write and execute Python code to  
calculate the result.  
  
Return only the final numerical result as plain text, without markdown or  
code blocks.  
  
""",  
  
    description="Executes Python code to perform calculations.",  
  
)  
  
# Session and Runner  
  
session_service = InMemorySessionService()
```

```
session = session_service.create_session(  
    app_name=APP_NAME, user_id=USER_ID, session_id=SESSION_ID  
)  
  
runner = Runner(agent=code_agent, app_name=APP_NAME,  
session_service=session_service)  
  
# Agent Interaction (Async)  
  
async def call_agent_async(query):  
  
    content = types.Content(role="user", parts=[types.Part(text=query)])  
  
    print(f"\n--- Running Query: {query} ---")  
  
    final_response_text = "No final text response captured."  
  
    try:
```

```
# Use run_async

    async for event in runner.run_async(
        user_id=USER_ID, session_id=SESSION_ID, new_message=content
    ):

        print(f"Event ID: {event.id}, Author: {event.author}")

# --- Check for specific parts FIRST ---

        has_specific_part = False

        if event.content and event.content.parts:

            for part in event.content.parts: # Iterate through all parts

                if part.executable_code:
```

```
# Access the actual code string via .code

        print(

            f"  Debug: Agent generated
code:\n```python\n{part.executable_code.code}\n```"

        )

has_specific_part = True

elif part.code_execution_result:

    # Access outcome and output correctly

    print(

        f"  Debug: Code Execution Result:
{part.code_execution_result.outcome} -
Output:\n{part.code_execution_result.output}"

    )

has_specific_part = True
```

```
        # Also print any text parts found in any event for
debugging

    elif part.text and not part.text.isspace():

        print(f"  Text: '{part.text.strip()}'")

    # Do not set has_specific_part=True here, as we want
the final response logic below

# --- Check for final response AFTER specific parts ---

# Only consider it final if it doesn't have the specific code parts
we just handled

if not has_specific_part and event.is_final_response():

    if (
        event.content

        and event.content.parts
```

```
        and event.content.parts[0].text

    ):

    final_response_text = event.content.parts[0].text.strip()

    print(f"==> Final Agent Response: {final_response_text}")

else:

    print("==> Final Agent Response: [No text content in final
event]")

except Exception as e:

    print(f"ERROR during agent run: {e}")

    print("-" * 30)
```

```
# Main async function to run the examples

async def main():

    await call_agent_async("Calculate the value of (5 + 7) * 3")

    await call_agent_async("What is 10 factorial?")


# Execute the main async function

try:

    asyncio.run(main())


except RuntimeError as e:

    # Handle specific error when running asyncio.run in an already running loop
    # (like Jupyter/Colab)
```

```
if "cannot be called from a running event loop" in str(e):  
  
    print("\nRunning in an existing event loop (like Colab/Jupyter).")  
  
    print("Please run `await main()` in a notebook cell instead.")  
  
# If in an interactive environment like a notebook, you might need to  
run:  
  
    # await main()  
  
else:  
  
    raise e # Re-raise other runtime errors
```

Vertex AI Search

The `vertex_ai_search_tool` uses Google Cloud's Vertex AI Search, enabling the agent to search across your private, configured data stores (e.g., internal documents, company policies, knowledge bases). This built-in tool requires you to provide the specific data store ID during configuration.

```
import asyncio
```

```
from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

from google.adk.tools import VertexAiSearchTool

# Replace with your actual Vertex AI Search Datastore ID

# Format:
projects/<PROJECT_ID>/locations/<LOCATION>/collections/default_collection/data
Stores/<DATASTORE_ID>

# e.g.,
"projects/12345/locations/us-central1/collections/default_collection/dataStore
s/my-datastore-123"

YOUR_DATASTORE_ID = "YOUR_DATASTORE_ID_HERE"
```

```
# Constants

APP_NAME_VSEARCH = "vertex_search_app"

USER_ID_VSEARCH = "user_vsearch_1"

SESSION_ID_VSEARCH = "session_vsearch_1"

AGENT_NAME_VSEARCH = "doc_qa_agent"

GEMINI_2_FLASH = "gemini-2.0-flash"

# Tool Instantiation

# You MUST provide your datastore ID here.

vertex_search_tool = VertexAiSearchTool(data_store_id=YOUR_DATASTORE_ID)
```

```
# Agent Definition

doc_qa_agent = LlmAgent(
    name=AGENT_NAME_VSEARCH,
    model=GEMINI_2_FLASH, # Requires Gemini model
    tools=[vertex_search_tool],
    instruction=f"""You are a helpful assistant that answers questions based on
information found in the document store: {YOUR_DATASTORE_ID}.

    Use the search tool to find relevant information before answering.

    If the answer isn't in the documents, say that you couldn't find the
information.

    """,
    description="Answers questions using a specific Vertex AI Search
datastore."
```

```
)  
  
# Session and Runner Setup  
  
session_service_vsearch = InMemorySessionService()  
  
runner_vsearch = Runner(  
  
    agent=doc_qa_agent, app_name=APP_NAME_VSEARCH,  
    session_service=session_service_vsearch  
  
)  
  
session_vsearch = session_service_vsearch.create_session(  
  
    app_name=APP_NAME_VSEARCH, user_id=USER_ID_VSEARCH,  
    session_id=SESSION_ID_VSEARCH  
  
)
```

```
# Agent Interaction Function

async def call_vsearch_agent_async(query):

    print("\n--- Running Vertex AI Search Agent ---")

    print(f"Query: {query}")

    if "YOUR_DATASTORE_ID_HERE" in YOUR_DATASTORE_ID:

        print("Skipping execution: Please replace YOUR_DATASTORE_ID_HERE with
your actual datastore ID.")

        print("-" * 30)

    return

    content = types.Content(role='user', parts=[types.Part(text=query)])

    final_response_text = "No response received."
```

```
try:

    async for event in runner_vsearch.run_async(
        user_id=USER_ID_VSEARCH, session_id=SESSION_ID_VSEARCH,
        new_message=content
    ):

        # Like Google Search, results are often embedded in the model's
        # response.

        if event.is_final_response() and event.content and
           event.content.parts:
            final_response_text = event.content.parts[0].text.strip()

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

        # You can inspect event.grounding_metadata for source citations

        if event.grounding_metadata:
```

```
        print(f"  (Grounding metadata found with  
{len(event.grounding_metadata.grounding_attributions)} attributions)")

except Exception as e:  
  
    print(f"An error occurred: {e}")  
  
    print("Ensure your datastore ID is correct and the service account has  
permissions.")  
  
    print("-" * 30)  
  
# --- Run Example ---  
  
async def run_vsearch_example():  
  
    # Replace with a question relevant to YOUR datastore content  
  
    await call_vsearch_agent_async("Summarize the main points about the Q2  
strategy document.")
```

```
    await call_vsearch_agent_async("What safety procedures are mentioned for  
lab X?")
```

```
# Execute the example
```

```
# await run_vsearch_example()
```

```
# Running locally due to potential colab asyncio issues with multiple awaits
```

```
try:
```

```
    asyncio.run(run_vsearch_example())
```

```
except RuntimeError as e:
```

```
    if "cannot be called from a running event loop" in str(e):
```

```
    print("Skipping execution in running event loop (like Colab/Jupyter).  
Run locally.")  
  
else:  
  
    raise e
```

Use Built-in tools with other tools

The following code sample demonstrates how to use multiple built-in tools or how to use built-in tools with other tools by using multiple agents:

Python

Java

```
from google.adk.tools import agent_tool  
  
from google.adk.agents import Agent  
  
from google.adk.tools import google_search  
  
from google.adk.code_executors import BuiltInCodeExecutor
```

```
search_agent = Agent(  
  
    model='gemini-2.0-flash',  
  
    name='SearchAgent',  
  
    instruction="""  
  
        You're a specialist in Google Search  
  
        """,  
  
    tools=[google_search],  
  
)  
  
coding_agent = Agent(  
  
    model='gemini-2.0-flash',  
  
    name='CodeAgent',
```

```
    instruction="""  
  
        You're a specialist in Code Execution  
  
        """ ,  
  
    code_executor=[BuiltInCodeExecutor],  
  
)  
  
root_agent = Agent(  
  
    name="RootAgent",  
  
    model="gemini-2.0-flash",  
  
    description="Root Agent",  
  
    tools=[agent_tool.AgentTool(agent=search_agent),  
agent_tool.AgentTool(agent=coding_agent)],  
  
)
```

Limitations

Warning

Currently, for each root agent or single agent, only one built-in tool is supported. No other tools of any type can be used in the same agent.

For example, the following approach that uses ***a built-in tool along with other tools*** within a single agent is **not** currently supported:

Python

Java

```
root_agent = Agent(  
  
    name="RootAgent",  
  
    model="gemini-2.0-flash",  
  
    description="Root Agent",  
  
    tools=[custom_function],  
  
    executor=[BuiltInCodeExecutor] # <-- not supported when used with tools  
)
```

Warning

Built-in tools cannot be used within a sub-agent.

For example, the following approach that uses built-in tools within sub-agents is **not** currently supported:

Python

Java

```
search_agent = Agent(
```

```
    model='gemini-2.0-flash',
```

```
    name='SearchAgent',
```

```
    instruction="""
```

```
        You're a specialist in Google Search
```

```
        """,
```

```
    tools=[google_search],
```

```
)
```

```
coding_agent = Agent(
```

```
    model='gemini-2.0-flash',  
  
    name='CodeAgent',  
  
    instruction="""  
  
        You're a specialist in Code Execution  
  
        """),  
  
    executor=[BuiltInCodeExecutor],  
  
)  
  
root_agent = Agent(  
  
    name="RootAgent",  
  
    model="gemini-2.0-flash",  
  
    description="Root Agent",  
  
    sub_agents=[
```

```
    search_agent,
```

```
    coding_agent
```

```
],
```

```
)
```

Third Party Tools



ADK is designed to be **highly extensible, allowing you to seamlessly integrate tools from other AI Agent frameworks** like CrewAI and LangChain. This interoperability is crucial because it allows for faster development time and allows you to reuse existing tools.

1. Using LangChain Tools

ADK provides the `LangchainTool` wrapper to integrate tools from the LangChain ecosystem into your agents.

Example: Web Search using LangChain's Tavily tool

[Tavily](#) provides a search API that returns answers derived from real-time search results, intended for use by applications like AI agents.

1. Follow [ADK installation and setup](#) guide.
2. **Install Dependencies:** Ensure you have the necessary LangChain packages installed. For example, to use the Tavily search tool, install its specific dependencies:

```
pip install langchain_community tavily-python
```

- 3.
4. Obtain a [Tavily API KEY](#) and export it as an environment variable.

```
export TAVILY_API_KEY=<REPLACE_WITH_API_KEY>
```

- 5.
6. **Import:** Import the `LangchainTool` wrapper from ADK and the specific LangChain tool you wish to use (e.g, `TavilySearchResults`).

```
from google.adk.tools.langchain_tool import LangchainTool
```

- 7.

```
from langchain_community.tools import TavilySearchResults
```

- 8.
9. **Instantiate & Wrap:** Create an instance of your LangChain tool and pass it to the `LangchainTool` constructor.

```
# Instantiate the LangChain tool
```

- 10.

```
tavily_tool_instance = TavilySearchResults(
```

11.

```
max_results=5,
```

12.

```
search_depth="advanced",
```

13.

```
include_answer=True,
```

14.

```
include_raw_content=True,
```

15.

```
include_images=True,
```

16.

)

17.

18.

```
# Wrap it with LangchainTool for ADK
```

19.

```
adk_tavily_tool = LangchainTool(tool=tavily_tool_instance)
```

20.

21. Add to Agent: Include the wrapped `LangchainTool` instance in your agent's `tools` list during definition.

```
from google.adk import Agent
```

22.

23.

```
# Define the ADK agent, including the wrapped tool
```

24.

```
my_agent = Agent(
```

25.

```
    name="langchain_tool_agent",
```

26.

```
    model="gemini-2.0-flash",
```

27.

```
    description="Agent to answer questions using TavilySearch.",
```

28.

```
    instruction="I can answer your questions by searching the internet. Just  
ask me anything!",
```

29.

```
    tools=[adk_tavily_tool] # Add the wrapped tool here
```

30.

```
)
```

31.

Full Example: Tavily Search

Here's the full code combining the steps above to create and run an agent using the LangChain Tavily search tool.

```
import os

from google.adk import Agent, Runner

from google.adk.sessions import InMemorySessionService
```

```
from google.adk.tools.langchain_tool import LangchainTool

from google.genai import types

from langchain_community.tools import TavilySearchResults

# Ensure TAVILY_API_KEY is set in your environment

if not os.getenv("TAVILY_API_KEY"):

    print("Warning: TAVILY_API_KEY environment variable not set.")

APP_NAME = "news_app"

USER_ID = "1234"

SESSION_ID = "session1234"
```

```
# Instantiate LangChain tool

tavily_search = TavilySearchResults(
    max_results=5,
    search_depth="advanced",
    include_answer=True,
    include_raw_content=True,
    include_images=True,
)

# Wrap with LangchainTool

adk_tavily_tool = LangchainTool(tool=tavily_search)
```

```
# Define Agent with the wrapped tool

my_agent = Agent(
    name="langchain_tool_agent",
    model="gemini-2.0-flash",
    description="Agent to answer questions using TavilySearch.",
    instruction="I can answer your questions by searching the internet. Just
ask me anything!",
    tools=[adk_tavily_tool] # Add the wrapped tool here
)

session_service = InMemorySessionService()
```

```
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)

runner = Runner(agent=my_agent, app_name=APP_NAME,
session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    for event in events:

        if event.is_final_response():
```

```
    final_response = event.content.parts[0].text

    print("Agent Response: ", final_response)

call_agent("stock price of GOOG")
```

2. Using CrewAI tools

ADK provides the `CrewaiTool` wrapper to integrate tools from the CrewAI library.

Example: Web Search using CrewAI's Serper API

[Serper API](#) provides access to Google Search results programmatically. It allows applications, like AI agents, to perform real-time Google searches (including news, images, etc.) and get structured data back without needing to scrape web pages directly.

1. Follow [ADK installation and setup](#) guide.
2. **Install Dependencies:** Install the necessary CrewAI tools package. For example, to use the `SerperDevTool`:

```
pip install crewai-tools
```

- 3.
4. Obtain a [Serper API KEY](#) and export it as an environment variable.

```
export SERPER_API_KEY=<REPLACE_WITH_API_KEY>
```

5.

6. **Import:** Import CrewaiTool from ADK and the desired CrewAI tool (e.g, SerperDevTool).

```
from google.adk.tools.crewai_tool import CrewaiTool
```

7.

```
from crewai_tools import SerperDevTool
```

8.

9. **Instantiate & Wrap:** Create an instance of the CrewAI tool. Pass it to the CrewaiTool constructor. **Crucially, you must provide a name and description** to the ADK wrapper, as these are used by ADK's underlying model to understand when to use the tool.

```
# Instantiate the CrewAI tool
```

10.

```
serper_tool_instance = SerperDevTool(
```

11.

```
    n_results=10,
```

12.

```
    save_file=False,
```

13.

```
    search_type="news",
```

14.

```
)
```

15.

16.

```
# Wrap it with CrewaiTool for ADK, providing name and description
```

17.

```
adk_serper_tool = CrewaiTool(
```

18.

```
    name="InternetNewsSearch",
```

19.

```
    description="Searches the internet specifically for recent news articles
using Serper.",
```

20.

```
    tool=serper_tool_instance
```

21.

```
)
```

22.

23. Add to Agent: Include the wrapped `CrewaiTool` instance in your agent's `tools` list.

```
from google.adk import Agent
```

24.

25.

```
# Define the ADK agent
```

26.

```
my_agent = Agent(
```

27.

```
    name="crewai_search_agent",
```

28.

```
    model="gemini-2.0-flash",
```

29.

```
    description="Agent to find recent news using the Serper search tool.",
```

30.

```
    instruction="I can find the latest news for you. What topic are you  
interested in?",
```

31.

```
    tools=[adk_serper_tool] # Add the wrapped tool here
```

32.

```
)
```

33.

Full Example: Serper API

Here's the full code combining the steps above to create and run an agent using the CrewAI Serper API search tool.

```
import os
```

```
from google.adk import Agent, Runner
```

```
from google.adk.sessions import InMemorySessionService
```

```
from google.adk.tools.crewai_tool import CrewaiTool

from google.genai import types

from crewai_tools import SerperDevTool

# Constants

APP_NAME = "news_app"

USER_ID = "user1234"

SESSION_ID = "1234"

# Ensure SERPER_API_KEY is set in your environment
```

```
if not os.getenv("SERPER_API_KEY"):

    print("Warning: SERPER_API_KEY environment variable not set.")

serper_tool_instance = SerperDevTool(
    n_results=10,
    save_file=False,
    search_type="news",
)

adk_serper_tool = CrewaiTool(
    name="InternetNewsSearch",
```

```
    description="Searches the internet specifically for recent news articles
    using Serper.",

    tool=serper_tool_instance

)

serper_agent = Agent(
    name="basic_search_agent",

    model="gemini-2.0-flash",

    description="Agent to answer questions using Google Search.",

    instruction="I can answer your questions by searching the internet. Just
    ask me anything!",

    # Add the Serper tool

    tools=[adk_serper_tool]
```

```
)  
  
# Session and Runner  
  
session_service = InMemorySessionService()  
  
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,  
session_id=SESSION_ID)  
  
runner = Runner(agent=serper_agent, app_name=APP_NAME,  
session_service=session_service)  
  
# Agent Interaction  
  
def call_agent(query):  
  
    content = types.Content(role='user', parts=[types.Part(text=query)])
```

```
events = runner.run(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

for event in events:

    if event.is_final_response():

        final_response = event.content.parts[0].text

        print("Agent Response: ", final_response)

call_agent("what's the latest news on AI Agents?")
```

Google Cloud Tools



Google Cloud tools make it easier to connect your agents to Google Cloud's products and services. With just a few lines of code you can use these tools to connect your agents with:

- **Any custom APIs** that developers host in Apigee.
- **100s of prebuilt connectors** to enterprise systems such as Salesforce, Workday, and SAP.
- **Automation workflows** built using application integration.
- **Databases** such as Spanner, AlloyDB, Postgres and more using the MCP Toolbox for databases.



Apigee API Hub Tools

ApiHubToolset lets you turn any documented API from Apigee API hub into a tool with a few lines of code. This section shows you the step by step instructions including setting up authentication for a secure connection to your APIs.

Prerequisites

1. [Install ADK](#)
2. Install the [Google Cloud CLI](#).
3. [Apigee API hub](#) instance with documented (i.e. OpenAPI spec) APIs
4. Set up your project structure and create required files

```
project_root_folder
```

```
|
```

```
`-- my_agent
```

```
|-- .env
```

```
|-- __init__.py
```

```
|-- agent.py
```

```
`-- tool.py
```

Create an API Hub Toolset

Note: This tutorial includes an agent creation. If you already have an agent, you only need to follow a subset of these steps.

1. Get your access token, so that APIHubToolset can fetch spec from API Hub API. In your terminal run the following command

```
gcloud auth print-access-token
```

- 2.

```
# Prints your access token like 'ya29....'
```

- 3.

4. Ensure that the account used has the required permissions. You can use the pre-defined role `roles/apihub.viewer` or assign the following permissions:
 - a. **apihub.specs.get (required)**
 - b. apihub.apis.get (optional)
 - c. apihub.apis.list (optional)
 - d. apihub.versions.get (optional)
 - e. apihub.versions.list (optional)
 - f. apihub.specs.list (optional)

5. Create a tool with `APIHubToolset`. Add the below to `tools.py`

If your API requires authentication, you must configure authentication for the tool. The following code sample demonstrates how to configure an API key.

ADK supports token based auth (API Key, Bearer token), service account, and OpenID Connect. We will soon add support for various OAuth2 flows.

```
from google.adk.tools.openapi_tool.auth.auth_helpers import  
token_to_scheme_credential
```

6.

```
from google.adk.tools.apihub_tool.apihub_toolset import APIHubToolset
```

7.

8.

```
# Provide authentication for your APIs. Not required if your APIs don't  
required authentication.
```

9.

```
auth_scheme, auth_credential = token_to_scheme_credential(
```

10.

```
    "apikey", "query", "apikey", apikey_credential_str
```

11.

```
)
```

12.

13.

```
sample_toolset_with_auth = APIHubToolset(
```

14.

```
    name="apihub-sample-tool",
```

15.

```
    description="Sample Tool",
```

16.

```
    access_token="...", # Copy your access token generated in step 1
```

17.

```
    apihub_resource_name="...", # API Hub resource name
```

18.

```
    auth_scheme=auth_scheme,
```

19.

```
    auth_credential=auth_credential,
```

20.

)

21.

22. For production deployment we recommend using a service account instead of an access token. In the code snippet above, use

`service_account_json=service_account_cred_json_str` and provide your security account credentials instead of the token.

For `apihub_resource_name`, if you know the specific ID of the OpenAPI Spec being used for your API, use

``projects/my-project-id/locations/us-west1/apis/my-api-id/versions/version-id/specs/spec-id``. If you would like the Toolset to automatically pull the first available spec from the API, use

``projects/my-project-id/locations/us-west1/apis/my-api-id``

23. Create your agent file [Agent.py](#) and add the created tools to your agent definition:

```
from google.adk.agents.llm_agent import LlmAgent
```

24.

```
from .tools import sample_toolset
```

25.

26.

```
root_agent = LlmAgent(
```

27.

```
model='gemini-2.0-flash',
```

28.

```
name='enterprise_assistant',
```

29.

```
instruction='Help user, leverage the tools you have access to',
```

30.

```
tools=sample_toolset.get_tools(),
```

31.

```
)
```

32.

33. Configure your `__init__.py` to expose your agent

```
from . import agent
```

34.

35. Start the Google ADK Web UI and try your agent:

```
# make sure to run `adk web` from your project_root_folder
```

36.

```
adk web
```

37.

Then go to <http://localhost:8000> to try your agent from the Web UI.

Application Integration Tools

With **ApplicationIntegrationToolset** you can seamlessly give your agents a secure and governed to enterprise applications using Integration Connector's 100+ pre-built connectors for systems like Salesforce, ServiceNow, JIRA, SAP, and more. Support for both on-prem and SaaS applications. In addition you can turn your existing Application Integration process automations into agentic workflows by providing application integration workflows as tools to your ADK agents.

Prerequisites

1. [Install ADK](#)
2. An existing [Application Integration](#) workflow or [Integrations Connector](#) connection you want to use with your agent
3. To use tool with default credentials: have Google Cloud CLI installed. See [installation guide](#).

Run:

```
gcloud config set project <project-id>
```

```
gcloud auth application-default login
```

```
gcloud auth application-default set-quota-project <project-id>
```

1. Set up your project structure and create required files

```
project_root_folder
```

2.

```
| -- .env
```

3.

```
`-- my_agent
```

4.

```
| -- __init__.py
```

5.

```
| -- agent.py
```

6.

```
`-- tools.py
```

7.

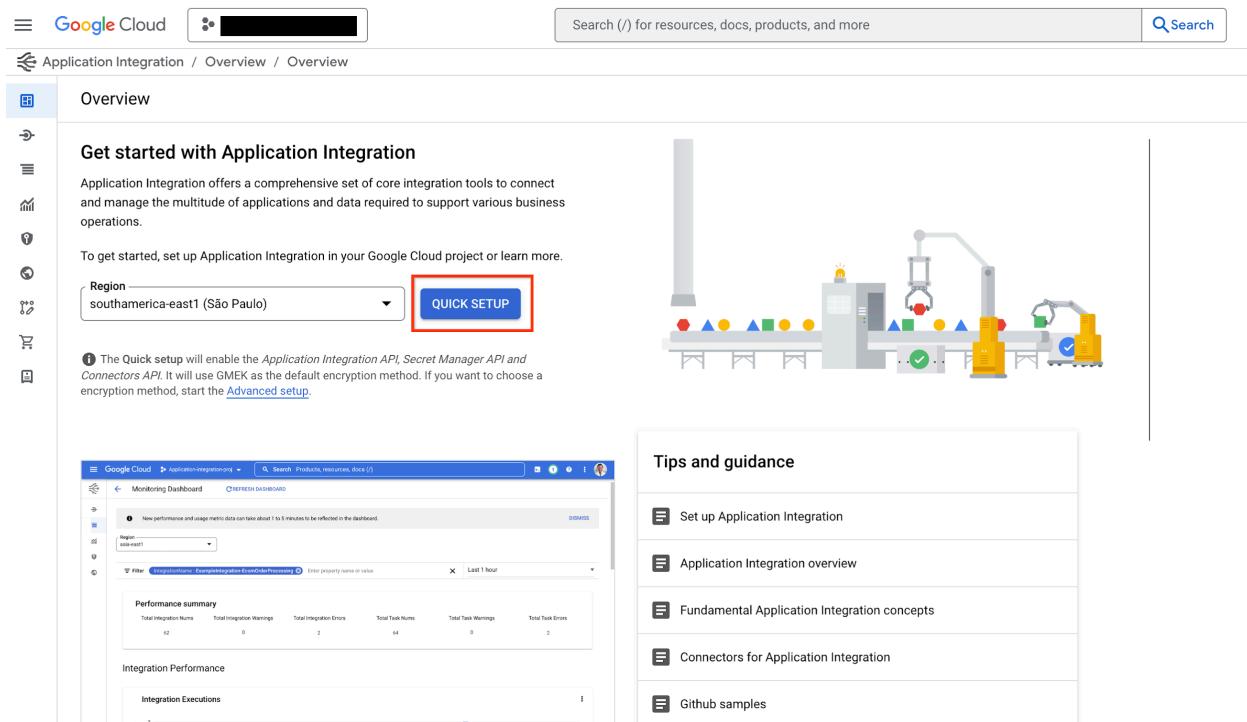
When running the agent, make sure to run adk web in project_root_folder

Use Integration Connectors

Connect your agent to enterprise applications using [Integration Connectors](#).

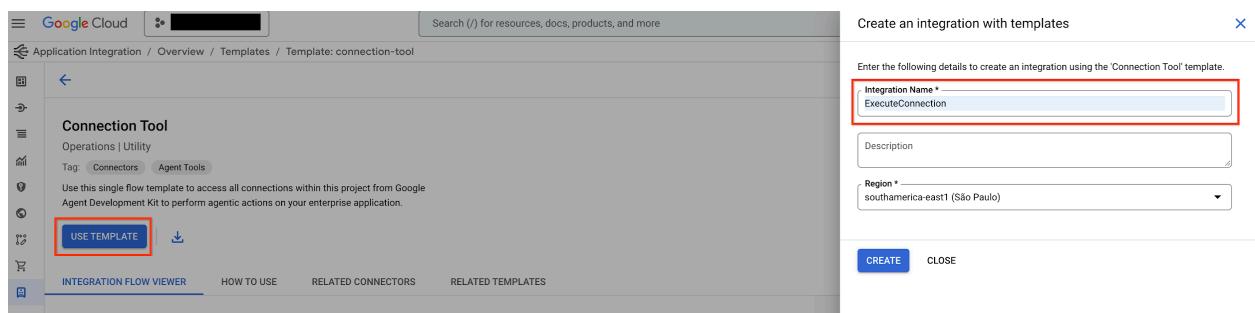
Prerequisites

1. To use a connector from Integration Connectors, you need to [provision Application Integration](#) in the same region as your connection by clicking on "QUICK SETUP" button.



The screenshot shows the Google Cloud Application Integration Overview page. On the left, there's a sidebar with icons for Overview, Get started with Application Integration, Monitoring Dashboard, and Tips and guidance. The main content area has a heading "Get started with Application Integration". It includes a "Region" dropdown set to "southamerica-east1 (São Paulo)" and a prominent blue "QUICK SETUP" button. Below the button, a note says: "The Quick setup will enable the Application Integration API, Secret Manager API and Connectors API. It will use GMEK as the default encryption method. If you want to choose a encryption method, start the [Advanced setup](#)." To the right of this text is a diagram of a factory floor with various industrial equipment and a conveyor belt. At the bottom of the main content area is a "Monitoring Dashboard" section with performance metrics like Total Integration Errors and Total Task Errors. The "Tips and guidance" sidebar on the right lists links for setting up integration, overview, fundamental concepts, connectors, and GitHub samples.

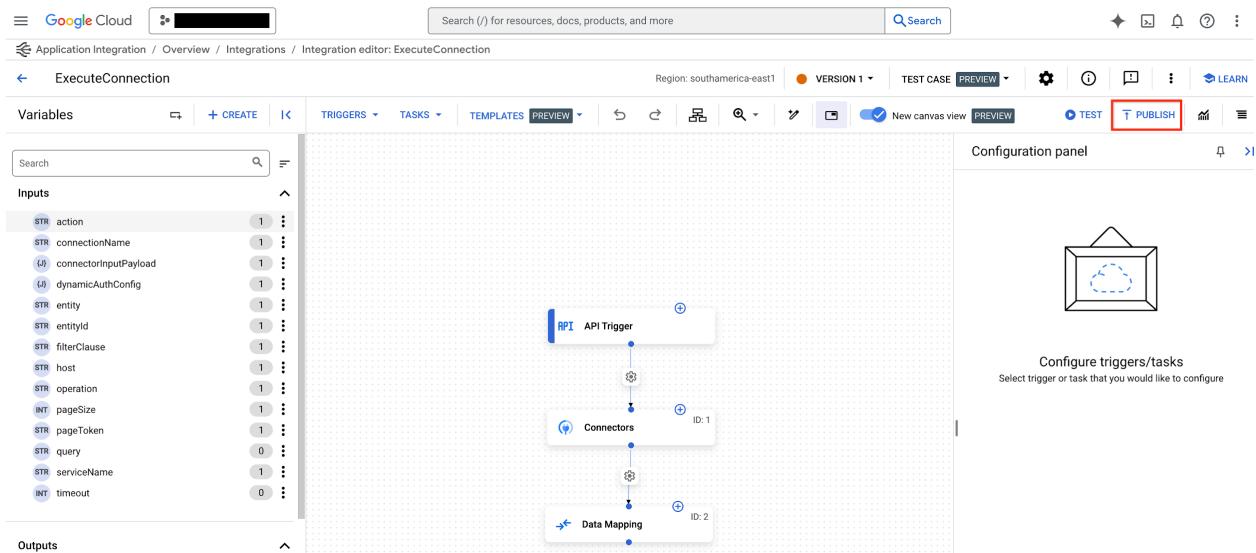
1. Go to [Connection Tool](#) template from the template library and click on "USE TEMPLATE" button.



The screenshot shows the Google Cloud Application Integration Templates page. The left sidebar lists "Connection Tool" under "Templates". The main content area shows a brief description of the Connection Tool template and a blue "USE TEMPLATE" button, which is highlighted with a red box. To the right is a modal window titled "Create an integration with templates". It contains fields for "Integration Name" (set to "ExecuteConnection"), "Description", and "Region" (set to "southamerica-east1 (São Paulo)"). At the bottom of the modal are "CREATE" and "CLOSE" buttons.

2. Fill the Integration Name as **ExecuteConnection** (It is mandatory to use this integration name only) and select the region same as the connection region. Click on "CREATE".

3. Publish the integration by using the "PUBLISH" button on the Application Integration Editor.



Steps:

1. Create a tool with `ApplicationIntegrationToolset` within your `tools.py` file

```
from
google.adk.tools.application_integration_tool.application_integration_toolset
import ApplicationIntegrationToolset
```

2.

3.

```
connector_tool = ApplicationIntegrationToolset()
```

4.

```
project="test-project", # TODO: replace with GCP project of the connection
```

5.

```
location="us-central1", #TODO: replace with location of the connection
```

6.

```
connection="test-connection", #TODO: replace with connection name
```

7.

```
entity_operations={"Entity_One": ["LIST", "CREATE"], "Entity_Two": []},#empty list for actions means all operations on the entity are supported.
```

8.

```
actions=[ "action1"], #TODO: replace with actions
```

9.

```
service_account_credentials='...', # optional. Stringified json for service account key
```

10.

```
tool_name_prefix="tool_prefix2",
```

11.

```
tool_instructions="..."
```

12.

)

13.

14. Note:

- You can provide service account to be used instead of using default credentials by generating [Service Account Key](#) and providing right Application Integration and Integration Connector IAM roles to the service account.
- To find the list of supported entities and actions for a connection, use the connectors apis: [listActions](#) or [listEntityTypes](#)

15. `ApplicationIntegrationToolset` now also supports providing `auth_scheme` and `auth_credential` for dynamic OAuth2 authentication for Integration Connectors. To use it, create a tool similar to this within your `tools.py` file:

```
from google.adk.tools.application_integration_tool.application_integration_toolset import ApplicationIntegrationToolset
```

16.

```
from google.adk.tools.openapi_tool.auth.auth_helpers import dict_to_auth_scheme
```

17.

```
from google.adk.auth import AuthCredential
```

18.

```
from google.adk.auth import AuthCredentialTypes
```

19.

```
from google.adk.auth import OAuth2Auth
```

20.

21.

```
oauth2_data_google_cloud = {
```

22.

```
    "type": "oauth2",
```

23.

```
    "flows": {
```

24.

```
        "authorizationCode": {
```

25.

```
            "authorizationUrl": "https://accounts.google.com/o/oauth2/auth",
```

26.

```
    "tokenUrl": "https://oauth2.googleapis.com/token",
```

27.

```
    "scopes": {
```

28.

```
        "https://www.googleapis.com/auth/cloud-platform": (
```

29.

```
            "View and manage your data across Google Cloud Platform"
```

30.

```
            " services"
```

31.

```
        ),
```

32.

```
        "https://www.googleapis.com/auth/calendar.readonly": "View your
calendars"
```

33.

```
    },
```

34.

```
    }
```

35.

```
},
```

36.

```
}
```

37.

38.

```
oauth_scheme = dict_to_auth_scheme(oauth2_data_google_cloud)
```

39.

40.

```
auth_credential = AuthCredential(
```

41.

```
auth_type=AuthCredentialTypes.OAUTH2,
```

42.

```
oauth2=OAuth2Auth(
```

43.

```
    client_id="...", #TODO: replace with client_id
```

44.

```
    client_secret="...", #TODO: replace with client_secret
```

45.

```
),
```

46.

```
)
```

47.

48.

```
connector_tool = ApplicationIntegrationToolset(
```

49.

```
    project="test-project", # TODO: replace with GCP project of the connection
```

50.

```
    location="us-central1", #TODO: replace with location of the connection
```

51.

```
    connection="test-connection", #TODO: replace with connection name
```

52.

```
    entity_operations={"Entity_One": ["LIST", "CREATE"], "Entity_Two": []},#empty list for actions means all operations on the entity are supported.
```

53.

```
    actions=[ "GET_calendars/%7BcalendarId%7D/events"], #TODO: replace with actions. this one is for list events
```

54.

```
    service_account_credentials='...', # optional. Stringified json for service account key
```

55.

```
    tool_name_prefix="tool_prefix2",
```

56.

```
    tool_instructions="...",
```

57.

```
    auth_scheme=oauth_scheme,
```

58.

```
    auth_credential=auth_credential
```

59.

```
)
```

60.

61. Add the tool to your agent. Update your agent.py file

```
from google.adk.agents.llm_agent import LlmAgent
```

62.

```
from .tools import connector_tool
```

63.

64.

```
root_agent = LlmAgent(
```

65.

```
    model='gemini-2.0-flash',
```

66.

```
    name='connector_agent',
```

67.

```
    instruction="Help user, leverage the tools you have access to",
```

68.

```
    tools=[connector_tool],
```

69.

```
)
```

70.

71. Configure your `__init__.py` to expose your agent

```
from . import agent
```

72.

73. Start the Google ADK Web UI and try your agent.

```
# make sure to run `adk web` from your project_root_folder
```

74.

```
adk web
```

75.

Then go to <http://localhost:8000>, and choose my_agent agent (same as the agent folder name)

Use App Integration Workflows

Use existing [Application Integration](#) workflow as a tool for your agent or create a new one.

Steps:

1. Create a tool with `ApplicationIntegrationToolset` within your `tools.py` file

```
integration_tool = ApplicationIntegrationToolset(
```

2.

```
    project="test-project", # TODO: replace with GCP project of the connection
```

3.

```
    location="us-central1", #TODO: replace with location of the connection
```

4.

```
    integration="test-integration", #TODO: replace with integration name
```

5.

```
    triggers=[ "api_trigger/test_trigger"],#TODO: replace with trigger id(s).  
Empty list would mean all api triggers in the integration to be considered.
```

6.

```
    service_account_credentials='...', #optional. Stringified json for  
service account key
```

7.

```
    tool_name_prefix="tool_prefix1",
```

8.

```
    tool_instructions="..."
```

9.

```
)
```

10.

11. Note: You can provide service account to be used instead of using default credentials by generating [Service Account Key](#) and providing right Application Integration and Integration Connector IAM roles to the service account.

12. Add the tool to your agent. Update your `agent.py` file

```
from google.adk.agents.llm_agent import LlmAgent
```

13.

```
from .tools import integration_tool, connector_tool
```

14.

15.

```
root_agent = LlmAgent(
```

16.

```
    model='gemini-2.0-flash',
```

17.

```
    name='integration_agent',
```

18.

```
    instruction="Help user, leverage the tools you have access to",
```

19.

```
    tools=[integration_tool],
```

20.

)

21.

22. Configure your `__init__.py` to expose your agent

```
from . import agent
```

23.

24. Start the Google ADK Web UI and try your agent.

```
# make sure to run `adk web` from your project_root_folder
```

25.

```
adk web
```

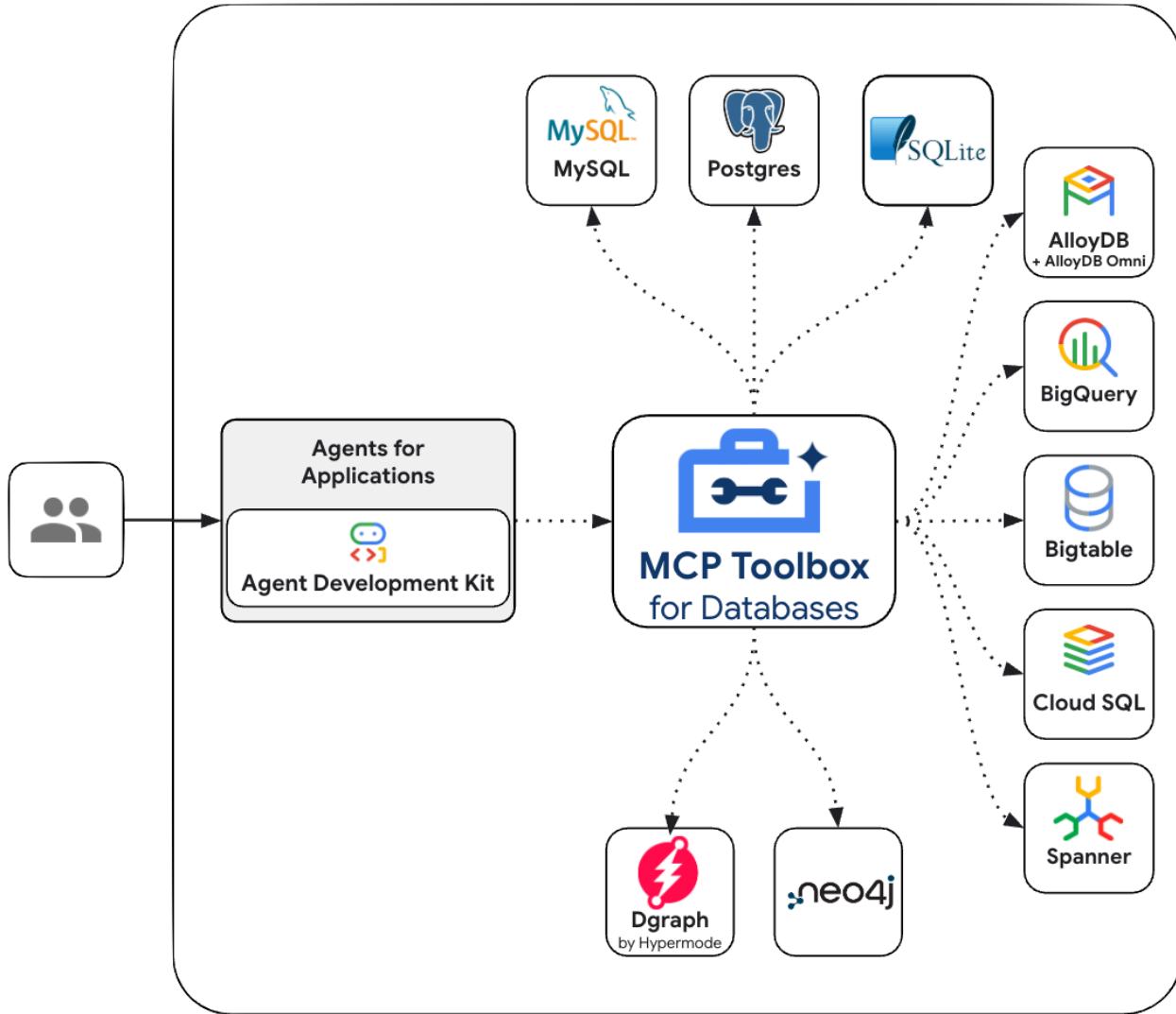
26.

27. Then go to <http://localhost:8000>, and choose my_agent agent (same as the agent folder name)

Toolbox Tools for Databases

[MCP Toolbox for Databases](#) is an open source MCP server for databases. It was designed with enterprise-grade and production-quality in mind. It enables you to develop tools easier, faster, and more securely by handling the complexities such as connection pooling, authentication, and more.

Google's Agent Development Kit (ADK) has built in support for Toolbox. For more information on [getting started](#) or [configuring](#) Toolbox, see the [documentation](#).



Configure and deploy

Toolbox is an open source server that you deploy and manage yourself. For more instructions on deploying and configuring, see the official Toolbox documentation:

- [Installing the Server](#)
- [Configuring Toolbox](#)

Install client SDK

ADK relies on the `toolbox-core` python package to use Toolbox. Install the package before getting started:

```
pip install toolbox-core
```

Loading Toolbox Tools

Once you're Toolbox server is configured and up and running, you can load tools from your server using ADK:

```
from google.adk.agents import Agent

from toolbox_core import ToolboxSyncClient

toolbox = ToolboxSyncClient("https://127.0.0.1:5000")

# Load a specific set of tools

tools = toolbox.load_toolset('my-toolset-name'),

# Load single tool
```

```
tools = toolbox.load_tool('my-tool-name'),  
  
root_agent = Agent(  
  
    ...,  
  
    tools=tools # Provide the list of tools to the Agent  
  
)
```

Advanced Toolbox Features

Toolbox has a variety of features to make developing Gen AI tools for databases. For more information, read more about the following features:

- [Authenticated Parameters](#): bind tool inputs to values from OIDC tokens automatically, making it easy to run sensitive queries without potentially leaking data
- [Authorized Invocations](#): restrict access to use a tool based on the users Auth token
- [OpenTelemetry](#): get metrics and tracing from Toolbox with OpenTelemetry

Model Context Protocol Tools

This guide walks you through two ways of integrating Model Context Protocol (MCP) with ADK.

What is Model Context Protocol (MCP)?

The Model Context Protocol (MCP) is an open standard designed to standardize how Large Language Models (LLMs) like Gemini and Claude communicate with external applications, data sources, and tools. Think of it as a universal connection mechanism that simplifies how LLMs obtain context, execute actions, and interact with various systems.

MCP follows a client-server architecture, defining how **data** (resources), **interactive templates** (prompts), and **actionable functions** (tools) are exposed by an **MCP server** and consumed by an **MCP client** (which could be an LLM host application or an AI agent).

This guide covers two primary integration patterns:

1. **Using Existing MCP Servers within ADK:** An ADK agent acts as an MCP client, leveraging tools provided by external MCP servers.
2. **Exposing ADK Tools via an MCP Server:** Building an MCP server that wraps ADK tools, making them accessible to any MCP client.

Prerequisites

Before you begin, ensure you have the following set up:

- **Set up ADK:** Follow the standard ADK [setup instructions](#) in the quickstart.
- **Install/update Python/Java:** MCP requires Python version of 3.9 or higher for Python or Java 17+.

- **Setup Node.js and npx: (Python only)** Many community MCP servers are distributed as Node.js packages and run using `npx`. Install Node.js (which includes `npx`) if you haven't already. For details, see <https://nodejs.org/en>.
- **Verify Installations: (Python only)** Confirm `adk` and `npx` are in your PATH within the activated virtual environment:

```
# Both commands should print the path to the executables.
```

```
which adk
```

```
which npx
```

1. Using MCP servers with ADK agents (ADK as an MCP client) in `adk web`

This section demonstrates how to integrate tools from external MCP (Model Context Protocol) servers into your ADK agents. This is the **most common** integration pattern when your ADK agent needs to use capabilities provided by an existing service that exposes an MCP interface. You will see how the `MCPToolset` class can be directly added to your agent's `tools` list, enabling seamless connection to an MCP server, discovery of its tools, and making them available for your agent to use. These examples primarily focus on interactions within the `adk web` development environment.

`MCPToolset` class

The `MCPToolset` class is ADK's primary mechanism for integrating tools from an MCP server. When you include an `MCPToolset` instance in your agent's `tools` list, it automatically handles the interaction with the specified MCP server. Here's how it works:

- Connection Management:** On initialization, `MCPToolset` establishes and manages the connection to the MCP server. This can be a local server process (using `StdioServerParameters` for communication over standard input/output) or a remote server (using `SseServerParams` for Server-Sent Events). The toolset also handles the graceful shutdown of this connection when the agent or application terminates.
- Tool Discovery & Adaptation:** Once connected, `MCPToolset` queries the MCP server for its available tools (via the `list_tools` MCP method). It then converts the schemas of these discovered MCP tools into ADK-compatible `BaseTool` instances.
- Exposure to Agent:** These adapted tools are then made available to your `LlmAgent` as if they were native ADK tools.
- Proxying Tool Calls:** When your `LlmAgent` decides to use one of these tools, `MCPToolset` transparently proxies the call (using the `call_tool` MCP method) to the MCP server, sends the necessary arguments, and returns the server's response back to the agent.
- Filtering (Optional):** You can use the `tool_filter` parameter when creating an `MCPToolset` to select a specific subset of tools from the MCP server, rather than exposing all of them to your agent.

The following examples demonstrate how to use `MCPToolset` within the `adk web` development environment. For scenarios where you need more fine-grained control over the MCP connection lifecycle or are not using `adk web`, refer to the "Using MCP Tools in your own Agent out of `adk web`" section later in this page.

Example 1: File System MCP Server

This example demonstrates connecting to a local MCP server that provides file system operations.

Step 1: Define your Agent with `MCPToolset`

Create an `agent.py` file (e.g., in `./adk_agent_samples/mcp_agent/agent.py`). The `MCPToolset` is instantiated directly within the `tools` list of your `LlmAgent`.

- Important:** Replace `"/path/to/your/folder"` in the `args` list with the **absolute path** to an actual folder on your local system that the MCP server can access.
- Important:** Place the `.env` file in the parent directory of the `./adk_agent_samples` directory.

```
# ./adk_agent_samples/mcp_agent/agent.py

import os # Required for path operations

from google.adk.agents import LlmAgent

from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset,
StdioServerParameters

# It's good practice to define paths dynamically if possible,
# or ensure the user understands the need for an ABSOLUTE path.

# For this example, we'll construct a path relative to this file,
# assuming '/path/to/your/folder' is in the same directory as agent.py.

# REPLACE THIS with an actual absolute path if needed for your setup.

TARGET_FOLDER_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)),
"/path/to/your/folder")
```

```
# Ensure TARGET_FOLDER_PATH is an absolute path for the MCP server.

# If you created ./adk_agent_samples/mcp_agent/your_folder,

root_agent = LlmAgent(
    model='gemini-2.0-flash',
    name='filesystem_assistant_agent',
    instruction='Help the user manage their files. You can list files, read
files, etc.',
    tools=[
        MCPToolset(
            connection_params=StdioServerParameters(
                command='npx',
                ...),
            ...),
        ...
    ],
    ...)
```

```
    args=[  
  
        "-y", # Argument for npx to auto-confirm install  
  
        "@modelcontextprotocol/server-filesystem",  
  
        # IMPORTANT: This MUST be an ABSOLUTE path to a folder the  
        # npx process can access.  
  
        # Replace with a valid absolute path on your system.  
        # For example: "/Users/youruser/accessible_mcp_files"  
  
        # or use a dynamically constructed absolute path:  
        os.path.abspath(TARGET_FOLDER_PATH),  
  
    ],  
),  
  
# Optional: Filter which tools from the MCP server are exposed
```

```
# tool_filter=['list_directory', 'read_file']
```

```
)
```

```
[,
```

```
)
```

Step 2: Create an `__init__.py` file

Ensure you have an `__init__.py` in the same directory as `agent.py` to make it a discoverable Python package for ADK.

```
# ./adk_agent_samples/mcp_agent/__init__.py
```

```
from . import agent
```

Step 3: Run `adk web` and Interact

Navigate to the parent directory of `mcp_agent` (e.g., `adk_agent_samples`) in your terminal and run:

```
cd ./adk_agent_samples # Or your equivalent parent directory
```

```
adk web
```

Note for Windows users

When hitting the `_make_subprocess_transport` `NotImplementedError`, consider using `adk web --no-reload` instead.

Once the ADK Web UI loads in your browser:

1. Select the `filesystem_assistant_agent` from the agent dropdown.
2. Try prompts like:
 - "List files in the current directory."
 - "Can you read the file named sample.txt?" (assuming you created it in `TARGET_FOLDER_PATH`).
 - "What is the content of `another_file.md`?"

You should see the agent interacting with the MCP file system server, and the server's responses (file listings, file content) relayed through the agent. The `adk web` console (terminal where you ran the command) might also show logs from the `npx` process if it outputs to stderr.



Example 2: Google Maps MCP Server

This example demonstrates connecting to the Google Maps MCP server.

Step 1: Get API Key and Enable APIs

1. **Google Maps API Key:** Follow the directions at [Use API keys](#) to obtain a Google Maps API Key.

2. Enable APIs: In your Google Cloud project, ensure the following APIs are enabled:

- Directions API
- Routes API For instructions, see the [Getting started with Google Maps Platform](#) documentation.

Step 2: Define your Agent with MCPToolset for Google Maps

Modify your `agent.py` file (e.g., in `./adk_agent_samples/mcp_agent/agent.py`). Replace `YOUR_GOOGLE_MAPS_API_KEY` with the actual API key you obtained.

```
# ./adk_agent_samples/mcp_agent/agent.py

import os

from google.adk.agents import LlmAgent

from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset,
StdioServerParameters

# Retrieve the API key from an environment variable or directly insert it.

# Using an environment variable is generally safer.

# Ensure this environment variable is set in the terminal where you run 'adk web'.
```

```
# Example: export GOOGLE_MAPS_API_KEY="YOUR_ACTUAL_KEY"

google_maps_api_key = os.environ.get("GOOGLE_MAPS_API_KEY")

if not google_maps_api_key:

    # Fallback or direct assignment for testing - NOT RECOMMENDED FOR
    PRODUCTION

    google_maps_api_key = "YOUR_GOOGLE_MAPS_API_KEY_HERE" # Replace if not
    using env var

    if google_maps_api_key == "YOUR_GOOGLE_MAPS_API_KEY_HERE":

        print("WARNING: GOOGLE_MAPS_API_KEY is not set. Please set it as an
              environment variable or in the script.")

    # You might want to raise an error or exit if the key is crucial and
    not found.
```

```
root_agent = LlmAgent(  
  
    model='gemini-2.0-flash',  
  
    name='maps_assistant_agent',  
  
    instruction='Help the user with mapping, directions, and finding places  
using Google Maps tools.',  
  
    tools=[  
  
        MCPToolset(  
  
            connection_params=StdioServerParameters(  
  
                command='npx',  
  
                args=[  
  
                    "-y",  
  
                    "@modelcontextprotocol/server-google-maps",  
                ]  
            )  
        )  
    ]  
)
```

```
    ],
    # Pass the API key as an environment variable to the npx
process
    # This is how the MCP server for Google Maps expects the key.

    env={
        "GOOGLE_MAPS_API_KEY": google_maps_api_key
    }

),
# You can filter for specific Maps tools if needed:
# tool_filter=['get_directions', 'find_place_by_id']

),
],
] ,
```

```
)
```

Step 3: Ensure `__init__.py` Exists

If you created this in Example 1, you can skip this. Otherwise, ensure you have an `__init__.py` in the `./adk_agent_samples/mcp_agent/` directory:

```
# ./adk_agent_samples/mcp_agent/__init__.py
```

```
from . import agent
```

Step 4: Run adk web and Interact

1. **Set Environment Variable (Recommended):** Before running `adk web`, it's best to set your Google Maps API key as an environment variable in your terminal:

```
export GOOGLE_MAPS_API_KEY="YOUR_ACTUAL_GOOGLE_MAPS_API_KEY"
```

- 2.
3. Replace `YOUR_ACTUAL_GOOGLE_MAPS_API_KEY` with your key.
4. **Run `adk web`:** Navigate to the parent directory of `mcp_agent` (e.g., `adk_agent_samples`) and run:

```
cd ./adk_agent_samples # Or your equivalent parent directory
```

- 5.

```
adk web
```

- 6.

7. **Interact in the UI:**

- Select the `maps_assistant_agent`.

- Try prompts like:
 - "Get directions from GooglePlex to SFO."
 - "Find coffee shops near Golden Gate Park."
 - "What's the route from Paris, France to Berlin, Germany?"

You should see the agent use the Google Maps MCP tools to provide directions or location-based information.



2. Building an MCP server with ADK tools (MCP server exposing ADK)

This pattern allows you to wrap existing ADK tools and make them available to any standard MCP client application. The example in this section exposes the ADK `load_web_page` tool through a custom-built MCP server.

Summary of steps

You will create a standard Python MCP server application using the `mcp` library. Within this server, you will:

1. Instantiate the ADK tool(s) you want to expose (e.g., `FunctionTool(load_web_page)`).
2. Implement the MCP server's `@app.list_tools()` handler to advertise the ADK tool(s). This involves converting the ADK tool definition to the MCP

- schema using the `adk_to_mcp_tool_type` utility from `google.adk.tools.mcp_tool.conversion_utils`.
3. Implement the MCP server's `@app.call_tool()` handler. This handler will:
 - Receive tool call requests from MCP clients.
 - Identify if the request targets one of your wrapped ADK tools.
 - Execute the ADK tool's `.run_async()` method.
 - Format the ADK tool's result into an MCP-compliant response (e.g., `mcp.types.TextContent`).

Prerequisites

Install the MCP server library in the same Python environment as your ADK installation:

```
pip install mcp
```

Step 1: Create the MCP Server Script

Create a new Python file for your MCP server, for example, `my_adk_mcp_server.py`.

Step 2: Implement the Server Logic

Add the following code to `my_adk_mcp_server.py`. This script sets up an MCP server that exposes the ADK `load_web_page` tool.

```
# my_adk_mcp_server.py
```

```
import asyncio
```

```
import json
```

```
import os

from dotenv import load_dotenv

# MCP Server Imports

from mcp import types as mcp_types # Use alias to avoid conflict

from mcp.server.lowlevel import Server, NotificationOptions

from mcp.server.models import InitializationOptions

import mcp.server.stdio # For running as a stdio server

# ADK Tool Imports

from google.adk.tools.function_tool import FunctionTool

from google.adk.tools.load_web_page import load_web_page # Example ADK tool
```

```
# ADK <-> MCP Conversion Utility

from google.adk.tools.mcp_tool.conversion_utils import adk_to_mcp_tool_type

# --- Load Environment Variables (If ADK tools need them, e.g., API keys) ---

load_dotenv() # Create a .env file in the same directory if needed

# --- Prepare the ADK Tool ---

# Instantiate the ADK tool you want to expose.

# This tool will be wrapped and called by the MCP server.

print("Initializing ADK load_web_page tool...")

adk_tool_to_expose = FunctionTool(load_web_page)
```

```
print(f"ADK tool '{adk_tool_to_expose.name}' initialized and ready to be exposed via MCP.")
```

```
# --- End ADK Tool Prep ---
```

```
# --- MCP Server Setup ---
```

```
print("Creating MCP Server instance...")
```

```
# Create a named MCP Server instance using the mcp.server library
```

```
app = Server("adk-tool-exposing-mcp-server")
```

```
# Implement the MCP server's handler to list available tools
```

```
@app.list_tools()
```

```
async def list_mcp_tools() -> list[mcp_types.Tool]:
```

```
"""MCP handler to list tools this server exposes."""

print("MCP Server: Received list_tools request.")

# Convert the ADK tool's definition to the MCP Tool schema format

mcp_tool_schema = adk_to_mcp_tool_type(adk_tool_to_expose)

print(f"MCP Server: Advertising tool: {mcp_tool_schema.name}")

return [mcp_tool_schema]

# Implement the MCP server's handler to execute a tool call

@app.call_tool()

async def call_mcp_tool(

    name: str, arguments: dict

) -> list[mcp_types.Content]: # MCP uses mcp_types.Content
```

```
"""MCP handler to execute a tool call requested by an MCP client."""

    print(f"MCP Server: Received call_tool request for '{name}' with args:
{arguments}")

# Check if the requested tool name matches our wrapped ADK tool

if name == adk_tool_to_expose.name:

    try:
        # Execute the ADK tool's run_async method.

        # Note: tool_context is None here because this MCP server is

        # running the ADK tool outside of a full ADK Runner invocation.

        # If the ADK tool requires ToolContext features (like state or
        auth),
```

```
# this direct invocation might need more sophisticated handling.

adk_tool_response = await adk_tool_to_expose.run_async(
    args=arguments,
    tool_context=None,
)

print(f"MCP Server: ADK tool '{name}' executed. Response:
{adk_tool_response}")

# Format the ADK tool's response (often a dict) into an
MCP-compliant format.

# Here, we serialize the response dictionary as a JSON string
within TextContent.

# Adjust formatting based on the ADK tool's output and client
needs.
```

```
    response_text = json.dumps(adk_tool_response, indent=2)

    # MCP expects a list of mcp_types.Content parts

    return [mcp_types.TextContent(type="text", text=response_text)]


except Exception as e:

    print(f"MCP Server: Error executing ADK tool '{name}': {e}")

    # Return an error message in MCP format

    error_text = json.dumps({"error": f"Failed to execute tool '{name}': {str(e)}"})

    return [mcp_types.TextContent(type="text", text=error_text)]


else:

    # Handle calls to unknown tools
```

```
    print(f"MCP Server: Tool '{name}' not found/exposed by this server.")

    error_text = json.dumps({"error": f"Tool '{name}' not implemented by
this server."})

    return [mcp_types.TextContent(type="text", text=error_text)]


# --- MCP Server Runner ---


async def run_mcp_stdio_server():

    """Runs the MCP server, listening for connections over standard
input/output."""

    # Use the stdio_server context manager from the mcp.server_stdio library

    async with mcp.server_stdio_stdio_server() as (read_stream, write_stream):

        print("MCP Stdio Server: Starting handshake with client...")

        await app.run(
```

```
    read_stream,  
  
    write_stream,  
  
    InitializationOptions(  
  
        server_name=app.name, # Use the server name defined above  
  
        server_version="0.1.0",  
  
        capabilities=app.get_capabilities(  
  
            # Define server capabilities - consult MCP docs for options  
  
            notification_options=NotificationOptions(),  
  
            experimental_capabilities={},  
  
        ),  
  
    ),  
)
```

```
    print("MCP Stdio Server: Run loop finished or client disconnected.")

if __name__ == "__main__":
    print("Launching MCP Server to expose ADK tools via stdio...")

    try:
        asyncio.run(run_mcp_stdio_server())
    except KeyboardInterrupt:
        print("\nMCP Server (stdio) stopped by user.")

    except Exception as e:
        print(f"MCP Server (stdio) encountered an error: {e}")

    finally:
```

```
    print("MCP Server (stdio) process exiting.")

# --- End MCP Server ---
```

Step 3: Test your Custom MCP Server with an ADK Agent

Now, create an ADK agent that will act as a client to the MCP server you just built. This ADK agent will use `MCPToolset` to connect to your `my_adk_mcp_server.py` script.

Create an `agent.py` (e.g., in `./adk_agent_samples/mcp_client_agent/agent.py`):

```
# ./adk_agent_samples/mcp_client_agent/agent.py

import os

from google.adk.agents import LlmAgent

from google.adk.tools.mcp_tool import MCPToolset, StdioServerParameters

# IMPORTANT: Replace this with the ABSOLUTE path to your my_adk_mcp_server.py
script

PATH_TO_YOUR_MCP_SERVER_SCRIPT = "/path/to/your/my_adk_mcp_server.py" # <<<
REPLACE
```

```
if PATH_TO_YOUR_MCP_SERVER_SCRIPT == "/path/to/your/my_adk_mcp_server.py":  
  
    print("WARNING: PATH_TO_YOUR_MCP_SERVER_SCRIPT is not set. Please update it  
in agent.py.")  
  
    # Optionally, raise an error if the path is critical  
  
root_agent = LlmAgent(  
  
    model='gemini-2.0-flash',  
  
    name='web_reader_mcp_client_agent',  
  
    instruction="Use the 'load_web_page' tool to fetch content from a URL  
provided by the user.",  
  
    tools=[
```

```
MCPToolset(  
    connection_params=StdioServerParameters(  
        command='python3', # Command to run your MCP server script  
        args=[PATH_TO_YOUR_MCP_SERVER_SCRIPT], # Argument is the path  
        to the script  
    )  
  
    # tool_filter=['load_web_page'] # Optional: ensure only specific  
    tools are loaded  
  
)  
]  
)
```

And an `__init__.py` in the same directory:

```
# ./adk_agent_samples/mcp_client_agent/__init__.py  
  
from . import agent
```

To run the test:

1. **Start your custom MCP server (optional, for separate observation):** You can run your `my_adk_mcp_server.py` directly in one terminal to see its logs:

```
python3 /path/to/your/my_adk_mcp_server.py
```

- 2.
3. It will print "Launching MCP Server..." and wait. The ADK agent (run via `adk web`) will then connect to this process if the `command` in `StdioServerParameters` is set up to execute it. (*Alternatively, MCPToolset will start this server script as a subprocess automatically when the agent initializes*).
4. **Run `adk web` for the client agent:** Navigate to the parent directory of `mcp_client_agent` (e.g., `adk_agent_samples`) and run:

```
cd ./adk_agent_samples # Or your equivalent parent directory
```

- 5.

```
adk web
```

- 6.
7. **Interact in the ADK Web UI:**
 - Select the `web_reader_mcp_client_agent`.
 - Try a prompt like: "Load the content from <https://example.com>"

The ADK agent (`web_reader_mcp_client_agent`) will use `MCPToolset` to start and connect to your `my_adk_mcp_server.py`. Your MCP server will receive the `call_tool` request, execute the ADK `load_web_page` tool, and return the result. The ADK agent will then relay this information. You should see logs from both the ADK Web UI (and its terminal) and potentially from your `my_adk_mcp_server.py` terminal if you ran it separately.

This example demonstrates how ADK tools can be encapsulated within an MCP server, making them accessible to a broader range of MCP-compliant clients, not just ADK agents.

Refer to the [documentation](#), to try it out with Claude Desktop.

Using MCP Tools in your own Agent out of adk web

This section is relevant to you if:

- You are developing your own Agent using ADK
- And, you are **NOT** using `adk web`,
- And, you are exposing the agent via your own UI

Using MCP Tools requires a different setup than using regular tools, due to the fact that specs for MCP Tools are fetched asynchronously from the MCP Server running remotely, or in another process.

The following example is modified from the "Example 1: File System MCP Server" example above. The main differences are:

1. Your tool and agent are created asynchronously
2. You need to properly manage the exit stack, so that your agents and tools are destructed properly when the connection to MCP Server is closed.

```
# agent.py (modify get_tools_async and other parts as needed)
```

```
# ./adk_agent_samples/mcp_agent/agent.py
```

```
import os
```

```
import asyncio

from dotenv import load_dotenv

from google.genai import types

from google.adk.agents.llm_agent import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.artifacts.in_memory_artifact_service import
InMemoryArtifactService # Optional

from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset, SseServerParams,
StdioServerParameters

# Load environment variables from .env file in the parent directory

# Place this near the top, before using env vars like API keys
```

```
load_dotenv('../.env')

# Ensure TARGET_FOLDER_PATH is an absolute path for the MCP server.

TARGET_FOLDER_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)),
"/path/to/your/folder")

# --- Step 1: Agent Definition ---

async def get_agent_async():

    """Creates an ADK Agent equipped with tools from the MCP Server."""

    toolset = MCPToolset()

        # Use StdioServerParameters for local process communication

        connection_params=StdioServerParameters(
```

```
    command='npx', # Command to run the server

    args=[ "-y",      # Arguments for the command

           "@modelcontextprotocol/server-filesystem",

           TARGET_FOLDER_PATH],

    ),

    tool_filter=['read_file', 'list_directory'] # Optional: filter specific
tools

# For remote servers, you would use SseServerParams instead:

    # connection_params=SseServerParams(url="http://remote-server:port/path",
headers={...})

)

# Use in an agent
```

```
root_agent = LlmAgent(  
  
    model='gemini-2.0-flash', # Adjust model name if needed based on  
availability  
  
    name='enterprise_assistant',  
  
    instruction='Help user accessing their file systems',  
  
    tools=[toolset], # Provide the MCP tools to the ADK agent  
  
)  
  
return root_agent, toolset  
  
# --- Step 2: Main Execution Logic ---  
  
async def async_main():  
  
    session_service = InMemorySessionService()
```

```
# Artifact service might not be needed for this example

artifacts_service = InMemoryArtifactService()

session = await session_service.create_session(
    state={}, app_name='mcp_filesystem_app', user_id='user_fs'
)

# TODO: Change the query to be relevant to YOUR specified folder.

# e.g., "list files in the 'documents' subfolder" or "read the file
'notes.txt'"

query = "list files in the tests folder"

print(f"User Query: '{query}'")
```

```
content = types.Content(role='user', parts=[types.Part(text=query)])  
  
root_agent, toolset = await get_agent_async()  
  
runner = Runner(  
  
    app_name='mcp_filesystem_app',  
  
    agent=root_agent,  
  
    artifact_service=artifacts_service, # Optional  
  
    session_service=session_service,  
  
)  
  
print("Running agent...")
```

```
events_async = runner.run_async(  
    session_id=session.id, user_id=session.user_id, new_message=content  
)  
  
async for event in events_async:  
  
    print(f"Event received: {event}")  
  
# Cleanup is handled automatically by the agent framework  
  
# But you can also manually close if needed:  
  
print("Closing MCP server connection...")  
  
await toolset.close()
```

```
print("Cleanup complete.")

if __name__ == '__main__':
    try:
        asyncio.run(async_main())
    except Exception as e:
        print(f"An error occurred: {e}")
```

Key considerations

When working with MCP and ADK, keep these points in mind:

- **Protocol vs. Library:** MCP is a protocol specification, defining communication rules. ADK is a Python library/framework for building agents. MCPToolset bridges these by implementing the client side of the MCP protocol within the ADK framework. Conversely, building an MCP server in Python requires using the model-context-protocol library.
- **ADK Tools vs. MCP Tools:**

- ADK Tools (BaseTool, FunctionTool, AgentTool, etc.) are Python objects designed for direct use within the ADK's LlmAgent and Runner.
- MCP Tools are capabilities exposed by an MCP Server according to the protocol's schema. MCPToolset makes these look like ADK tools to an LlmAgent.
- Langchain/CrewAI Tools are specific implementations within those libraries, often simple functions or classes, lacking the server/protocol structure of MCP. ADK offers wrappers (LangchainTool, CrewaiTool) for some interoperability.
- **Asynchronous nature:** Both ADK and the MCP Python library are heavily based on the asyncio Python library. Tool implementations and server handlers should generally be async functions.
- **Stateful sessions (MCP):** MCP establishes stateful, persistent connections between a client and server instance. This differs from typical stateless REST APIs.
 - **Deployment:** This statefulness can pose challenges for scaling and deployment, especially for remote servers handling many users. The original MCP design often assumed client and server were co-located. Managing these persistent connections requires careful infrastructure considerations (e.g., load balancing, session affinity).
 - **ADK MCPToolset:** Manages this connection lifecycle. The exit_stack pattern shown in the examples is crucial for ensuring the connection (and potentially the server process) is properly terminated when the ADK agent finishes.

Further Resources

- [Model Context Protocol Documentation](#)
- [MCP Specification](#)
- [MCP Python SDK & Examples](#)

OpenAPI Integration



Integrating REST APIs with OpenAPI

ADK simplifies interacting with external REST APIs by automatically generating callable tools directly from an [OpenAPI Specification \(v3.x\)](#). This eliminates the need to manually define individual function tools for each API endpoint.

Core Benefit

Use `OpenAPIToolset` to instantly create agent tools (`RestApiTool`) from your existing API documentation (OpenAPI spec), enabling agents to seamlessly call your web services.

Key Components

- `OpenAPIToolset`: This is the primary class you'll use. You initialize it with your OpenAPI specification, and it handles the parsing and generation of tools.
- `RestApiTool`: This class represents a single, callable API operation (like `GET /pets/{petId}` or `POST /pets`). `OpenAPIToolset` creates one `RestApiTool` instance for each operation defined in your spec.

How it Works

The process involves these main steps when you use OpenAPIToolset:

1. **Initialization & Parsing:**
 - You provide the OpenAPI specification to OpenAPIToolset either as a Python dictionary, a JSON string, or a YAML string.
 - The toolset internally parses the spec, resolving any internal references (`$ref`) to understand the complete API structure.
2. **Operation Discovery:**
 - It identifies all valid API operations (e.g., `GET`, `POST`, `PUT`, `DELETE`) defined within the `paths` object of your specification.
3. **Tool Generation:**
 - For each discovered operation, OpenAPIToolset automatically creates a corresponding `RestApiTool` instance.
 - **Tool Name:** Derived from the `operationId` in the spec (converted to `snake_case`, max 60 chars). If `operationId` is missing, a name is generated from the method and path.
 - **Tool Description:** Uses the `summary` or `description` from the operation for the LLM.
 - **API Details:** Stores the required HTTP method, path, server base URL, parameters (path, query, header, cookie), and request body schema internally.
4. **RestApiTool Functionality:** Each generated `RestApiTool`:
 - **Schema Generation:** Dynamically creates a `FunctionDeclaration` based on the operation's parameters and request body. This schema tells the LLM how to call the tool (what arguments are expected).
 - **Execution:** When called by the LLM, it constructs the correct HTTP request (URL, headers, query params, body) using the arguments provided by the LLM and the details from the OpenAPI spec. It handles authentication (if configured) and executes the API call using the `requests` library.
 - **Response Handling:** Returns the API response (typically JSON) back to the agent flow.
5. **Authentication:** You can configure global authentication (like API keys or OAuth - see [Authentication](#) for details) when initializing OpenAPIToolset. This authentication configuration is automatically applied to all generated `RestApiTool` instances.

Usage Workflow

Follow these steps to integrate an OpenAPI spec into your agent:

1. **Obtain Spec:** Get your OpenAPI specification document (e.g., load from a `.json` or `.yaml` file, fetch from a URL).
2. **Instantiate Toolset:** Create an `OpenAPIToolset` instance, passing the spec content and type (`spec_str/spec_dict`, `spec_str_type`). Provide authentication details (`auth_scheme, auth_credential`) if required by the API.

```
from google.adk.tools.openapi_tool.openapi_spec_parser.openapi_toolset import  
OpenAPIToolset
```

3.

4.

```
# Example with a JSON string
```

5.

```
openapi_spec_json = '...' # Your OpenAPI JSON string
```

6.

```
toolset = OpenAPIToolset(spec_str=openapi_spec_json, spec_str_type="json")
```

7.

8.

```
# Example with a dictionary
```

9.

```
# openapi_spec_dict = {...} # Your OpenAPI spec as a dict
```

10.

```
# toolset = OpenAPIToolset(spec_dict=openapi_spec_dict)
```

11.

12. **Add to Agent:** Include the retrieved tools in your `LlmAgent`'s `tools` list.

```
from google.adk.agents import LlmAgent
```

13.

14.

```
my_agent = LlmAgent(
```

15.

```
    name="api_interacting_agent",
```

16.

```
    model="gemini-2.0-flash", # Or your preferred model
```

17.

```
    tools=[toolset], # Pass the toolset
```

18.

```
# ... other agent config ...
```

19.

)

20.

21. **Instruct Agent:** Update your agent's instructions to inform it about the new API capabilities and the names of the tools it can use (e.g., `list_pets`, `create_pet`).

The tool descriptions generated from the spec will also help the LLM.

22. **Run Agent:** Execute your agent using the `Runner`. When the LLM determines it needs to call one of the APIs, it will generate a function call targeting the appropriate `RestApiTool`, which will then handle the HTTP request automatically.

Example

This example demonstrates generating tools from a simple Pet Store OpenAPI spec (using `httpbin.org` for mock responses) and interacting with them via an agent.

Code: Pet Store API

```
openapi_example.py
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
#     http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
import asyncio
```

```
import uuid # For unique session IDs

from dotenv import load_dotenv

from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

# --- OpenAPI Tool Imports ---

from google.adk.tools.openapi_tool.openapi_spec_parser.openapi_toolset import
OpenAPIToolset
```

```
# --- Load Environment Variables (If ADK tools need them, e.g., API keys) ---
```

```
load_dotenv() # Create a .env file in the same directory if needed
```

```
# --- Constants ---
```

```
APP_NAME_OPENAPI = "openapi_petstore_app"
```

```
USER_ID_OPENAPI = "user_openapi_1"
```

```
SESSION_ID_OPENAPI = f"session_openapi_{uuid.uuid4()}" # Unique session ID
```

```
AGENT_NAME_OPENAPI = "petstore_manager_agent"
```

```
GEMINI_MODEL = "gemini-2.0-flash"
```

```
# --- Sample OpenAPI Specification (JSON String) ---  
  
# A basic Pet Store API example using httpbin.org as a mock server  
  
openapi_spec_string = """  
  
{  
  
    "openapi": "3.0.0",  
  
    "info": {  
  
        "title": "Simple Pet Store API (Mock)",  
  
        "version": "1.0.1",  
  
        "description": "An API to manage pets in a store, using httpbin for responses."  
  
    },  
  
    "servers": [  
  
]
```

```
{  
  "url": "https://httpbin.org",  
  
  "description": "Mock server (httpbin.org)"  
}  
  
],  
  
  "paths": {  
    "/get": {  
      "get": {  
        "summary": "List all pets (Simulated)",  
        "operationId": "listPets",  
  
        "description": "Simulates returning a list of pets. Uses httpbin's /get endpoint which echoes query parameters."  
      }  
    }  
  }  
}
```

```
  "parameters": [  
    {  
      "name": "limit",  
      "in": "query",  
      "description": "Maximum number of pets to return",  
      "required": false,  
      "schema": { "type": "integer", "format": "int32" }  
    },  
    {  
      "name": "status",  
      "in": "query",  
      "description": "Filter by status  
      valid values are  
      'available',  
      'pending',  
      'sold'"  
    }  
  ]
```

```
        "description": "Filter pets by status",  
  
        "required": false,  
  
        "schema": { "type": "string", "enum": [ "available", "pending", "sold" ] }  
  
    },  
  
],  
  
"responses": {  
  
    "200": {  
  
        "description": "A list of pets (echoed query params).",  
  
        "content": { "application/json": { "schema": { "type": "object" } } }  
  
    },  
}
```

```
        },  
  
    },  
  
,  
  
"/post": {  
  
    "post": {  
  
        "summary": "Create a pet (Simulated)",  
  
        "operationId": "createPet",  
  
        "description": "Simulates adding a new pet. Uses httpbin's /post endpoint  
which echoes the request body.",  
  
        "requestBody": {  
  
            "description": "Pet object to add",  
  
            "required": true,  
        },  
    },  
},
```

```
    "content": {  
  
        "application/json": {  
  
            "schema": {  
  
                "type": "object",  
  
                "required": [ "name" ],  
  
                "properties": {  
  
                    "name": {"type": "string", "description": "Name of the pet"},  
  
                    "tag": {"type": "string", "description": "Optional tag for the  
pet"}  
                }  
            }  
        }  
    }
```

```
        }
```

```
    }
```

```
},
```

```
    "responses": {
```

```
        "201": {
```

```
            "description": "Pet created successfully (echoed request body).",
```

```
            "content": { "application/json": { "schema": { "type": "object" } } }
```

```
        }
```

```
    }
```

```
    },
```

```
},
```

```
  "/get?petId={petId}": {  
    "get": {  
      "summary": "Info for a specific pet (Simulated)",  
      "operationId": "showPetById",  
      "description": "Simulates returning info for a pet ID. Uses httpbin's /get endpoint.",  
      "parameters": [  
        {  
          "name": "petId",  
          "in": "path",  
          "description": "This is actually passed as a query param to httpbin /get",  
          "required": true,  
          "type": "string"  
        }  
      ]  
    }  
  }  
}
```

```
        "schema": { "type": "integer", "format": "int64" }

    }

],
"responses": {
    "200": {
        "description": "Information about the pet (echoed query params)",
        "content": { "application/json": { "schema": { "type": "object" } } }
    },
    "404": { "description": "Pet not found (simulated)" }
}

}
```

```
    }
```

```
}
```

```
}
```

```
    ...
```

```
# --- Create OpenAPIToolset ---
```

```
petstore_toolset = OpenAPIToolset(
```

```
    spec_str=openapi_spec_string,
```

```
    spec_str_type='json',
```

```
    # No authentication needed for httpbin.org
```

```
)
```

```
# --- Agent Definition ---

root_agent = LlmAgent(
    name=AGENT_NAME_OPENAPI,
    model=GEMINI_MODEL,
    tools=[petstore_toolset], # Pass the list of RestApiTool objects
    instruction="""You are a Pet Store assistant managing pets via an API.

    Use the available tools to fulfill user requests.

    When creating a pet, confirm the details echoed back by the API.

    When listing pets, mention any filters used (like limit or status).

    When showing a pet by ID, state the ID you requested.

```

```
    """ ,
```

```
        description="Manages a Pet Store using tools generated from an OpenAPI spec."
```

```
)
```

```
# --- Session and Runner Setup ---
```

```
async def setup_session_and_runner():
```

```
    session_service_openapi = InMemorySessionService()
```

```
    runner_openapi = Runner(
```

```
        agent=root_agent,
```

```
        app_name=APP_NAME_OPENAPI,
```

```
        session_service=session_service_openapi,
```

```
    )

    await session_service_openapi.create_session(
        app_name=APP_NAME_OPENAPI,
        user_id=USER_ID_OPENAPI,
        session_id=SESSION_ID_OPENAPI,
    )

    return runner_openapi

# --- Agent Interaction Function ---

async def call.openapi_agent_async(query, runner_openapi):

    print("\n--- Running OpenAPI Pet Store Agent ---")
```

```
    print(f"Query: {query}")

    content = types.Content(role='user', parts=[types.Part(text=query)])

    final_response_text = "Agent did not provide a final text response."

try:

    async for event in runner.openapi.run_async(
        user_id=USER_ID_OPENAPI, session_id=SESSION_ID_OPENAPI,
        new_message=content

    ):

# Optional: Detailed event logging for debugging

        # print(f" DEBUG Event: Author={event.author}, Type={'Final' if
event.is_final_response() else 'Intermediate'},
Content={str(event.content)[:100]}...")
```

```
if event.get_function_calls():

    call = event.get_function_calls()[0]

    print(f"  Agent Action: Called function '{call.name}' with args
{call.args}")

elif event.get_function_responses():

    response = event.get_function_responses()[0]

    print(f"  Agent Action: Received response for '{response.name}'")

    # print(f"  Tool Response Snippet:
#{str(response.response)[:200]}...") # Uncomment for response details

elif event.is_final_response() and event.content and
event.content.parts:

    # Capture the last final text response

    final_response_text = event.content.parts[0].text.strip()
```

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

except Exception as e:

    print(f"An error occurred during agent run: {e}")

import traceback

traceback.print_exc() # Print full traceback for errors

print("-" * 30)

# --- Run Examples ---

async def run.openapi_example():
```

```
runner.openapi = await setup_session_and_runner()

# Trigger listPets

await call.openapi_agent_async("Show me the pets available.", runner.openapi)

# Trigger createPet

await call.openapi_agent_async("Please add a new dog named 'Dukey'.",
runner.openapi)

# Trigger showPetById

await call.openapi_agent_async("Get info for pet with ID 123.", runner.openapi)

# --- Execute ---
```

```
if __name__ == "__main__":
    print("Executing OpenAPI example...")

# Use asyncio.run() for top-level execution

try:
    asyncio.run(run.openapi_example())

except RuntimeError as e:
    if "cannot be called from a running event loop" in str(e):
        print("Info: Cannot run asyncio.run from a running event loop (e.g., Jupyter/Colab).")

    # If in Jupyter/Colab, you might need to run like this:

    # await run.openapi_example()

else:
```

```
    raise e
```

```
print("OpenAPI example finished.")
```

[Previous](#)
[MCP tools](#)

[Next](#)
[Authentication](#)

[Material for MkDocs](#)

Authenticating with Tools



Core Concepts

Many tools need to access protected resources (like user data in Google Calendar, Salesforce records, etc.) and require authentication. ADK provides a system to handle various authentication methods securely.

The key components involved are:

1. **AuthScheme**: Defines *how* an API expects authentication credentials (e.g., as an API Key in a header, an OAuth 2.0 Bearer token). ADK supports the same types of authentication schemes as OpenAPI 3.0. To know more about what each type of credential is, refer to [OpenAPI doc: Authentication](#). ADK uses specific classes like `APIKey`, `HTTPBearer`, `OAuth2`, `OpenIdConnectWithConfig`.
2. **AuthCredential**: Holds the *initial* information needed to *start* the authentication process (e.g., your application's OAuth Client ID/Secret, an API key value). It includes an `auth_type` (like `API_KEY`, `OAUTH2`, `SERVICE_ACCOUNT`) specifying the credential type.

The general flow involves providing these details when configuring a tool. ADK then attempts to automatically exchange the initial credential for a usable one (like an access token) before the tool makes an API call. For flows requiring user interaction (like OAuth consent), a specific interactive process involving the Agent Client application is triggered.

Supported Initial Credential Types

- **API_KEY**: For simple key/value authentication. Usually requires no exchange.
- **HTTP**: Can represent Basic Auth (not recommended/supported for exchange) or already obtained Bearer tokens. If it's a Bearer token, no exchange is needed.
- **OAUTH2**: For standard OAuth 2.0 flows. Requires configuration (client ID, secret, scopes) and often triggers the interactive flow for user consent.
- **OPEN_ID_CONNECT**: For authentication based on OpenID Connect. Similar to OAuth2, often requires configuration and user interaction.
- **SERVICE_ACCOUNT**: For Google Cloud Service Account credentials (JSON key or Application Default Credentials). Typically exchanged for a Bearer token.

Configuring Authentication on Tools

You set up authentication when defining your tool:

- **RestApiTool / OpenAPIToolset**: Pass `auth_scheme` and `auth_credential` during initialization
- **GoogleApiToolSet Tools**: ADK has built-in 1st party tools like Google Calendar, BigQuery etc.,. Use the toolset's specific method.
- **APIHubToolset / ApplicationIntegrationToolset**: Pass `auth_scheme` and `auth_credential` during initialization, if the API managed in API Hub / provided by Application Integration requires authentication.

WARNING

Storing sensitive credentials like access tokens and especially refresh tokens directly in the session state might pose security risks depending on your session storage backend (`SessionService`) and overall application security posture.

- **InMemorySessionService**: Suitable for testing and development, but data is lost when the process ends. Less risk as it's transient.
- **Database/Persistent Storage**: **Strongly consider encrypting** the token data before storing it in the database using a robust encryption library (like `cryptography`) and managing encryption keys securely (e.g., using a key management service).
- **Secure Secret Stores**: For production environments, storing sensitive credentials in a dedicated secret manager (like Google Cloud Secret Manager or HashiCorp Vault) is the **most recommended approach**. Your tool could potentially store only short-lived access tokens or secure references (not the refresh token itself) in the session state, fetching the necessary secrets from the secure store when needed.

Journey 1: Building Agentic Applications with Authenticated Tools

This section focuses on using pre-existing tools (like those from `RestApiTool`/`OpenAPIToolset`, `APIHubToolset`, `GoogleApiToolSet`) that require authentication within your agentic application. Your main responsibility is configuring the tools and handling the client-side part of interactive authentication flows (if required by the tool).

1. Configuring Tools with Authentication

When adding an authenticated tool to your agent, you need to provide its required `AuthScheme` and your application's initial `AuthCredential`.

A. Using OpenAPI-based Toolsets (`OpenAPIToolset`, `APIHubToolset`, etc.)

Pass the scheme and credential during toolset initialization. The toolset applies them to all generated tools. Here are few ways to create tools with authentication in ADK.

[API Key](#)

[OAuth2](#)

[Service Account](#)

[OpenID connect](#)

Create a tool requiring an API Key.

```
from google.adk.tools.openapi_tool.auth.auth_helpers import  
token_to_scheme_credential
```

```
from google.adk.tools.apihub_tool.apihub_toolset import APIHubToolset
```

```
auth_scheme, auth_credential = token_to_scheme_credential(
```

```
    "apikey", "query", "apikey", YOUR_API_KEY_STRING  
)  
  
sample_api_toolset = APIHubToolset(  
  
    name="sample-api-requiring-api-key",  
  
    description="A tool using an API protected by API Key",  
  
    apihub_resource_name="...",  
  
    auth_scheme=auth_scheme,  
  
    auth_credential=auth_credential,
```

```
)
```

B. Using Google API Toolsets (e.g., calendar_tool_set)

These toolsets often have dedicated configuration methods.

Tip: For how to create a Google OAuth Client ID & Secret, see this guide: [Get your Google API Client ID](#)

```
# Example: Configuring Google Calendar Tools
```

```
from google.adk.tools.google_api_tool import calendar_tool_set
```

```
client_id = "YOUR_GOOGLE_OAUTH_CLIENT_ID.apps.googleusercontent.com"
```

```
client_secret = "YOUR_GOOGLE_OAUTH_CLIENT_SECRET"
```

```
# Use the specific configure method for this toolset type
```

```
calendar_tool_set.configure_auth()
```

```
    client_id=oauth_client_id, client_secret=oauth_client_secret  
)  
  
# agent = LlmAgent(..., tools=calendar_tool_set.get_tool('calendar_tool_set'))
```

The sequence diagram of auth request flow (where tools are requesting auth credentials) looks like below:



2. Handling the Interactive OAuth/OIDC Flow (Client-Side)

If a tool requires user login/consent (typically OAuth 2.0 or OIDC), the ADK framework pauses execution and signals your **Agent Client** application. There are two cases:

- **Agent Client** application runs the agent directly (via `runner.run_async`) in the same process. e.g. UI backend, CLI app, or Spark job etc.
- **Agent Client** application interacts with ADK's fastapi server via `/run` or `/run_sse` endpoint. While ADK's fastapi server could be setup on the same server or different server as **Agent Client** application

The second case is a special case of first case, because `/run` or `/run_sse` endpoint also invokes `runner.run_async`. The only differences are:

- Whether to call a python function to run the agent (first case) or call a service endpoint to run the agent (second case).
- Whether the result events are in-memory objects (first case) or serialized json string in http response (second case).

Below sections focus on the first case and you should be able to map it to the second case very straightforward. We will also describe some differences to handle for the second case if necessary.

Here's the step-by-step process for your client application:

Step 1: Run Agent & Detect Auth Request

- Initiate the agent interaction using `runner.run_async`.
- Iterate through the yielded events.
- Look for a specific function call event whose function call has a special name: `adk_request_credential`. This event signals that user interaction is needed. You can use helper functions to identify this event and extract necessary information. (For the second case, the logic is similar. You deserialize the event from the http response).

```
# runner = Runner(...)

# session = await session_service.create_session(...)

# content = types.Content(...) # User's initial query

print("\nRunning agent...")

events_async = runner.run_async(
    session_id=session.id, user_id='user', new_message=content
```

```
)
```

```
auth_request_function_call_id, auth_config = None, None
```

```
async for event in events_async:
```

```
    # Use helper to check for the specific auth request event
```

```
    if (auth_request_function_call := get_auth_request_function_call(event)):
```

```
        print("--> Authentication required by agent.")
```

```
        # Store the ID needed to respond later
```

```
        if not (auth_request_function_call_id :=  
auth_request_function_call.id):
```

```
            raise ValueError(f'Cannot get function call id from function call:  
{auth_request_function_call}')
```

```
# Get the AuthConfig containing the auth_uri etc.

auth_config = get_auth_config(auth_request_function_call)

break # Stop processing events for now, need user interaction

if not auth_request_function_call_id:

    print("\nAuth not required or agent finished.")

# return # Or handle final response if received
```

Helper functions helpers.py:

```
from google.adk.events import Event

from google.adk.auth import AuthConfig # Import necessary type

from google.genai import types
```

```
def get_auth_request_function_call(event: Event) -> types.FunctionCall:  
  
    # Get the special auth request function call from the event  
  
    if not event.content or event.content.parts:  
  
        return  
  
    for part in event.content.parts:  
  
        if (  
  
            part  
  
            and part.function_call  
  
            and part.function_call.name == 'adk_request_credential'  
  
            and event.long_running_tool_ids  
  
            and part.function_call.id in event.long_running_tool_ids  
  
        ):
```

```
        return part.function_call

def get_auth_config(auth_request_function_call: types.FunctionCall) ->
AuthConfig:

    # Extracts the AuthConfig object from the arguments of the auth request
    function call

    if not auth_request_function_call.args or not (auth_config := auth_request_function_call.args.get('auth_config')):

        raise ValueError(f'Cannot get auth config from function call:
{auth_request_function_call}')

    if not isinstance(auth_config, AuthConfig):

        raise ValueError(f'Cannot get auth config {auth_config} is not an
instance of AuthConfig.')

    return auth_config
```

Step 2: Redirect User for Authorization

- Get the authorization URL (`auth_uri`) from the `auth_config` extracted in the previous step.
- **Crucially, append your application's `redirect_uri`** as a query parameter to this `auth_uri`. This `redirect_uri` must be pre-registered with your OAuth provider (e.g., [Google Cloud Console](#), [Okta admin panel](#)).
- Direct the user to this complete URL (e.g., open it in their browser).

```
# (Continuing after detecting auth needed)
```

```
if auth_request_function_call_id and auth_config:
```

```
    # Get the base authorization URL from the AuthConfig
```

```
    base_auth_uri = auth_config.exchanged_auth_credential.oauth2.auth_uri
```

```
    if base_auth_uri:
```

```
        redirect_uri = 'http://localhost:8000/callback' # MUST match your OAuth
client app config
```

```
# Append redirect_uri (use urlencode in production)

auth_request_uri = base_auth_uri + f'&redirect_uri={redirect_uri}'

# Now you need to redirect your end user to this auth_request_uri or
ask them to open this auth_request_uri in their browser

# This auth_request_uri should be served by the corresponding auth
provider and the end user should login and authorize your applicaiton to
access their data

# And then the auth provider will redirect the end user to the
redirect_uri you provided

# Next step: Get this callback URL from the user (or your web server
handler)

else:

    print("ERROR: Auth URI not found in auth_config.")

# Handle error
```

Step 3. Handle the Redirect Callback (Client):

- Your application must have a mechanism (e.g., a web server route at the `redirect_uri`) to receive the user after they authorize the application with the provider.
- The provider redirects the user to your `redirect_uri` and appends an `authorization_code` (and potentially `state`, `scope`) as query parameters to the URL.
- Capture the **full callback URL** from this incoming request.
- (This step happens outside the main agent execution loop, in your web server or equivalent callback handler.)

Step 4. Send Authentication Result Back to ADK (Client):

- Once you have the full callback URL (containing the authorization code), retrieve the `auth_request_function_call_id` and the `auth_config` object saved in Client Step 1.
- Set the captured callback URL into the `exchanged_auth_credential.oauth2.auth_response_uri` field. Also ensure `exchanged_auth_credential.oauth2.redirect_uri` contains the redirect URI you used.
- Create a `types.Content` object containing a `types.Part` with a `types.FunctionResponse`.
 - Set `name` to `"adk_request_credential"`. (Note: This is a special name for ADK to proceed with authentication. Do not use other names.)
 - Set `id` to the `auth_request_function_call_id` you saved.
 - Set `response` to the `serialized` (e.g., `.model_dump()`) updated `AuthConfig` object.
- Call `runner.run_async again` for the same session, passing this `FunctionResponse` content as the `new_message`.

```
# (Continuing after user interaction)
```

```
# Simulate getting the callback URL (e.g., from user paste or web handler)
```

```
auth_response_uri = await get_user_input(  
    f'Paste the full callback URL here:\n> '  
)  
  
auth_response_uri = auth_response_uri.strip() # Clean input  
  
if not auth_response_uri:  
    print("Callback URL not provided. Aborting.")  
  
    return  
  
# Update the received AuthConfig with the callback details  
  
    auth_config.exchanged_auth_credential.oauth2.auth_response_uri =  
    auth_response_uri
```

```
# Also include the redirect_uri used, as the token exchange might need it

auth_config.exchanged_auth_credential.oauth2.redirect_uri = redirect_uri

# Construct the FunctionResponse Content object

auth_content = types.Content(
    role='user', # Role can be 'user' when sending a FunctionResponse

    parts=[

        types.Part(
            function_response=types.FunctionResponse(
                id=auth_request_function_call_id, # Link to the
                original_request

                name='adk_request_credential', # Special framework function
                name
```

```
        response=auth_config.model_dump() # Send back the *updated*
AuthConfig

    )

    )

],
)

# --- Resume Execution ---

print("\nSubmitting authentication details back to the agent...")

events_async_after_auth = runner.run_async(
    session_id=session.id,
    user_id='user',
```

```

        new_message=auth_content, # Send the FunctionResponse back

    )

# --- Process Final Agent Output ---

print("\n--- Agent Response after Authentication ---")

async for event in events_async_after_auth:

    # Process events normally, expecting the tool call to succeed now

    print(event) # Print the full event for inspection

```

Step 5: ADK Handles Token Exchange & Tool Retry and gets Tool result

- ADK receives the `FunctionResponse` for `adk_request_credential`.
- It uses the information in the updated `AuthConfig` (including the callback URL containing the code) to perform the OAuth **token exchange** with the provider's token endpoint, obtaining the access token (and possibly refresh token).
- ADK internally makes these tokens available by setting them in the session state).
- ADK **automatically retries** the original tool call (the one that initially failed due to missing auth).

- This time, the tool finds the valid tokens (via `tool_context.get_auth_response()`) and successfully executes the authenticated API call.
 - The agent receives the actual result from the tool and generates its final response to the user.
-

The sequence diagram of auth response flow (where Agent Client send back the auth response and ADK retries tool calling) looks like below:



Journey 2: Building Custom Tools (`FunctionTool`) Requiring Authentication

This section focuses on implementing the authentication logic *inside* your custom Python function when creating a new ADK Tool. We will implement a `FunctionTool` as an example.

Prerequisites

Your function signature *must* include `tool_context: ToolContext`. ADK automatically injects this object, providing access to state and auth mechanisms.

```
from google.adk.tools import FunctionTool, ToolContext
```

```
from typing import Dict
```

```
def my_authenticated_tool_function(param1: str, ..., tool_context:  
ToolContext) -> dict:  
  
    # ... your logic ...  
  
    pass  
  
  
  
my_tool = FunctionTool(func=my_authenticated_tool_function)
```

Authentication Logic within the Tool Function

Implement the following steps inside your function:

Step 1: Check for Cached & Valid Credentials:

Inside your tool function, first check if valid credentials (e.g., access/refresh tokens) are already stored from a previous run in this session. Credentials for the current sessions should be stored in `tool_context.invocation_context.session.state` (a dictionary of state). Check existence of existing credentials by checking `tool_context.invocation_context.session.state.get(credential_name, None)`.

```
from google.oauth2.credentials import Credentials
```

```
from google.auth.transport.requests import Request

# Inside your tool function

TOKEN_CACHE_KEY = "my_tool_tokens" # Choose a unique key

SCOPES = ["scope1", "scope2"] # Define required scopes

creds = None

cached_token_info = tool_context.state.get(TOKEN_CACHE_KEY)

if cached_token_info:

    try:

        creds = Credentials.from_authorized_user_info(cached_token_info,
SCOPES)
```

```
    if not creds.valid and creds.expired and creds.refresh_token:

        creds.refresh(Request())

    tool_context.state[TOKEN_CACHE_KEY] = json.loads(creds.to_json()) # Update cache

    elif not creds.valid:

        creds = None # Invalid, needs re-auth

    tool_context.state[TOKEN_CACHE_KEY] = None

except Exception as e:

    print(f"Error loading/refreshing cached creds: {e}")

    creds = None

    tool_context.state[TOKEN_CACHE_KEY] = None
```

```
if creds and creds.valid:  
  
    # Skip to Step 5: Make Authenticated API Call  
  
    pass  
  
else:  
  
    # Proceed to Step 2...  
  
    pass
```

Step 2: Check for Auth Response from Client

- If Step 1 didn't yield valid credentials, check if the client just completed the interactive flow by calling `exchanged_credential = tool_context.get_auth_response()`.
- This returns the updated `exchanged_credential` object sent back by the client (containing the callback URL in `auth_response_uri`).

```
# Use auth_scheme and auth_credential configured in the tool.  
  
# exchanged_credential: AuthCredential | None
```

```
exchanged_credential = tool_context.get_auth_response(AuthConfig(  
  
    auth_scheme=auth_scheme,  
  
    raw_auth_credential=auth_credential,  
  
))  
  
# If exchanged_credential is not None, then there is already an exchanged  
# credential from the auth response.  
  
if exchanged_credential:  
  
    # ADK exchanged the access token already for us  
  
    access_token = exchanged_credential.oauth2.access_token  
  
    refresh_token = exchanged_credential.oauth2.refresh_token  
  
    creds = Credentials(  
  
        token=access_token,
```

```
        refresh_token=refresh_token,  
  
        token_uri=auth_scheme.flows.authorizationCode.tokenUrl,  
  
        client_id=auth_credential.oauth2.client_id,  
  
        client_secret=auth_credential.oauth2.client_secret,  
  
        scopes=list(auth_scheme.flows.authorizationCode.scopes.keys()),  
  
    )  
  
    # Cache the token in session state and call the API, skip to step 5
```

Step 3: Initiate Authentication Request

If no valid credentials (Step 1.) and no auth response (Step 2.) are found, the tool needs to start the OAuth flow. Define the AuthScheme and initial AuthCredential and call `tool_context.request_credential()`. Return a response indicating authorization is needed.

```
# Use auth_scheme and auth_credential configured in the tool.
```

```
tool_context.request_credential(AuthConfig(
```

```
    auth_scheme=auth_scheme,  
  
    raw_auth_credential=auth_credential,  
  
))  
  
return {'pending': true, 'message': 'Awaiting user authentication.'}  
  
# By setting request_credential, ADK detects a pending authentication event.  
It pauses execution and ask end user to login.
```

Step 4: Exchange Authorization Code for Tokens

ADK automatically generates oauth authorization URL and presents it to your Agent Client application. your Agent Client application should follow the same way described in Journey 1 to redirect the user to the authorization URL (with `redirect_uri` appended). Once a user completes the login flow following the authorization URL and ADK extracts the authentication callback url from Agent Client applications, automatically parses the auth code, and generates auth token. At the next Tool call, `tool_context.get_auth_response` in step 2 will contain a valid credential to use in subsequent API calls.

Step 5: Cache Obtained Credentials

After successfully obtaining the token from ADK (Step 2) or if the token is still valid (Step 1), **immediately store** the new `Credentials` object in `tool_context.state` (serialized, e.g., as JSON) using your cache key.

```
# Inside your tool function, after obtaining 'creds' (either refreshed or
# newly exchanged)

# Cache the new/refreshed tokens

tool_context.state[TOKEN_CACHE_KEY] = json.loads(creds.to_json())

print(f"DEBUG: Cached/updated tokens under key: {TOKEN_CACHE_KEY}")

# Proceed to Step 6 (Make API Call)
```

Step 6: Make Authenticated API Call

- Once you have a valid `Credentials` object (`creds` from Step 1 or Step 4), use it to make the actual call to the protected API using the appropriate client library (e.g., `googleapiclient`, `requests`). Pass the `credentials=creds` argument.
- Include error handling, especially for `HttpError 401/403`, which might mean the token expired or was revoked between calls. If you get such an error, consider clearing the cached token (`tool_context.state.pop(...)`) and potentially returning the `auth_required` status again to force re-authentication.

```
# Inside your tool function, using the valid 'creds' object

# Ensure creds is valid before proceeding

if not creds or not creds.valid:
```

```
    return {"status": "error", "error_message": "Cannot proceed without valid
credentials."}

try:

    service = build("calendar", "v3", credentials=creds) # Example

    api_result = service.events().list(...).execute()

    # Proceed to Step 7

except Exception as e:

    # Handle API errors (e.g., check for 401/403, maybe clear cache and
    re-request auth)

    print(f"ERROR: API call failed: {e}")

    return {"status": "error", "error_message": f"API call failed: {e}"}
```

Step 7: Return Tool Result

- After a successful API call, process the result into a dictionary format that is useful for the LLM.
- **Crucially, include a** along with the data.

```
# Inside your tool function, after successful API call
```

```
    processed_result = [...] # Process api_result for the LLM

    return {"status": "success", "data": processed_result}
```

Runtime

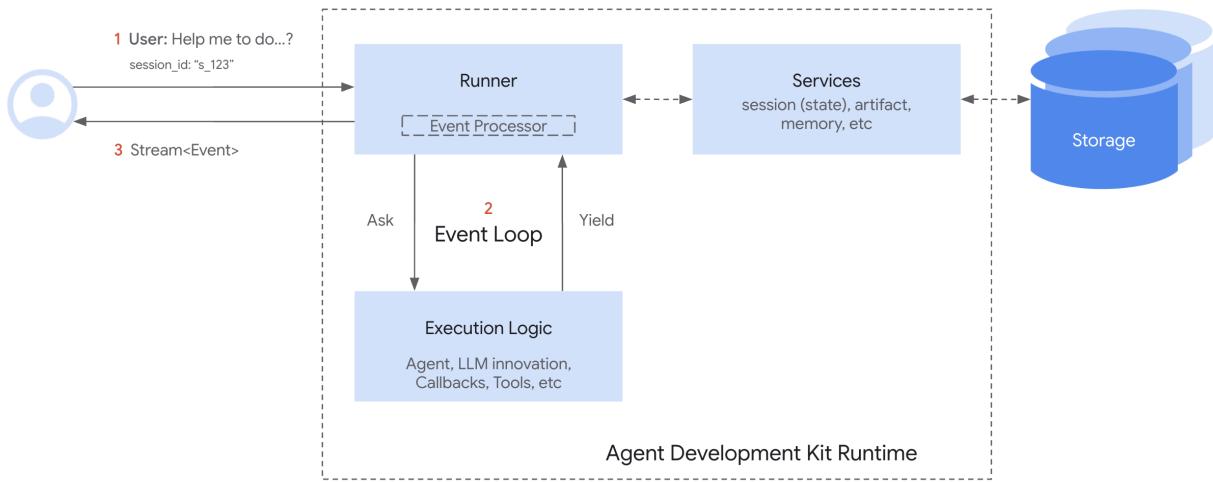
What is runtime?

The ADK Runtime is the underlying engine that powers your agent application during user interactions. It's the system that takes your defined agents, tools, and callbacks and orchestrates their execution in response to user input, managing the flow of information, state changes, and interactions with external services like LLMs or storage.

Think of the Runtime as the "**engine**" of your agentic application. You define the parts (agents, tools), and the Runtime handles how they connect and run together to fulfill a user's request.

Core Idea: The Event Loop

At its heart, the ADK Runtime operates on an **Event Loop**. This loop facilitates a back-and-forth communication between the `Runner` component and your defined "Execution Logic" (which includes your Agents, the LLM calls they make, Callbacks, and Tools).



In simple terms:

1. The **Runner** receives a user query and asks the main **Agent** to start processing.
2. The **Agent** (and its associated logic) runs until it has something to report (like a response, a request to use a tool, or a state change) – it then **yields** or **emits** an **Event**.
3. The **Runner** receives this **Event**, processes any associated actions (like saving state changes via **Services**), and forwards the event onwards (e.g., to the user interface).
4. Only *after* the **Runner** has processed the event does the **Agent's** logic **resume** from where it paused, now potentially seeing the effects of the changes committed by the **Runner**.
5. This cycle repeats until the agent has no more events to yield for the current user query.

This event-driven loop is the fundamental pattern governing how ADK executes your agent code.

The Heartbeat: The Event Loop - Inner workings

The Event Loop is the core operational pattern defining the interaction between the `Runner` and your custom code (Agents, Tools, Callbacks, collectively referred to as "Execution Logic" or "Logic Components" in the design document). It establishes a clear division of responsibilities:

Note

The specific method names and parameter names may vary slightly by SDK language (e.g., `agent_to_run.runAsync(...)` in Java, `agent_to_run.run_async(...)` in Python). Refer to the language-specific API documentation for details.

Runner's Role (Orchestrator)

The `Runner` acts as the central coordinator for a single user invocation. Its responsibilities in the loop are:

1. **Initiation:** Receives the end user's query (`new_message`) and typically appends it to the session history via the `SessionService`.
2. **Kick-off:** Starts the event generation process by calling the main agent's execution method (e.g., `agent_to_run.run_async(...)`).
3. **Receive & Process:** Waits for the agent logic to `yield` or `emit` an `Event`. Upon receiving an event, the `Runner` **promptly processes** it. This involves:
 - Using configured Services (`SessionService`, `ArtifactService`, `MemoryService`) to commit changes indicated in `event.actions` (like `state_delta`, `artifact_delta`).
 - Performing other internal bookkeeping.
4. **Yield Upstream:** Forwards the processed event onwards (e.g., to the calling application or UI for rendering).
5. **Iterate:** Signals the agent logic that processing is complete for the yielded event, allowing it to resume and generate the *next* event.

Conceptual Runner Loop:

Python

Java

```
# Simplified view of Runner's main loop logic
```

```
def run(new_query, ...) -> Generator[Event]:  
  
    # 1. Append new_query to session event history (via SessionService)  
  
        session_service.append_event(session, Event(author='user',  
content=new_query))  
  
    # 2. Kick off event loop by calling the agent  
  
        agent_event_generator = agent_to_run.run_async(context)  
  
        async for event in agent_event_generator:  
  
            # 3. Process the generated event and commit changes  
  
                session_service.append_event(session, event) # Commits state/artifact  
deltas etc.  
  
                # memory_service.update_memory(...) # If applicable
```

```
# artifact_service might have already been called via context during  
agent run
```

```
# 4. Yield event for upstream processing (e.g., UI rendering)
```

```
yield event
```

```
# Runner implicitly signals agent generator can continue after yielding
```

Execution Logic's Role (Agent, Tool, Callback)

Your code within agents, tools, and callbacks is responsible for the actual computation and decision-making. Its interaction with the loop involves:

1. **Execute:** Runs its logic based on the current `InvocationContext`, including the session state *as it was when execution resumed*.
2. **Yield:** When the logic needs to communicate (send a message, call a tool, report a state change), it constructs an `Event` containing the relevant content and actions, and then `yields` this event back to the `Runner`.
3. **Pause:** Crucially, execution of the agent logic **pauses immediately** after the `yield` statement (or `return` in RxJava). It waits for the `Runner` to complete step 3 (processing and committing).
4. **Resume:** *Only after* the `Runner` has processed the yielded event does the agent logic resume execution from the statement immediately following the `yield`.
5. **See Updated State:** Upon resumption, the agent logic can now reliably access the session state (`ctx.session.state`) reflecting the changes that were committed by the `Runner` from the *previously yielded* event.

Conceptual Execution Logic:

Python

Java

```
# Simplified view of logic inside Agent.run_async, callbacks, or tools

# ... previous code runs based on current state ...

# 1. Determine a change or output is needed, construct the event

# Example: Updating state

update_data = {'field_1': 'value_2'}

event_with_state_change = Event(
    author=author,
    actions=EventActions(state_delta=update_data),
    content=types.Content(parts=[types.Part(text="State updated.")]))
```

```
# ... other event fields ...

)

# 2. Yield the event to the Runner for processing & commit

yield event_with_state_change

# <<<<<<<<< EXECUTION PAUSES HERE >>>>>>>>>

# <<<<<<<<< RUNNER PROCESSES & COMMITS THE EVENT >>>>>>>>>

# 3. Resume execution ONLY after Runner is done processing the above event.

# Now, the state committed by the Runner is reliably reflected.
```

```
# Subsequent code can safely assume the change from the yielded event  
happened.
```

```
val = ctx.session.state['field_1']
```

```
# here `val` is guaranteed to be "value_2" (assuming Runner committed  
successfully)
```

```
print(f'Resumed execution. Value of field_1 is now: {val}')
```

```
# ... subsequent code continues ...
```

```
# Maybe yield another event later...
```

This cooperative yield/pause/resume cycle between the `Runner` and your Execution Logic, mediated by `Event` objects, forms the core of the ADK Runtime.

Key components of the Runtime

Several components work together within the ADK Runtime to execute an agent invocation. Understanding their roles clarifies how the event loop functions:

1. `Runner`

- **Role:** The main entry point and orchestrator for a single user query (`run_async`).
- **Function:** Manages the overall Event Loop, receives events yielded by the Execution Logic, coordinates with Services to process and commit event actions (state/artifact changes), and forwards processed events upstream (e.g., to the UI). It essentially drives the conversation turn by turn based on yielded events. (Defined in `google.adk.runners.runner`).

2. Execution Logic Components

- **Role:** The parts containing your custom code and the core agent capabilities.
- **Components:**
 - **Agent** (`BaseAgent`, `LlmAgent`, etc.): Your primary logic units that process information and decide on actions. They implement the `_run_async_impl` method which yields events.
 - **Tools** (`BaseTool`, `FunctionTool`, `AgentTool`, etc.): External functions or capabilities used by agents (often `LlmAgent`) to interact with the outside world or perform specific tasks. They execute and return results, which are then wrapped in events.
 - **Callbacks (Functions):** User-defined functions attached to agents (e.g., `before_agent_callback`, `after_model_callback`) that hook into specific points in the execution flow, potentially modifying behavior or state, whose effects are captured in events.
 - **Function:** Perform the actual thinking, calculation, or external interaction. They communicate their results or needs by **yielding Event objects** and pausing until the Runner processes them.

3. Event

- **Role:** The message passed back and forth between the `Runner` and the Execution Logic.
- **Function:** Represents an atomic occurrence (user input, agent text, tool call/result, state change request, control signal). It carries both the content of the occurrence and the intended side effects (`actions` like `state_delta`).

4. Services

- **Role:** Backend components responsible for managing persistent or shared resources. Used primarily by the `Runner` during event processing.
- **Components:**

- SessionService (BaseSessionService, InMemorySessionService, etc.): Manages Session objects, including saving/loading them, applying state_delta to the session state, and appending events to the event history.
- ArtifactService (BaseArtifactService, InMemoryArtifactService, GcsArtifactService, etc.): Manages the storage and retrieval of binary artifact data. Although save_artifact is called via context during execution logic, the artifact_delta in the event confirms the action for the Runner/SessionService.
- MemoryService (BaseMemoryService, etc.): (Optional) Manages long-term semantic memory across sessions for a user.
- **Function:** Provide the persistence layer. The Runner interacts with them to ensure changes signaled by event.actions are reliably stored *before* the Execution Logic resumes.

5. Session

- **Role:** A data container holding the state and history for *one specific conversation* between a user and the application.
- **Function:** Stores the current state dictionary, the list of all past events (event history), and references to associated artifacts. It's the primary record of the interaction, managed by the SessionService.

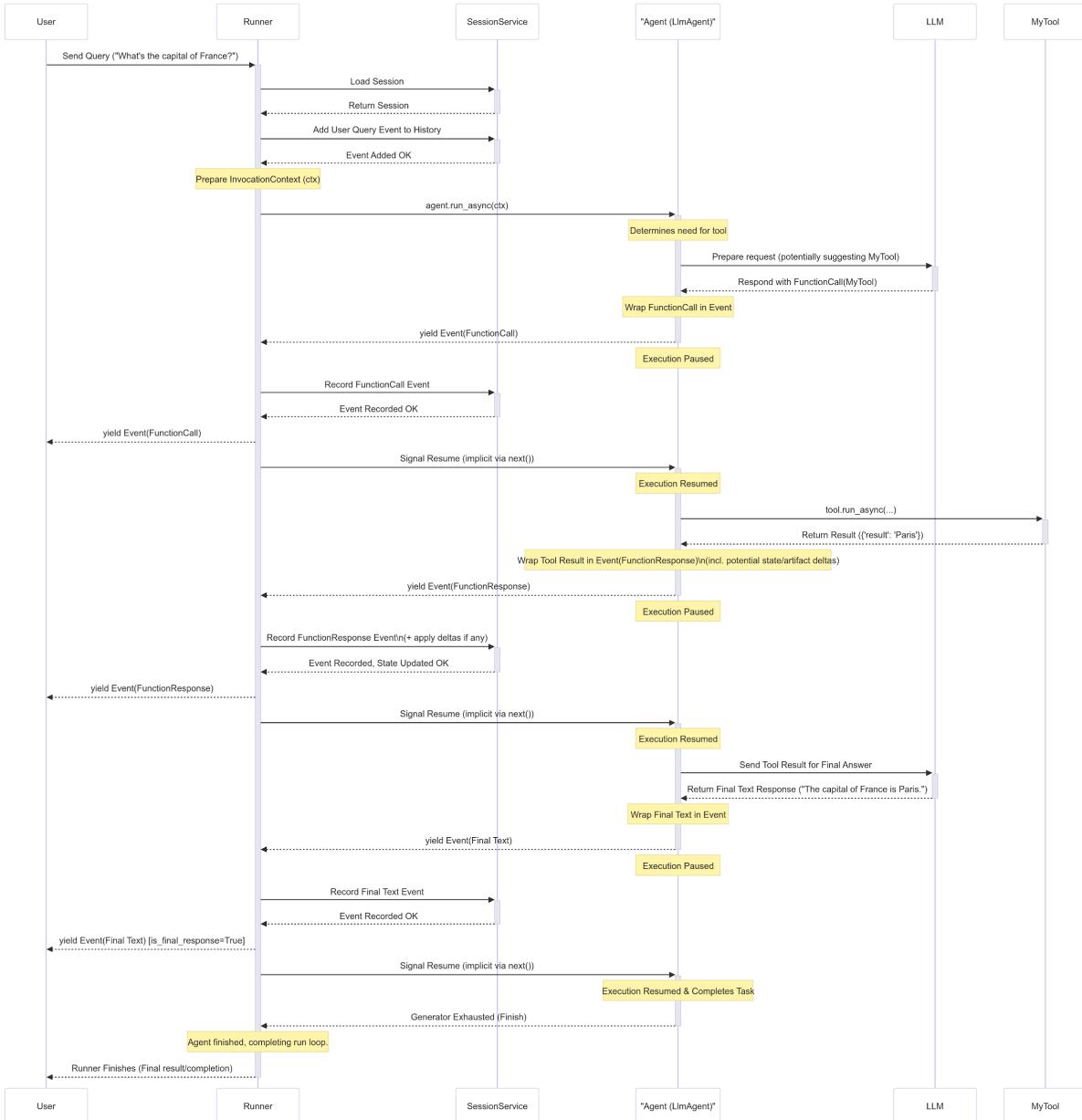
6. Invocation

- **Role:** A conceptual term representing everything that happens in response to a *single* user query, from the moment the Runner receives it until the agent logic finishes yielding events for that query.
- **Function:** An invocation might involve multiple agent runs (if using agent transfer or AgentTool), multiple LLM calls, tool executions, and callback executions, all tied together by a single invocation_id within the InvocationContext.

These players interact continuously through the Event Loop to process a user's request.

How It Works: A Simplified Invocation

Let's trace a simplified flow for a typical user query that involves an LLM agent calling a tool:



Step-by-Step Breakdown

- User Input:** The User sends a query (e.g., "What's the capital of France?").
- Runner Starts:** Runner.run_async begins. It interacts with the SessionService to load the relevant Session and adds the user query as the first Event to the session history. An InvocationContext (ctx) is prepared.

3. **Agent Execution:** The Runner calls `agent.run_async(ctx)` on the designated root agent (e.g., an `LlmAgent`).
4. **LLM Call (Example):** The `Agent_Llm` determines it needs information, perhaps by calling a tool. It prepares a request for the `LLM`. Let's assume the `LLM` decides to call `MyTool`.
5. **Yield FunctionCall Event:** The `Agent_Llm` receives the `FunctionCall` response from the `LLM`, wraps it in an `Event(author='Agent_Llm', content=Content(parts=[Part(function_call=...)]))`, and yields or emits this event.
6. **Agent Pauses:** The `Agent_Llm`'s execution pauses immediately after the `yield`.
7. **Runner Processes:** The `Runner` receives the `FunctionCall` event. It passes it to the `SessionService` to record it in the history. The `Runner` then yields the event upstream to the `User` (or application).
8. **Agent Resumes:** The `Runner` signals that the event is processed, and `Agent_Llm` resumes execution.
9. **Tool Execution:** The `Agent_Llm`'s internal flow now proceeds to execute the requested `MyTool`. It calls `tool.run_async(...)`.
10. **Tool Returns Result:** `MyTool` executes and returns its result (e.g., `{'result': 'Paris'}`).
11. **Yield FunctionResponse Event:** The agent (`Agent_Llm`) wraps the tool result into an `Event` containing a `FunctionResponse` part (e.g., `Event(author='Agent_Llm', content=Content(role='user', parts=[Part(function_response=...)]))`). This event might also contain actions if the tool modified state (`state_delta`) or saved artifacts (`artifact_delta`). The agent `yields` this event.
12. **Agent Pauses:** `Agent_Llm` pauses again.
13. **Runner Processes:** `Runner` receives the `FunctionResponse` event. It passes it to `SessionService` which applies any `state_delta`/`artifact_delta` and adds the event to history. `Runner` yields the event upstream.
14. **Agent Resumes:** `Agent_Llm` resumes, now knowing the tool result and any state changes are committed.
15. **Final LLM Call (Example):** `Agent_Llm` sends the tool result back to the `LLM` to generate a natural language response.
16. **Yield Final Text Event:** `Agent_Llm` receives the final text from the `LLM`, wraps it in an `Event(author='Agent_Llm', content=Content(parts=[Part(text=...)]))`, and yields it.
17. **Agent Pauses:** `Agent_Llm` pauses.

18. **Runner Processes:** `Runner` receives the final text event, passes it to `SessionService` for history, and yields it upstream to the `User`. This is likely marked as the `is_final_response()`.
19. **Agent Resumes & Finishes:** `Agent_Llm` resumes. Having completed its task for this invocation, its `run_async` generator finishes.
20. **Runner Completes:** The `Runner` sees the agent's generator is exhausted and finishes its loop for this invocation.

This yield/pause/process/resume cycle ensures that state changes are consistently applied and that the execution logic always operates on the most recently committed state after yielding an event.

Important Runtime Behaviors

Understanding a few key aspects of how the ADK Runtime handles state, streaming, and asynchronous operations is crucial for building predictable and efficient agents.

State Updates & Commitment Timing

- **The Rule:** When your code (in an agent, tool, or callback) modifies the session state (e.g., `context.state['my_key'] = 'new_value'`), this change is initially recorded locally within the current `InvocationContext`. The change is only **guaranteed to be persisted** (saved by the `SessionService`) *after* the Event carrying the corresponding `state_delta` in its `actions` has been `yield-ed` by your code and subsequently processed by the `Runner`.
- **Implication:** Code that runs *after* resuming from a `yield` can reliably assume that the state changes signaled in the *yielded* event have been committed.

Python

Java

```
# Inside agent logic (conceptual)
```

```
# 1. Modify state
```

```
ctx.session.state['status'] = 'processing'
```

```
event1 = Event(..., actions=EventActions(state_delta={'status':  
'processing'}))
```

```
# 2. Yield event with the delta
```

```
yield event1
```

```
# --- PAUSE --- Runner processes event1, SessionService commits 'status' =  
'processing' ---
```

```
# 3. Resume execution
```

```
# Now it's safe to rely on the committed state
```

```
current_status = ctx.session.state['status'] # Guaranteed to be 'processing'
```

```
print(f"Status after resuming: {current_status}")
```

"Dirty Reads" of Session State

- **Definition:** While commitment happens *after* the yield, code running *later within the same invocation*, but *before* the state-changing event is actually yielded and processed, **can often see the local, uncommitted changes**. This is sometimes called a "dirty read".
- **Example:**

Python

Java

```
# Code in before_agent_callback
```

```
callback_context.state['field_1'] = 'value_1'
```

```
# State is locally set to 'value_1', but not yet committed by Runner
```

```
# ... agent runs ...
```

```
# Code in a tool called later *within the same invocation*
```

```

# Readable (dirty read), but 'value_1' isn't guaranteed persistent yet.

val = tool_context.state['field_1'] # 'val' will likely be 'value_1' here

print(f"Dirty read value in tool: {val}")

# Assume the event carrying the state_delta={'field_1': 'value_1'}

# is yielded *after* this tool runs and is processed by the Runner.

```

- **Implications:**
- **Benefit:** Allows different parts of your logic within a single complex step (e.g., multiple callbacks or tool calls before the next LLM turn) to coordinate using state without waiting for a full yield/commit cycle.
- **Caveat:** Relying heavily on dirty reads for critical logic can be risky. If the invocation fails *before* the event carrying the `state_delta` is yielded and processed by the `Runner`, the uncommitted state change will be lost. For critical state transitions, ensure they are associated with an event that gets successfully processed.

Streaming vs. Non-Streaming Output (`partial=True`)

This primarily relates to how responses from the LLM are handled, especially when using streaming generation APIs.

- **Streaming:** The LLM generates its response token-by-token or in small chunks.
- The framework (often within `BaseLlmFlow`) yields multiple `Event` objects for a single conceptual response. Most of these events will have `partial=True`.
- The `Runner`, upon receiving an event with `partial=True`, typically **forwards it immediately upstream** (for UI display) but **skips processing its actions** (like `state_delta`).
- Eventually, the framework yields a final event for that response, marked as non-partial (`partial=False` or implicitly via `turn_complete=True`).
- The `Runner` **fully processes only this final event**, committing any associated `state_delta` or `artifact_delta`.
- **Non-Streaming:** The LLM generates the entire response at once. The framework yields a single event marked as non-partial, which the `Runner` processes fully.
- **Why it Matters:** Ensures that state changes are applied atomically and only once based on the *complete* response from the LLM, while still allowing the UI to display text progressively as it's generated.

Async is Primary (`run_async`)

- **Core Design:** The ADK Runtime is fundamentally built on asynchronous libraries (like Python's `asyncio` and Java's `RxJava`) to handle concurrent operations (like waiting for LLM responses or tool executions) efficiently without blocking.
- **Main Entry Point:** `Runner.run_async` is the primary method for executing agent invocations. All core runnable components (Agents, specific flows) use `asynchronous` methods internally.
- **Synchronous Convenience (`run`):** A synchronous `Runner.run` method exists mainly for convenience (e.g., in simple scripts or testing environments). However, internally, `Runner.run` typically just calls `Runner.run_async` and manages the async event loop execution for you.
- **Developer Experience:** We recommend designing your applications (e.g., web servers using ADK) to be asynchronous for best performance. In Python, this means using `asyncio`; in Java, leverage `RxJava`'s reactive programming model.
- **Sync Callbacks/Tools:** The ADK framework supports both asynchronous and synchronous functions for tools and callbacks.

- **Blocking I/O:** For long-running synchronous I/O operations, the framework attempts to prevent stalls. Python ADK may use `asyncio.to_thread`, while Java ADK often relies on appropriate RxJava schedulers or wrappers for blocking calls.
- **CPU-Bound Work:** Purely CPU-intensive synchronous tasks will still block their execution thread in both environments.

Understanding these behaviors helps you write more robust ADK applications and debug issues related to state consistency, streaming updates, and asynchronous execution.

Runtime Configuration

`RunConfig` defines runtime behavior and options for agents in the ADK. It controls speech and streaming settings, function calling, artifact saving, and limits on LLM calls.

When constructing an agent run, you can pass a `RunConfig` to customize how the agent interacts with models, handles audio, and streams responses. By default, no streaming is enabled and inputs aren't retained as artifacts. Use `RunConfig` to override these defaults.

Class Definition

The `RunConfig` class holds configuration parameters for an agent's runtime behavior.

- Python ADK uses Pydantic for this validation.
- Java ADK typically uses immutable data classes.

Python

Java

```
class RunConfig(BaseModel):
```

```
    """Configs for runtime behavior of agents."""
```

```
    model_config = ConfigDict(  
  
        extra='forbid',  
  
    )  
  
    speech_config: Optional[types.SpeechConfig] = None  
  
    response_modalities: Optional[list[str]] = None  
  
    save_input_blobs_as_artifacts: bool = False  
  
    support_cfc: bool = False  
  
    streaming_mode: StreamingMode = StreamingMode.NONE  
  
    output_audio_transcription: Optional[types.AudioTranscriptionConfig] = None
```

```
max_llm_calls: int = 500
```

Runtime Parameters

Parameter	Python Type	Java Type	Default (Py / Java)	Description
speech_config	Optional[types.Spe echConfig]	SpeechConf ig (nullable via @Nullable)	None / null	Configures speech synthesis (voice, language) using the SpeechConfig type.
response_modalities	Optional[list[str]]	ImmutableL ist<Modali ty>	None / Empty Immutabl eList	List of desired output modalities (e.g., Python: ["TEXT", "AUDIO"]; Java: uses structured Modality objects).

<code>save_input_blobs_as_artifacts</code>	<code>bool</code>	<code>boolean</code>	<code>False / false</code>	If <code>true</code> , saves input blobs (e.g., uploaded files) as run artifacts for debugging/auditing.
--	-------------------	----------------------	----------------------------	--

<code>streaming_mode</code>	<code>StreamingMode</code>	<i>Currently not supported</i>	<code>StreamingMode.NONE / N/A</code>	Sets the streaming behavior: <code>NONE</code> (default), <code>SSE</code> (server-sent events), or <code>BIDI</code> (bidirectional).
-----------------------------	----------------------------	--------------------------------	---------------------------------------	--

<code>output_audio_transcription_on</code>	<code>Optional[types.AudioTranscriptionConfig]</code>	<code>AudioTranscriptionConfig</code> (nullable via <code>@Nullable</code>)	<code>None / null</code>	Configures transcription of generated audio output using the <code>AudioTranscriptionConfig</code> type.
--	---	--	--------------------------	--

<code>max_llm_calls</code>	<code>int</code>	<code>int</code>	<code>500 / 500</code>	Limits total LLM calls per run. <code>0</code> or negative means unlimited (warned); <code>sys.maxsize</code> raises <code>ValueError</code> .
----------------------------	------------------	------------------	------------------------	--

support_cfc	bool	<i>Currently not supported</i>	False / N/A	Python: Enables Compositional Function Calling. Requires streaming_mode=SSE and uses the LIVE API. Experimental.
-------------	------	--------------------------------	-------------	---

speech_config

Note

The interface or definition of `SpeechConfig` is the same, irrespective of the language.

Speech configuration settings for live agents with audio capabilities. The `SpeechConfig` class has the following structure:

```
class SpeechConfig(_common.BaseModel):  
  
    """The speech generation configuration."""  
  
    voice_config: Optional[VoiceConfig] = Field(  
  
        default=None,
```

```
        description="""The configuration for the speaker to use."""" ,  
    )  
  
    language_code: Optional[str] = Field(  
  
        default=None,  
  
        description="""Language code (ISO 639. e.g. en-US) for the speech  
synthesization.  
  
        Only available for Live API."""" ,  
    )
```

The `voice_config` parameter uses the `VoiceConfig` class:

```
class VoiceConfig(_common.BaseModel):  
  
    """The configuration for the voice to use."  
  
    prebuilt_voice_config: Optional[PrebuiltVoiceConfig] = Field(  
        description="""The configuration for the voice to use."  
    )
```

```
    default=None,  
  
    description="""The configuration for the speaker to use.""",  
  
)
```

And PrebuiltVoiceConfig has the following structure:

```
class PrebuiltVoiceConfig(_common.BaseModel):  
  
    """The configuration for the prebuilt speaker to use."""  
  
    voice_name: Optional[str] = Field(  
  
        default=None,  
  
        description="""The name of the prebuilt voice to use."""),  
  
)
```

These nested configuration classes allow you to specify:

- `voice_config`: The name of the prebuilt voice to use (in the `PrebuiltVoiceConfig`)
- `language_code`: ISO 639 language code (e.g., "en-US") for speech synthesis

When implementing voice-enabled agents, configure these parameters to control how your agent sounds when speaking.

`response_modalities`

Defines the output modalities for the agent. If not set, defaults to AUDIO. Response modalities determine how the agent communicates with users through various channels (e.g., text, audio).

`save_input_blobs_as_artifacts`

When enabled, input blobs will be saved as artifacts during agent execution. This is useful for debugging and audit purposes, allowing developers to review the exact data received by agents.

`support_cfc`

Enables Compositional Function Calling (CFC) support. Only applicable when using `StreamingMode.SSE`. When enabled, the LIVE API will be invoked as only it supports CFC functionality.

Warning

The `support_cfc` feature is experimental and its API or behavior might change in future releases.

`streaming_mode`

Configures the streaming behavior of the agent. Possible values:

- `StreamingMode.NONE`: No streaming; responses delivered as complete units
- `StreamingMode.SSE`: Server-Sent Events streaming; one-way streaming from server to client
- `StreamingMode.BIDI`: Bidirectional streaming; simultaneous communication in both directions

Streaming modes affect both performance and user experience. SSE streaming lets users see partial responses as they're generated, while BIDI streaming enables real-time interactive experiences.

`output_audio_transcription`

Configuration for transcribing audio outputs from live agents with audio response capability. This enables automatic transcription of audio responses for accessibility, record-keeping, and multi-modal applications.

`max_llm_calls`

Sets a limit on the total number of LLM calls for a given agent run.

- Values greater than 0 and less than `sys.maxsize`: Enforces a bound on LLM calls
- Values less than or equal to 0: Allows unbounded LLM calls (*not recommended for production*)

This parameter prevents excessive API usage and potential runaway processes. Since LLM calls often incur costs and consume resources, setting appropriate limits is crucial.

Validation Rules

The `RunConfig` class validates its parameters to ensure proper agent operation. While Python ADK uses `Pydantic` for automatic type validation, Java ADK relies on its static typing and may include explicit checks in the `RunConfig`'s construction. For the `max_llm_calls` parameter specifically:

1. Extremely large values (like `sys.maxsize` in Python or `Integer.MAX_VALUE` in Java) are typically disallowed to prevent issues.
2. Values of zero or less will usually trigger a warning about unlimited LLM interactions.

Examples

Basic runtime configuration

Python

Java

```
from google.genai.adk import RunConfig, StreamingMode

config = RunConfig(
    streaming_mode=StreamingMode.NONE,
    max_llm_calls=100
)
```

This configuration creates a non-streaming agent with a limit of 100 LLM calls, suitable for simple task-oriented agents where complete responses are preferable.

Enabling streaming

Python

Java

```
from google.genai.adk import RunConfig, StreamingMode

config = RunConfig(
    streaming_mode=StreamingMode.SSE,
    max_llm_calls=200
)
```

Using SSE streaming allows users to see responses as they're generated, providing a more responsive feel for chatbots and assistants.

Enabling speech support

Python

Java

```
from google.genai.adk import RunConfig, StreamingMode
```

```
from google.genai import types

config = RunConfig(
    speech_config=types.SpeechConfig(
        language_code="en-US",
        voice_config=types.VoiceConfig(
            prebuilt_voice_config=types.PrebuiltVoiceConfig(
                voice_name="Kore"
            )
        ),
        response_modalities=["AUDIO", "TEXT"],
    ),
)
```

```
    save_input_blobs_as_artifacts=True,  
  
    support_cfc=True,  
  
    streaming_mode=StreamingMode.SSE,  
  
    max_llm_calls=1000,  
)  
)
```

This comprehensive example configures an agent with:

- Speech capabilities using the "Kore" voice (US English)
- Both audio and text output modalities
- Artifact saving for input blobs (useful for debugging)
- Experimental CFC support enabled (**Python only**)
- SSE streaming for responsive interaction
- A limit of 1000 LLM calls

Enabling Experimental CFC Support



```
from google.genai.adk import RunConfig, StreamingMode
```

```
config = RunConfig(  
    streaming_mode=StreamingMode.SSE,  
  
    support_cfc=True,  
  
    max_llm_calls=150  
)
```

Enabling Compositional Function Calling creates an agent that can dynamically execute functions based on model outputs, powerful for applications requiring complex workflows.

Deploy to Vertex AI Agent Engine



[Agent Engine](#) is a fully managed Google Cloud service enabling developers to deploy, manage, and scale AI agents in production. Agent Engine handles the infrastructure to scale agents in production so you can focus on creating intelligent and impactful applications.

```
from vertexai import agent_engines
```

```
remote_app = agent_engines.create(  
    agent_engine=root_agent,  
  
    requirements=[  
  
        "google-cloud-aiplatform[adk,agent_engines]",  
  
    ]  
)  
)
```

Install Vertex AI SDK

Agent Engine is part of the Vertex AI SDK for Python. For more information, you can review the [Agent Engine quickstart documentation](#).

Install the Vertex AI SDK

```
pip install google-cloud-aiplatform[adk,agent_engines]
```

Info

Agent Engine only supported Python version >=3.9 and <=3.12.

Initialization

```
import vertexai

PROJECT_ID = "your-project-id"

LOCATION = "us-central1"

STAGING_BUCKET = "gs://your-google-cloud-storage-bucket"

vertexai.init(
    project=PROJECT_ID,
    location=LOCATION,
```

```
        staging_bucket=STAGING_BUCKET,  
    )
```

For `LOCATION`, you can check out the list of supported regions in Agent Engine.

Create your agent

You can use the sample agent below, which has two tools (to get weather or retrieve the time in a specified city):

```
import datetime  
  
from zoneinfo import ZoneInfo  
  
from google.adk.agents import Agent  
  
  
def get_weather(city: str) -> dict:  
    """Retrieves the current weather report for a specified city.  
    """
```

Args:

```
    city (str): The name of the city for which to retrieve the weather report.
```

Returns:

```
    dict: status and result or error msg.
```

```
    """
```

```
if city.lower() == "new york":
```

```
    return {
```

```
        "status": "success",
```

```
        "report": (
```

```
            "The weather in New York is sunny with a temperature of 25 degrees"
```

```
        " Celsius (77 degrees Fahrenheit)."

    ),
}

else:

    return {

        "status": "error",

        "error_message": f"Weather information for '{city}' is not
available.",

    }

def get_current_time(city: str) -> dict:
```

```
    """Returns the current time in a specified city.
```

```
Args:
```

```
    city (str): The name of the city for which to retrieve the current  
time.
```

```
Returns:
```

```
    dict: status and result or error msg.
```

```
    ....
```

```
if city.lower() == "new york":
```

```
    tz_identifier = "America/New_York"
```

```
    else:

        return {

            "status": "error",

            "error_message": (

                f"Sorry, I don't have timezone information for {city}."

            ),

        }

    tz = ZoneInfo(tz_identifier)

    now = datetime.datetime.now(tz)

    report = (
```

```
f'The current time in {city} is {now.strftime("%Y-%m-%d %H:%M:%S\n%Z%z")}'\n\n)\n\nreturn {"status": "success", "report": report}\n\nroot_agent = Agent(\n    name="weather_time_agent",\n    model="gemini-2.0-flash",\n    description=(\n        "Agent to answer questions about the time and weather in a city.",\n    ),
```

```
instruction=

    "You are a helpful agent who can answer user questions about the time
and weather in a city." ,  
  
tools=[get_weather, get_current_time],  
  
)
```

Prepare your agent for Agent Engine

Use `reasoning_engines.AdkApp()` to wrap your agent to make it deployable to Agent Engine

```
from vertexai.preview import reasoning_engines
```

```
app = reasoning_engines.AdkApp(
```

```
    agent=root_agent,  
  
    enable_tracing=True,
```

```
)
```

Try your agent locally

You can try it locally before deploying to Agent Engine.

Create session (local)

```
session = app.create_session(user_id="u_123")
```

```
session
```

Expected output for `create_session (local)`:

```
Session(id='c6a33dae-26ef-410c-9135-b434a528291f',
app_name='default-app-name', user_id='u_123', state={}, events=[],
last_update_time=1743440392.8689594)
```

List sessions (local)

```
app.list_sessions(user_id="u_123")
```

Expected output for `list_sessions (local)`:

```
ListSessionsResponse(session_ids=['c6a33dae-26ef-410c-9135-b434a528291f'])
```

Get a specific session (local)

```
session = app.get_session(user_id="u_123", session_id=session.id)
```

```
session
```

Expected output for `get_session (local)`:

```
Session(id='c6a33dae-26ef-410c-9135-b434a528291f',  
app_name='default-app-name', user_id='u_123', state={}, events=[],  
last_update_time=1743681991.95696)
```

Send queries to your agent (local)

```
for event in app.stream_query(
```

```
    user_id="u_123",
```

```
    session_id=session.id,
```

```
    message="whats the weather in new york",
```

```
):
```

```
print(event)
```

Expected output for `stream_query (local)`:

```
{'parts': [ {'function_call': {'id': 'af-a33fedb0-29e6-4d0c-9eb3-00c402969395',  
'args': {'city': 'new york'}, 'name': 'get_weather'}}], 'role': 'model'}
```

```
{'parts': [ {'function_response': {'id': 'af-a33fdb0-29e6-4d0c-9eb3-00c402969395', 'name': 'get_weather', 'response': {'status': 'success', 'report': 'The weather in New York is sunny with a temperature of 25 degrees Celsius (41 degrees Fahrenheit).'} }], 'role': 'user' }
```

```
{'parts': [ {'text': 'The weather in New York is sunny with a temperature of 25 degrees Celsius (41 degrees Fahrenheit).'} ], 'role': 'model' }
```

Deploy your agent to Agent Engine

```
from vertexai import agent_engines

remote_app = agent_engines.create(
    agent_engine=root_agent,
    requirements=[

        "google-cloud-aiplatform[adk,agent_engines]"
    ]
)
```

```
)
```

This step may take several minutes to finish. Each deployed agent has a unique identifier. You can run the following command to get the resource_name identifier for your deployed agent:

```
remote_app.resource_name
```

The response should look like the following string:

```
f"projects/{PROJECT_NUMBER}/locations/{LOCATION}/reasoningEngines/{RESOURCE_ID}  
}"
```

For additional details, you can visit the Agent Engine documentation [deploying an agent](#) and [managing deployed agents](#).

Try your agent on Agent Engine

Create session (remote)

```
remote_session = remote_app.create_session(user_id="u_456")
```

```
remote_session
```

Expected output for `create_session (remote)`:

```
{'events': [],
```

```
'user_id': 'u_456',
```

```
'state': {},  
  
'id': '7543472750996750336',  
  
'app_name': '7917477678498709504',  
  
'last_update_time': 1743683353.030133}
```

`id` is the session ID, and `app_name` is the resource ID of the deployed agent on Agent Engine.

List sessions (remote)

```
remote_app.list_sessions(user_id="u_456")
```

Get a specific session (remote)

```
remote_app.get_session(user_id="u_456", session_id=remote_session["id"])
```

Note

While using your agent locally, session ID is stored in `session.id`, when using your agent remotely on Agent Engine, session ID is stored in `remote_session["id"]`.

Send queries to your agent (remote)

```
for event in remote_app.stream_query()
```

```
    user_id="u_456",  
  
    session_id=remote_session["id"],  
  
    message="whats the weather in new york",  
):  
  
    print(event)
```

Expected output for `stream_query` (remote):

```
{"parts": [{"function_call": {"id": "af-f1906423-a531-4ecf-a1ef-723b05e85321",  
"args": {"city": "new york"}, "name": "get_weather"}}], "role": "model"}  
  
{"parts": [{"function_response": {"id":  
"af-f1906423-a531-4ecf-a1ef-723b05e85321", "name": "get_weather", "response":  
{"status": "success", "report": "The weather in New York is sunny with a  
temperature of 25 degrees Celsius (41 degrees Fahrenheit)."}}], "role":  
"user"}  
  
{"parts": [{"text": "The weather in New York is sunny with a temperature of 25  
degrees Celsius (41 degrees Fahrenheit)."}], "role": "model"}
```

Clean up

After you have finished, it is a good practice to clean up your cloud resources. You can delete the deployed Agent Engine instance to avoid any unexpected charges on your Google Cloud account.

```
remote_app.delete(force=True)
```

`force=True` will also delete any child resources that were generated from the deployed agent, such as sessions.

Deploy to Cloud Run

Cloud Run is a fully managed platform that enables you to run your code directly on top of Google's scalable infrastructure.

To deploy your agent, you can use either the `adk deploy cloud_run` command (*recommended for Python*), or with `gcloud run deploy` command through Cloud Run.

Agent sample

For each of the commands, we will reference a the Capital Agent sample defined on the [LLM agent](#) page. We will assume it's in a directory (eg: `capital_agent`).

To proceed, confirm that your agent code is configured as follows:

Python

Java

1. Agent code is in a file called `agent.py` within your agent directory.
2. Your agent variable is named `root_agent`.
3. `__init__.py` is within your agent directory and contains `from . import agent`.

Environment variables

Set your environment variables as described in the [Setup and Installation guide](#).

```
export GOOGLE_CLOUD_PROJECT=your-project-id
```

```
export GOOGLE_CLOUD_LOCATION=us-central1 # Or your preferred location
```

```
export GOOGLE_GENAI_USE_VERTEXAI=True
```

(Replace `your-project-id` with your actual GCP project ID)

Deployment commands

[Python - adk CLI](#)

[Python - gcloud CLI](#)

[Java - gcloud CLI](#)

adk CLI

The `adk deploy cloud_run` command deploys your agent code to Google Cloud Run.

Ensure you have authenticated with Google Cloud (`gcloud auth login` and `gcloud config set project <your-project-id>`).

Setup environment variables

Optional but recommended: Setting environment variables can make the deployment commands cleaner.

```
# Set your Google Cloud Project ID
```

```
export GOOGLE_CLOUD_PROJECT="your-gcp-project-id"
```

```
# Set your desired Google Cloud Location
```

```
export GOOGLE_CLOUD_LOCATION="us-central1" # Example location
```

```
# Set the path to your agent code directory
```

```
export AGENT_PATH="../capital_agent" # Assuming capital_agent is in the current directory

# Set a name for your Cloud Run service (optional)

export SERVICE_NAME="capital-agent-service"

# Set an application name (optional)

export APP_NAME="capital-agent-app"
```

Command usage

Minimal command

```
adk deploy cloud_run \
```

```
--project=$GOOGLE_CLOUD_PROJECT \
```

```
--region=$GOOGLE_CLOUD_LOCATION \
```

```
$AGENT_PATH
```

Full command with optional flags

```
adk deploy cloud_run \
```

```
--project=$GOOGLE_CLOUD_PROJECT \
```

```
--region=$GOOGLE_CLOUD_LOCATION \
```

```
--service_name=$SERVICE_NAME \
```

```
--app_name=$APP_NAME \
```

```
--with_ui \
```

```
$AGENT_PATH
```

Arguments

- `AGENT_PATH`: (Required) Positional argument specifying the path to the directory containing your agent's source code (e.g., `$AGENT_PATH` in the examples, or `capital_agent/`). This directory must contain at least an `__init__.py` and your main agent file (e.g., `agent.py`).

Options

- `--project TEXT`: (Required) Your Google Cloud project ID (e.g., `$GOOGLE_CLOUD_PROJECT`).
- `--region TEXT`: (Required) The Google Cloud location for deployment (e.g., `$GOOGLE_CLOUD_LOCATION`, `us-central1`).
- `--service_name TEXT`: (Optional) The name for the Cloud Run service (e.g., `$SERVICE_NAME`). Defaults to `adk-default-service-name`.
- `--app_name TEXT`: (Optional) The application name for the ADK API server (e.g., `$APP_NAME`). Defaults to the name of the directory specified by `AGENT_PATH` (e.g., `capital_agent` if `AGENT_PATH` is `./capital_agent`).
- `--agent_engine_id TEXT`: (Optional) If you are using a managed session service via Vertex AI Agent Engine, provide its resource ID here.
- `--port INTEGER`: (Optional) The port number the ADK API server will listen on within the container. Defaults to 8000.
- `--with_ui`: (Optional) If included, deploys the ADK dev UI alongside the agent API server. By default, only the API server is deployed.

- `--temp_folder` TEXT: (Optional) Specifies a directory for storing intermediate files generated during the deployment process. Defaults to a timestamped folder in the system's temporary directory. (*Note: This option is generally not needed unless troubleshooting issues*).
- `--help`: Show the help message and exit.

Authenticated access

During the deployment process, you might be prompted: `Allow unauthenticated invocations to [your-service-name] (y/N)?`.

- Enter `y` to allow public access to your agent's API endpoint without authentication.
- Enter `N` (or press Enter for the default) to require authentication (e.g., using an identity token as shown in the "Testing your agent" section).

Upon successful execution, the command will deploy your agent to Cloud Run and provide the URL of the deployed service.

Testing your agent

Once your agent is deployed to Cloud Run, you can interact with it via the deployed UI (if enabled) or directly with its API endpoints using tools like `curl`. You'll need the service URL provided after deployment.

[UI Testing](#)

[API Testing \(curl\)](#)

UI Testing

If you deployed your agent with the UI enabled:

- **adk CLI:** You included the `--with_ui` flag during deployment.
- **gcloud CLI:** You set `SERVE_WEB_INTERFACE = True` in your `main.py`.

You can test your agent by simply navigating to the Cloud Run service URL provided after deployment in your web browser.

```
# Example URL format
```

```
# https://your-service-name-abc123xyz.a.run.app
```

The ADK dev UI allows you to interact with your agent, manage sessions, and view execution details directly in the browser.

To verify your agent is working as intended, you can:

1. Select your agent from the dropdown menu.
2. Type a message and verify that you receive an expected response from your agent.

If you experience any unexpected behavior, check the [Cloud Run console logs](#).

Deploy to GKE

[GKE](#) is Google Cloud's managed Kubernetes service. It allows you to deploy and manage containerized applications using Kubernetes.

To deploy your agent you will need to have a Kubernetes cluster running on GKE. You can create a cluster using the Google Cloud Console or the `gcloud` command line tool.

In this example we will deploy a simple agent to GKE. The agent will be a FastAPI application that uses `Gemini 2.0 Flash` as the LLM. We can use Vertex AI or AI Studio as the LLM provider using a Environment variable.

Agent sample

For each of the commands, we will reference a `capital_agent` sample defined in on the [LLM agent](#) page. We will assume it's in a `capital_agent` directory.

To proceed, confirm that your agent code is configured as follows:

1. Agent code is in a file called `agent.py` within your agent directory.
2. Your agent variable is named `root_agent`.
3. `__init__.py` is within your agent directory and contains `from . import agent`.

Environment variables

Set your environment variables as described in the [Setup and Installation](#) guide. You also need to install the `kubectl` command line tool. You can find instructions to do so in the [Google Kubernetes Engine Documentation](#).

```
export GOOGLE_CLOUD_PROJECT=your-project-id # Your GCP project ID
```

```
export GOOGLE_CLOUD_LOCATION=us-central1 # Or your preferred location
```

```
export GOOGLE_GENAI_USE_VERTEXAI=true # Set to true if using Vertex AI
```

```
export GOOGLE_CLOUD_PROJECT_NUMBER=$(gcloud projects describe --format json $GOOGLE_CLOUD_PROJECT | jq -r ".projectNumber")
```

If you don't have `jq` installed, you can use the following command to get the project number:

```
gcloud projects describe $GOOGLE_CLOUD_PROJECT
```

And copy the project number from the output.

```
export GOOGLE_CLOUD_PROJECT_NUMBER=YOUR_PROJECT_NUMBER
```

Deployment commands

gcloud CLI

You can deploy your agent to GKE using the `gcloud` and `kubectl` cli and Kubernetes manifest files.

Ensure you have authenticated with Google Cloud (`gcloud auth login` and `gcloud config set project <your-project-id>`).

Enable APIs

Enable the necessary APIs for your project. You can do this using the `gcloud` command line tool.

```
gcloud services enable \
```

```
container.googleapis.com \
artifactregistry.googleapis.com \
cloudbuild.googleapis.com \
aiplatform.googleapis.com
```

Create a GKE cluster

You can create a GKE cluster using the `gcloud` command line tool. This example creates an Autopilot cluster named `adk-cluster` in the `us-central1` region.

If creating a GKE Standard cluster, make sure [Workload Identity](#) is enabled. Workload Identity is enabled by default in an AutoPilot cluster.

```
gcloud container clusters create-auto adk-cluster \
--location=$GOOGLE_CLOUD_LOCATION \
--project=$GOOGLE_CLOUD_PROJECT
```

After creating the cluster, you need to connect to it using `kubectl`. This command configures `kubectl` to use the credentials for your new cluster.

```
gcloud container clusters get-credentials adk-cluster \
```

```
--location=$GOOGLE_CLOUD_LOCATION \
```

```
--project=$GOOGLE_CLOUD_PROJECT
```

Project Structure

Organize your project files as follows:

```
your-project-directory/
```

```
    |__ capital_agent/
```

```
        |__ __init__.py
```

```
    |__ agent.py          # Your agent code (see "Agent sample" tab)
```

```
    |__ main.py           # FastAPI application entry point
```

```
    |__ requirements.txt  # Python dependencies
```

```
    |__ Dockerfile         # Container build instructions
```

Create the following files (`main.py`, `requirements.txt`, `Dockerfile`) in the root of your-project-directory/.

Code files

1. This file sets up the FastAPI application using `get_fast_api_app()` from ADK:
2. `main.py`

```
import os
```

3.

4.

```
import uvicorn
```

5.

```
from fastapi import FastAPI
```

6.

```
from google.adk.cli.fast_api import get_fast_api_app
```

7.

8.

```
# Get the directory where main.py is located
```

9.

```
AGENT_DIR = os.path.dirname(os.path.abspath(__file__))
```

10.

```
# Example session DB URL (e.g., SQLite)
```

11.

```
SESSION_DB_URL = "sqlite:///sessions.db"
```

12.

```
# Example allowed origins for CORS
```

13.

```
ALLOWED_ORIGINS = ["http://localhost", "http://localhost:8080", "*"]
```

14.

```
# Set web=True if you intend to serve a web interface, False otherwise
```

15.

```
SERVE_WEB_INTERFACE = True
```

16.

17.

```
# Call the function to get the FastAPI app instance
```

18.

```
# Ensure the agent directory name ('capital_agent') matches your agent folder
```

19.

```
app: FastAPI = get_fast_api_app(
```

20.

```
    agents_dir=AGENT_DIR,
```

21.

```
    session_db_url=SESSION_DB_URL,
```

22.

```
    allow_origins=ALLOWED_ORIGINS,
```

23.

```
    web=SERVE_WEB_INTERFACE,
```

24.

```
)
```

25.

26.

```
# You can add more FastAPI routes or configurations below if needed
```

27.

```
# Example:
```

28.

```
# @app.get("/hello")
```

29.

```
# async def read_root():
```

30.

```
#     return {"Hello": "World"}
```

31.

32.

```
if __name__ == "__main__":
```

33.

```
    # Use the PORT environment variable provided by Cloud Run, defaulting to  
    8080
```

34.

```
uvicorn.run(app, host="0.0.0.0", port=int(os.environ.get("PORT", 8080)))
```

35.

36. *Note: We specify agent_dir to the directory main.py is in and use os.environ.get("PORT", 8080) for Cloud Run compatibility.*

37. List the necessary Python packages:

38. **requirements.txt**

```
google_adk
```

39.

```
# Add any other dependencies your agent needs
```

40.

41. Define the container image:

42. **Dockerfile**

```
FROM python:3.13-slim
```

43.

```
WORKDIR /app
```

44.

45.

```
COPY requirements.txt .
```

46.

```
RUN pip install --no-cache-dir -r requirements.txt
```

47.

48.

```
RUN adduser --disabled-password --gecos "" myuser && \
```

49.

```
chown -R myuser:myuser /app
```

50.

51.

```
COPY . .
```

52.

53.

```
USER myuser
```

54.

55.

```
ENV PATH="/home/myuser/.local/bin:$PATH"
```

56.

57.

```
CMD [ "sh", "-c", "uvicorn main:app --host 0.0.0.0 --port $PORT" ]
```

58.

Build the container image

You need to create a Google Artifact Registry repository to store your container images. You can do this using the `gcloud` command line tool.

```
gcloud artifacts repositories create adk-repo \  
    --repository-format=docker \  
    --location=$GOOGLE_CLOUD_LOCATION \  
    --description="ADK repository"
```

Build the container image using the `gcloud` command line tool. This example builds the image and tags it as `adk-repo/adk-agent:latest`.

```
gcloud builds submit \  
    --tag
```

```
$GOOGLE_CLOUD_LOCATION-docker.pkg.dev/$GOOGLE_CLOUD_PROJECT/adk-repo/adk-agent  
:latest \  
    --tag
```

```
--project=$GOOGLE_CLOUD_PROJECT \
```

```
.
```

Verify the image is built and pushed to the Artifact Registry:

```
gcloud artifacts docker images list \
```

```
$GOOGLE_CLOUD_LOCATION-docker.pkg.dev/$GOOGLE_CLOUD_PROJECT/adk-repo \
```

```
--project=$GOOGLE_CLOUD_PROJECT
```

Configure Kubernetes Service Account for Vertex AI

If your agent uses Vertex AI, you need to create a Kubernetes service account with the necessary permissions. This example creates a service account named `adk-agent-sa` and binds it to the `Vertex AI User` role.

If you are using AI Studio and accessing the model with an API key you can skip this step.

```
kubectl create serviceaccount adk-agent-sa
```

```
gcloud projects add-iam-policy-binding projects/${GOOGLE_CLOUD_PROJECT} \
```

```
--role=roles/aiplatform.user \
```

```
--member=principal://iam.googleapis.com/projects/${GOOGLE_CLOUD_PROJECT_NUMBER}  
}/locations/global/workloadIdentityPools/${GOOGLE_CLOUD_PROJECT}.svc.id.goog/  
subject/ns/default/sa/adk-agent-sa \
```

```
--condition=None
```

Create the Kubernetes manifest files

Create a Kubernetes deployment manifest file named `deployment.yaml` in your project directory. This file defines how to deploy your application on GKE.

deployment.yaml

```
cat << EOF > deployment.yaml
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: adk-agent
```

```
spec:
```

```
  replicas: 1
```

```
  selector:
```

```
    matchLabels:  
  
        app: adk-agent  
  
    template:  
  
        metadata:  
  
            labels:  
  
                app: adk-agent  
  
        spec:  
  
            serviceAccount: adk-agent-sa  
  
            containers:  
  
                - name: adk-agent  
  
                    imagePullPolicy: Always  
  
                    image:  
$GOOGLE_CLOUD_LOCATION-docker.pkg.dev/$GOOGLE_CLOUD_PROJECT/adk-repo/adk-agent  
:latest  
  
            resources:
```

```
    limits:
```

```
        memory: "128Mi"
```

```
        cpu: "500m"
```

```
        ephemeral-storage: "128Mi"
```

```
    requests:
```

```
        memory: "128Mi"
```

```
        cpu: "500m"
```

```
        ephemeral-storage: "128Mi"
```

```
    ports:
```

```
        - containerPort: 8080
```

```
    env:
```

```
        - name: PORT
```

```
            value: "8080"
```

```
        - name: GOOGLE_CLOUD_PROJECT
```

```
        value: GOOGLE_CLOUD_PROJECT

      - name: GOOGLE_CLOUD_LOCATION
        value: GOOGLE_CLOUD_LOCATION

      - name: GOOGLE_GENAI_USE_VERTEXAI
        value: GOOGLE_GENAI_USE_VERTEXAI

# If using AI Studio, set GOOGLE_GENAI_USE_VERTEXAI to false and set
the following:
```

```
# - name: GOOGLE_API_KEY
#   value: GOOGLE_API_KEY

# Add any other necessary environment variables your agent might need
```

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: adk-agent
```

```
spec:
```

```
  type: LoadBalancer
```

```
  ports:
```

```
    - port: 80
```

```
      targetPort: 8080
```

```
  selector:
```

```
    app: adk-agent
```

```
EOF
```

Deploy the Application

Deploy the application using the `kubectl` command line tool. This command applies the deployment and service manifest files to your GKE cluster.

```
kubectl apply -f deployment.yaml
```

After a few moments, you can check the status of your deployment using:

```
kubectl get pods -l=app=adk-agent
```

This command lists the pods associated with your deployment. You should see a pod with a status of `Running`.

Once the pod is running, you can check the status of the service using:

```
kubectl get service adk-agent
```

If the output shows a `External IP`, it means your service is accessible from the internet. It may take a few minutes for the external IP to be assigned.

You can get the external IP address of your service using:

```
kubectl get svc adk-agent -o=jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

Testing your agent

Once your agent is deployed to GKE, you can interact with it via the deployed UI (if enabled) or directly with its API endpoints using tools like `curl`. You'll need the service URL provided after deployment.

[UI Testing](#)

[API Testing \(curl\)](#)

UI Testing

If you deployed your agent with the UI enabled:

You can test your agent by simply navigating to the kubernetes service URL in your web browser.

The ADK dev UI allows you to interact with your agent, manage sessions, and view execution details directly in the browser.

To verify your agent is working as intended, you can:

1. Select your agent from the dropdown menu.
2. Type a message and verify that you receive an expected response from your agent.

If you experience any unexpected behavior, check the pod logs for your agent using:

```
kubectl logs -l app=adk-agent
```

Troubleshooting

These are some common issues you might encounter when deploying your agent to GKE:

403 Permission Denied for Gemini 2.0 Flash

This usually means that the Kubernetes service account does not have the necessary permission to access the Vertex AI API. Ensure that you have created the service account and bound it to the `Vertex AI User` role as described in the [Configure Kubernetes Service Account for Vertex AI](#) section. If you are using AI Studio, ensure that you have set the `GOOGLE_API_KEY` environment variable in the deployment manifest and it is valid.

Attempt to write a readonly database

You might see there is no session id created in the UI and the agent does not respond to any messages. This is usually caused by the SQLite database being read-only. This can happen if you run the agent locally and then create the container image which copies the SQLite database into the container. The database is then read-only in the container.

```
sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) attempt to write a  
readonly database
```

```
[SQL: UPDATE app_states SET state=?, update_time=CURRENT_TIMESTAMP WHERE  
app_states.app_name = ?]
```

To fix this issue, you can either:

Delete the SQLite database file from your local machine before building the container image. This will create a new SQLite database when the container is started.

```
rm -f sessions.db
```

or (recommended) you can add a `.dockerignore` file to your project directory to exclude the SQLite database from being copied into the container image.

```
.dockerignore  
sessions.db
```

Build the container image abd deploy the application again.

Cleanup

To delete the GKE cluster and all associated resources, run:

```
gcloud container clusters delete adk-cluster \  
--location=$GOOGLE_CLOUD_LOCATION \  
--force-delete-pods --force-delete-persistent-volumes
```

```
--project=$GOOGLE_CLOUD_PROJECT
```

To delete the Artifact Registry repository, run:

```
gcloud artifacts repositories delete adk-repo \
```

```
--location=$GOOGLE_CLOUD_LOCATION \
```

```
--project=$GOOGLE_CLOUD_PROJECT
```

You can also delete the project if you no longer need it. This will delete all resources associated with the project, including the GKE cluster, Artifact Registry repository, and any other resources you created.

```
gcloud projects delete $GOOGLE_CLOUD_PROJECT
```

Introduction to Conversational Context: Session, State, and Memory

Why Context Matters

Meaningful, multi-turn conversations require agents to understand context. Just like humans, they need to recall the conversation history: what's been said and done to maintain continuity and avoid repetition. The Agent Development Kit (ADK) provides structured ways to manage this context through [Session, State, and Memory](#).

Core Concepts

Think of different instances of your conversations with the agent as distinct **conversation threads**, potentially drawing upon **long-term knowledge**.

1. **Session**: The Current Conversation Thread
 - Represents a *single, ongoing interaction* between a user and your agent system.
 - Contains the chronological sequence of messages and actions taken by the agent (referred to **Events**) during *that specific interaction*.
 - A **Session** can also hold temporary data (**State**) relevant only *during this conversation*.
2. **State** (`session.state`): Data Within the Current Conversation
 - Data stored within a specific **Session**.
 - Used to manage information relevant *only* to the *current, active* conversation thread (e.g., items in a shopping cart *during this chat*, user preferences mentioned *in this session*).
3. **Memory**: Searchable, Cross-Session Information
 - Represents a store of information that might span *multiple past sessions* or include external data sources.
 - It acts as a knowledge base the agent can **search** to recall information or context beyond the immediate conversation.

Managing Context: Services

ADK provides services to manage these concepts:

1. **SessionService**: Manages the different conversation threads (**Session** objects)
 - Handles the lifecycle: creating, retrieving, updating (appending **Events**, modifying **State**), and deleting individual **Sessions**.
2. **MemoryService**: Manages the Long-Term Knowledge Store (**Memory**)
 - Handles ingesting information (often from completed **Sessions**) into the long-term store.

- Provides methods to search this stored knowledge based on queries.

Implementations: ADK offers different implementations for both `SessionService` and `MemoryService`, allowing you to choose the storage backend that best fits your application's needs. Notably, **in-memory implementations** are provided for both services; these are designed specifically for **local testing and fast development**. It's important to remember that **all data stored using these in-memory options (sessions, state, or long-term knowledge) is lost when your application restarts**. For persistence and scalability beyond local testing, ADK also offers cloud-based and database service options.

In Summary:

- **Session & State:** Focus on the **current interaction** – the history and data of the *single, active conversation*. Managed primarily by a `SessionService`.
- **Memory:** Focuses on the **past and external information** – a *searchable archive* potentially spanning across conversations. Managed by a `MemoryService`.

What's Next?

In the following sections, we'll dive deeper into each of these components:

- **Session:** Understanding its structure and `Events`.
- **State:** How to effectively read, write, and manage session-specific data.
- **SessionService:** Choosing the right storage backend for your sessions.
- **MemoryService:** Exploring options for storing and retrieving broader context.

Understanding these concepts is fundamental to building agents that can engage in complex, stateful, and context-aware conversations.

Session: Tracking Individual Conversations

Following our Introduction, let's dive into the `Session`. Think back to the idea of a "conversation thread." Just like you wouldn't start every text message from scratch,

agents need context regarding the ongoing interaction. `Session` is the ADK object designed specifically to track and manage these individual conversation threads.

The Session Object

When a user starts interacting with your agent, the `SessionService` creates a `Session` object (`google.adk.sessions.Session`). This object acts as the container holding everything related to that *one specific chat thread*. Here are its key properties:

- **Identification (`id`, `appName`, `userId`)**: Unique labels for the conversation.
 - `id`: A unique identifier for *this specific* conversation thread, essential for retrieving it later. A `SessionService` object can handle multiple `Session`(s). This field identifies which particular session object are we referring to. For example, "test_id_modification".
 - `app_name`: Identifies which agent application this conversation belongs to. For example, "id_modifier_workflow".
 - `userId`: Links the conversation to a particular user.
- **History (`events`)**: A chronological sequence of all interactions (`Event` objects – user messages, agent responses, tool actions) that have occurred within this specific thread.
- **Session State (`state`)**: A place to store temporary data relevant *only* to this specific, ongoing conversation. This acts as a scratchpad for the agent during the interaction. We will cover how to use and manage `state` in detail in the next section.
- **Activity Tracking (`lastUpdateTime`)**: A timestamp indicating the last time an event occurred in this conversation thread.

Example: Examining Session Properties

Python

Java

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

```
# Create a simple session to examine its properties

temp_service = InMemorySessionService()

example_session = await temp_service.create_session(
    app_name="my_app",
    user_id="example_user",
    state={"initial_key": "initial_value"} # State can be initialized
)

print(f"--- Examining Session Properties ---")

print(f"ID (`id`): {example_session.id}")
print(f"Application Name (`app_name`): {example_session.app_name}")
```

```
print(f"User ID (`user_id`):           {example_session.user_id}")  
  
print(f"State (`state`):           {example_session.state}") # Note: Only  
shows initial state here  
  
print(f"Events (`events`):           {example_session.events}") # Initially  
empty  
  
print(f"Last Update (`last_update_time`):  
{example_session.last_update_time:.2f}")  
  
print("-----")  
  
# Clean up (optional for this example)  
  
temp_service = await  
temp_service.delete_session(app_name=example_session.app_name,  
                           user_id=example_session.user_id,  
                           session_id=example_session.id)  
  
print("The final status of temp_service - ", temp_service)
```

(Note: The state shown above is only the initial state. State updates happen via events, as discussed in the State section.)

Managing Sessions with a SessionService

As seen above, you don't typically create or manage `Session` objects directly. Instead, you use a `SessionService`. This service acts as the central manager responsible for the entire lifecycle of your conversation sessions.

Its core responsibilities include:

- **Starting New Conversations:** Creating fresh `Session` objects when a user begins an interaction.
- **Resuming Existing Conversations:** Retrieving a specific `Session` (using its ID) so the agent can continue where it left off.
- **Saving Progress:** Appending new interactions (`Event` objects) to a session's history. This is also the mechanism through which session `state` gets updated (more in the `State` section).
- **Listing Conversations:** Finding the active session threads for a particular user and application.
- **Cleaning Up:** Deleting `Session` objects and their associated data when conversations are finished or no longer needed.

SessionService Implementations

ADK provides different `SessionService` implementations, allowing you to choose the storage backend that best suits your needs:

1. `InMemorySessionService`
 - **How it works:** Stores all session data directly in the application's memory.

- **Persistence:** None. **All conversation data is lost if the application restarts.**
- **Requires:** Nothing extra.
- **Best for:** Quick development, local testing, examples, and scenarios where long-term persistence isn't required.

2. [Python](#)

3. [Java](#)

```
from google.adk.sessions import InMemorySessionService
```

4.

```
session_service = InMemorySessionService()
```

5.

6. [VertexAiSessionService](#)

- **How it works:** Uses Google Cloud's Vertex AI infrastructure via API calls for session management.
- **Persistence:** Yes. Data is managed reliably and scalably via [Vertex AI Agent Engine](#).
- **Requires:**
 - A Google Cloud project (`pip install vertexai`)
 - A Google Cloud storage bucket that can be configured by this [step](#).
 - A Reasoning Engine resource name/ID that can setup following this [tutorial](#).
- **Best for:** Scalable production applications deployed on Google Cloud, especially when integrating with other Vertex AI features.

7. [Python](#)

8. [Java](#)

```
# Requires: pip install google-adk[vertexai]
```

9.

```
# Plus GCP setup and authentication
```

10.

```
from google.adk.sessions import VertexAiSessionService
```

11.

12.

```
PROJECT_ID = "your-gcp-project-id"
```

13.

```
LOCATION = "us-central1"
```

14.

```
# The app_name used with this service should be the Reasoning Engine ID or  
name
```

15.

```
REASONING_ENGINE_APP_NAME =  
"projects/your-gcp-project-id/locations/us-central1/reasoningEngines/your-engine-id"
```

16.

17.

```
session_service = VertexAiSessionService(project=PROJECT_ID,  
location=LOCATION)
```

18.

```
# Use REASONING_ENGINE_APP_NAME when calling service methods, e.g.:
```

19.

```
# session_service = await  
session_service.create_session(app_name=REASONING_ENGINE_APP_NAME, ...)
```

20.

21. DatabaseSessionService



- **How it works:** Connects to a relational database (e.g., PostgreSQL, MySQL, SQLite) to store session data persistently in tables.
- **Persistence:** Yes. Data survives application restarts.
- **Requires:** A configured database.
- **Best for:** Applications needing reliable, persistent storage that you manage yourself.

```
from google.adk.sessions import DatabaseSessionService
```

22.

```
# Example using a local SQLite file:
```

23.

```
db_url = "sqlite:///./my_agent_data.db"
```

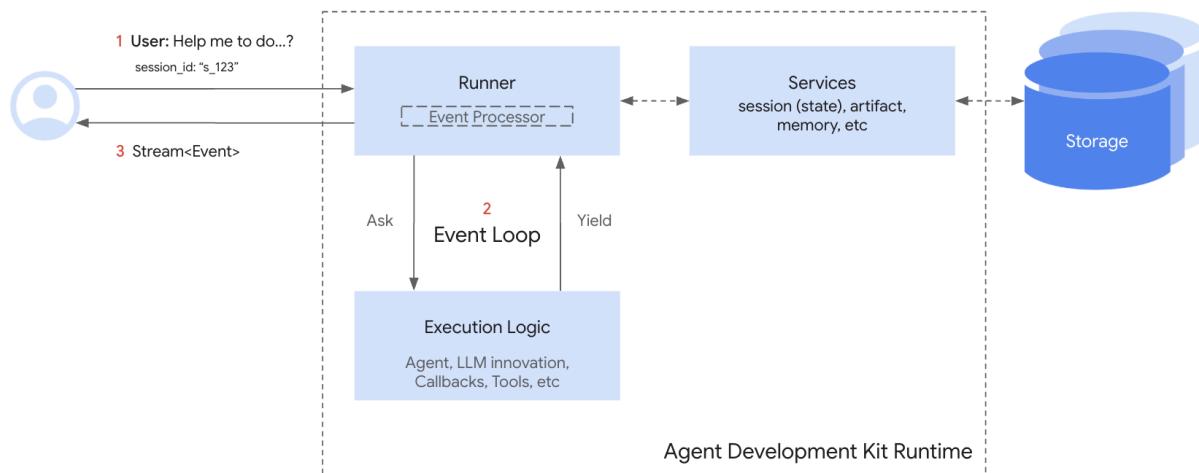
24.

```
session_service = DatabaseSessionService(db_url=db_url)
```

25.

Choosing the right `SessionService` is key to defining how your agent's conversation history and temporary data are stored and persist.

The Session Lifecycle



Here's a simplified flow of how `Session` and `SessionService` work together during a conversation turn:

1. **Start or Resume:** Your application's `Runner` uses the `SessionService` to either `create_session` (for a new chat) or `get_session` (to retrieve an existing one).
2. **Context Provided:** The `Runner` gets the appropriate `Session` object from the appropriate service method, providing the agent with access to the corresponding `Session`'s `state` and `events`.
3. **Agent Processing:** The user prompts the agent with a query. The agent analyzes the query and potentially the `session state` and `events` history to determine the response.
4. **Response & State Update:** The agent generates a response (and potentially flags data to be updated in the `state`). The `Runner` packages this as an `Event`.
5. **Save Interaction:** The `Runner` calls `sessionService.append_event(session, event)` with the `session` and the new `event` as the arguments. The service adds the `Event` to the history and updates the `session's state` in storage based on information within the event. The `session's last_update_time` also get updated.
6. **Ready for Next:** The agent's response goes to the user. The updated `Session` is now stored by the `SessionService`, ready for the next turn (which restarts the cycle at step 1, usually with the continuation of the conversation in the current session).
7. **End Conversation:** When the conversation is over, your application calls `sessionService.delete_session(...)` to clean up the stored session data if it is no longer required.

This cycle highlights how the `SessionService` ensures conversational continuity by managing the history and state associated with each `Session` object.

State: The Session's Scratchpad

Within each `Session` (our conversation thread), the `state` attribute acts like the agent's dedicated scratchpad for that specific interaction. While `session.events` holds the full history, `session.state` is where the agent stores and updates dynamic details needed *during* the conversation.

What is `session.state`?

Conceptually, `session.state` is a collection (dictionary or Map) holding key-value pairs. It's designed for information the agent needs to recall or track to make the current conversation effective:

- **Personalize Interaction:** Remember user preferences mentioned earlier (e.g., `'user_preference_theme': 'dark'`).
- **Track Task Progress:** Keep tabs on steps in a multi-turn process (e.g., `'booking_step': 'confirm_payment'`).
- **Accumulate Information:** Build lists or summaries (e.g., `'shopping_cart_items': ['book', 'pen']`).
- **Make Informed Decisions:** Store flags or values influencing the next response (e.g., `'user_is_authenticated': True`).

Key Characteristics of State

1. Structure: Serializable Key-Value Pairs

- Data is stored as `key: value`.
- **Keys:** Always strings (`str`). Use clear names (e.g., `'departure_city'`, `'user:language_preference'`).
- **Values:** Must be **Serializable**. This means they can be easily saved and loaded by the `SessionService`. Stick to basic types in the specific languages (Python/ Java) like strings, numbers, booleans, and simple lists or dictionaries containing *only* these basic types. (See API documentation for precise details).
- **⚠ Avoid Complex Objects: Do not store non-serializable objects** (custom class instances, functions, connections, etc.) directly in the state. Store simple identifiers if needed, and retrieve the complex object elsewhere.

2. Mutability: It Changes

- The contents of the `state` are expected to change as the conversation evolves.

3. Persistence: Depends on `SessionService`

- Whether state survives application restarts depends on your chosen service:
- `InMemorySessionService`: **Not Persistent**. State is lost on restart.

- DatabaseSessionService / VertexAiSessionService: **Persistent**. State is saved reliably.

Note

The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., `session.state['current_intent'] = 'book_flight'` in Python, `session.state().put("current_intent", "book_flight")` in Java). Refer to the language-specific API documentation for details.

Organizing State with Prefixes: Scope Matters

Prefixes on state keys define their scope and persistence behavior, especially with persistent services:

- **No Prefix (Session State):**
 - **Scope:** Specific to the *current session* (`id`).
 - **Persistence:** Only persists if the `SessionService` is persistent (Database, VertexAI).
 - **Use Cases:** Tracking progress within the current task (e.g., `'current_booking_step'`), temporary flags for this interaction (e.g., `'needs_clarification'`).
 - **Example:** `session.state['current_intent'] = 'book_flight'`
- **user: Prefix (User State):**
 - **Scope:** Tied to the `user_id`, shared across *all* sessions for that user (within the same `app_name`).
 - **Persistence:** Persistent with `Database` or `VertexAI`. (Stored by `InMemory` but lost on restart).
 - **Use Cases:** User preferences (e.g., `'user:theme'`), profile details (e.g., `'user:name'`).
 - **Example:** `session.state['user:preferred_language'] = 'fr'`
- **app: Prefix (App State):**
 - **Scope:** Tied to the `app_name`, shared across *all* users and sessions for that application.
 - **Persistence:** Persistent with `Database` or `VertexAI`. (Stored by `InMemory` but lost on restart).
 - **Use Cases:** Global settings (e.g., `'app:api_endpoint'`), shared templates.

- **Example:** `session.state['app:global_discount_code'] = 'SAVE10'`
- **temp:** **Prefix (Temporary Session State):**
 - **Scope:** Specific to the *current* session processing turn.
 - **Persistence: Never Persistent.** Guaranteed to be discarded, even with persistent services.
 - **Use Cases:** Intermediate results needed only immediately, data you explicitly don't want stored.
 - **Example:** `session.state['temp:raw_api_response'] = {...}`

How the Agent Sees It: Your agent code interacts with the *combined* state through the single `session.state` collection (dict/ Map). The `SessionService` handles fetching/merging state from the correct underlying storage based on prefixes.

How State is Updated: Recommended Methods

State should **always** be updated as part of adding an `Event` to the session history using `session_service.append_event()`. This ensures changes are tracked, persistence works correctly, and updates are thread-safe.

1. The Easy Way: `output_key` (for Agent Text Responses)

This is the simplest method for saving an agent's final text response directly into the state. When defining your `LlmAgent`, specify the `output_key`:

Python

Java

```
from google.adk.agents import LlmAgent
```

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

```
from google.adk.runners import Runner
```

```
from google.genai.types import Content, Part
```

```
# Define agent with output_key

greeting_agent = LlmAgent(
    name="Greeter",
    model="gemini-2.0-flash", # Use a valid model
    instruction="Generate a short, friendly greeting.",
    output_key="last_greeting" # Save response to state['last_greeting']
)

# --- Setup Runner and Session ---

app_name, user_id, session_id = "state_app", "user1", "session1"
```

```
session_service = InMemorySessionService()

runner = Runner(
    agent=greeting_agent,
    app_name=app_name,
    session_service=session_service
)

session = await session_service.create_session(app_name=app_name,
                                               user_id=user_id,
                                               session_id=session_id)

print(f"Initial state: {session.state}")

# --- Run the Agent ---
```

```
# Runner handles calling append_event, which uses the output_key

# to automatically create the state_delta.

user_message = Content(parts=[Part(text="Hello")])

for event in runner.run(user_id=user_id,
                        session_id=session_id,
                        new_message=user_message):

    if event.is_final_response():

        print(f"Agent responded.") # Response text is also in event.content

# --- Check Updated State ---

updated_session = await session_service.get_session(app_name=APP_NAME,
                                                      user_id=USER_ID, session_id=session_id)
```

```
print(f"State after agent run: {updated_session.state}")

# Expected output might include: {'last_greeting': 'Hello there! How can I
help you today?'}
```

Behind the scenes, the `Runner` uses the `output_key` to create the necessary `EventActions` with a `state_delta` and calls `append_event`.

2. The Standard Way: `EventActions.state_delta` (for Complex Updates)

For more complex scenarios (updating multiple keys, non-string values, specific scopes like `user:` or `app:`, or updates not tied directly to the agent's final text), you manually construct the `state_delta` within `EventActions`.

Python

Java

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

from google.adk.events import Event, EventActions

from google.genai.types import Part, Content

import time

# --- Setup ---
```

```
session_service = InMemorySessionService()

app_name, user_id, session_id = "state_app_manual", "user2", "session2"

session = await session_service.create_session(
    app_name=app_name,
    user_id=user_id,
    session_id=session_id,
    state={"user:login_count": 0, "task_status": "idle"}
)

print(f"Initial state: {session.state}")

# --- Define State Changes ---

current_time = time.time()
```

```
state_changes = {  
  
    "task_status": "active",           # Update session state  
  
    "user:login_count": session.state.get("user:login_count", 0) + 1, # Update  
    user state  
  
    "user:last_login_ts": current_time,   # Add user state  
  
    "temp:validation_needed": True       # Add temporary state (will be  
    discarded)  
  
}  
  
# --- Create Event with Actions ---  
  
actions_with_update = EventActions(state_delta=state_changes)  
  
# This event might represent an internal system action, not just an agent  
response  
  
system_event = Event(
```

```
    invocation_id="inv_login_update",  
  
    author="system", # Or 'agent', 'tool' etc.  
  
    actions=actions_with_update,  
  
    timestamp=current_time  
  
    # content might be None or represent the action taken  
)  
  
# --- Append the Event (This updates the state) ---  
  
await session_service.append_event(session, system_event)  
  
print(`append_event` called with explicit state delta.)
```

```

# --- Check Updated State ---

updated_session = await session_service.get_session(app_name=app_name,
                                                    user_id=user_id,
                                                    session_id=session_id)

print(f"State after event: {updated_session.state}")

# Expected: {'user:login_count': 1, 'task_status': 'active',
# 'user:last_login_ts': <timestamp>}

# Note: 'temp:validation_needed' is NOT present.

```

3. Via `CallbackContext` or `ToolContext` (Recommended for Callbacks and Tools)

Modifying state within agent callbacks (e.g., `on_before_agent_call`, `on_after_agent_call`) or tool functions is best done using the `state` attribute of the `CallbackContext` or `ToolContext` provided to your function.

- `callback_context.state['my_key'] = my_value`
- `tool_context.state['my_key'] = my_value`

These context objects are specifically designed to manage state changes within their respective execution scopes. When you modify `context.state`, the ADK framework ensures that these changes are automatically captured and correctly routed into the `EventActions.state_delta` for the event being generated by the callback or tool. This delta is then processed by the `SessionService` when the event is appended, ensuring proper persistence and tracking.

This method abstracts away the manual creation of `EventActions` and `state_delta` for most common state update scenarios within callbacks and tools, making your code cleaner and less error-prone.

For more comprehensive details on context objects, refer to the [Context documentation](#).

Python

Java

```
# In an agent callback or tool function

from google.adk.agents import CallbackContext # or ToolContext

def my_callback_or_tool_function(context: CallbackContext, # Or ToolContext

                                # ... other parameters ...):

    # Update existing state

    count = context.state.get("user_action_count", 0)

    context.state["user_action_count"] = count + 1
```

```
# Add new state

context.state["temp:last_operation_status"] = "success"

# State changes are automatically part of the event's state_delta

# ... rest of callback/tool logic ...
```

What `append_event` Does:

- Adds the `Event` to `session.events`.
- Reads the `state_delta` from the event's `actions`.
- Applies these changes to the state managed by the `SessionService`, correctly handling prefixes and persistence based on the service type.
- Updates the session's `last_update_time`.
- Ensures thread-safety for concurrent updates.

⚠ A Warning About Direct State Modification

Avoid directly modifying the `session.state` collection (dictionary/Map) on a `Session` object that was obtained directly from the `SessionService` (e.g., via `session_service.get_session()` or `session_service.create_session()`) *outside* of the managed lifecycle of an agent invocation (i.e., not through a `CallbackContext` or `ToolContext`). For example, code like `retrieved_session = await session_service.get_session(...); retrieved_session.state['key'] = value` is problematic.

State modifications *within* callbacks or tools using `CallbackContext.state` or `ToolContext.state` are the correct way to ensure changes are tracked, as these context objects handle the necessary integration with the event system.

Why direct modification (outside of contexts) is strongly discouraged:

1. **Bypasses Event History:** The change isn't recorded as an `Event`, losing auditability.
2. **Breaks Persistence:** Changes made this way **will likely NOT be saved** by `DatabaseSessionService` or `VertexAiSessionService`. They rely on `append_event` to trigger saving.
3. **Not Thread-Safe:** Can lead to race conditions and lost updates.
4. **Ignores Timestamps/Logic:** Doesn't update `last_update_time` or trigger related event logic.

Recommendation: Stick to updating state via `output_key`, `EventActions.state_delta` (when manually creating events), or by modifying the `state` property of `CallbackContext` or `ToolContext` objects when within their respective scopes. These methods ensure reliable, trackable, and persistent state management. Use direct access to `session.state` (from a `SessionService`-retrieved session) only for *reading* state.

Best Practices for State Design Recap

- **Minimalism:** Store only essential, dynamic data.
- **Serialization:** Use basic, serializable types.
- **Descriptive Keys & Prefixes:** Use clear names and appropriate prefixes (`user:`, `app:`, `temp:`, or `none`).
- **Shallow Structures:** Avoid deep nesting where possible.
- **Standard Update Flow:** Rely on `append_event`.

Memory: Long-Term Knowledge with

MemoryService



We've seen how `Session` tracks the history (events) and temporary data (state) for a *single, ongoing conversation*. But what if an agent needs to recall information from *past* conversations or access external knowledge bases? This is where the concept of **Long-Term Knowledge** and the `MemoryService` come into play.

Think of it this way:

- **Session / State:** Like your short-term memory during one specific chat.
- **Long-Term Knowledge (MemoryService):** Like a searchable archive or knowledge library the agent can consult, potentially containing information from many past chats or other sources.

The MemoryService Role

The `BaseMemoryService` defines the interface for managing this searchable, long-term knowledge store. Its primary responsibilities are:

1. **Ingesting Information (`add_session_to_memory`)**: Taking the contents of a (usually completed) `Session` and adding relevant information to the long-term knowledge store.
2. **Searching Information (`search_memory`)**: Allowing an agent (typically via a `Tool`) to query the knowledge store and retrieve relevant snippets or context based on a search query.

MemoryService Implementations

ADK provides different ways to implement this long-term knowledge store:

1. `InMemoryMemoryService`

- **How it works:** Stores session information in the application's memory and performs basic keyword matching for searches.
- **Persistence:** None. **All stored knowledge is lost if the application restarts.**
- **Requires:** Nothing extra.
- **Best for:** Prototyping, simple testing, scenarios where only basic keyword recall is needed and persistence isn't required.

```
from google.adk.memory import InMemoryMemoryService
```

2.

```
memory_service = InMemoryMemoryService()
```

3.

4. `VertexAiRagMemoryService`

- **How it works:** Leverages Google Cloud's Vertex AI RAG (Retrieval-Augmented Generation) service. It ingests session data into a specified RAG Corpus and uses powerful semantic search capabilities for retrieval.
- **Persistence:** Yes. The knowledge is stored persistently within the configured Vertex AI RAG Corpus.

- **Requires:** A Google Cloud project, appropriate permissions, necessary SDKs (`pip install google-adk[vertexai]`), and a pre-configured Vertex AI RAG Corpus resource name/ID.
- **Best for:** Production applications needing scalable, persistent, and semantically relevant knowledge retrieval, especially when deployed on Google Cloud.

```
# Requires: pip install google-adk[vertexai]
```

5.

```
# Plus GCP setup, RAG Corpus, and authentication
```

6.

```
from google.adk.memory import VertexAiRagMemoryService
```

7.

8.

```
# The RAG Corpus name or ID
```

9.

```
RAG_CORPUS_RESOURCE_NAME =  
"projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id"
```

10.

```
# Optional configuration for retrieval
```

11.

```
SIMILARITY_TOP_K = 5
```

12.

```
VECTOR_DISTANCE_THRESHOLD = 0.7
```

13.

14.

```
memory_service = VertexAiRagMemoryService(
```

15.

```
    rag_corpus=RAG_CORPUS_RESOURCE_NAME,
```

16.

```
    similarity_top_k=SIMILARITY_TOP_K,
```

17.

```
    vector_distance_threshold=VECTOR_DISTANCE_THRESHOLD
```

18.

```
)
```

19.

How Memory Works in Practice

The typical workflow involves these steps:

1. **Session Interaction:** A user interacts with an agent via a `Session`, managed by a `SessionService`. Events are added, and state might be updated.
2. **Ingestion into Memory:** At some point (often when a session is considered complete or has yielded significant information), your application calls `memory_service.add_session_to_memory(session)`. This extracts relevant information from the session's events and adds it to the long-term knowledge store (in-memory dictionary or RAG Corpus).
3. **Later Query:** In a *different* (or the same) session, the user might ask a question requiring past context (e.g., "What did we discuss about project X last week?").
4. **Agent Uses Memory Tool:** An agent equipped with a memory-retrieval tool (like the built-in `load_memory` tool) recognizes the need for past context. It calls the tool, providing a search query (e.g., "discussion project X last week").
5. **Search Execution:** The tool internally calls `memory_service.search_memory(app_name, user_id, query)`.
6. **Results Returned:** The `MemoryService` searches its store (using keyword matching or semantic search) and returns relevant snippets as a `SearchMemoryResponse` containing a list of `MemoryResult` objects (each potentially holding events from a relevant past session).
7. **Agent Uses Results:** The tool returns these results to the agent, usually as part of the context or function response. The agent can then use this retrieved information to formulate its final answer to the user.

Example: Adding and Searching Memory

This example demonstrates the basic flow using the `InMemory` services for simplicity.

Full Code

```
import asyncio

from google.adk.agents import LlmAgent
```

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

from google.adk.memory import InMemoryMemoryService # Import MemoryService

from google.adk.runners import Runner

from google.adk.tools import load_memory # Tool to query memory

from google.genai.types import Content, Part

# --- Constants ---

APP_NAME = "memory_example_app"

USER_ID = "mem_user"
```

```
MODEL = "gemini-2.0-flash" # Use a valid model

# --- Agent Definitions ---

# Agent 1: Simple agent to capture information

info_capture_agent = LlmAgent(
    model=MODEL,
    name="InfoCaptureAgent",
    instruction="Acknowledge the user's statement.",
    # output_key="captured_info" # Could optionally save to state too
```

```
)
```

```
# Agent 2: Agent that can use memory
```

```
memory_recall_agent = LlmAgent(
```

```
    model=MODEL,
```

```
    name="MemoryRecallAgent",
```

```
    instruction="Answer the user's question. Use the 'load_memory' tool "
```

```
        "if the answer might be in past conversations.",
```

```
    tools=[load_memory] # Give the agent the tool
```

```
)
```

```
# --- Services and Runner ---
```

```
session_service = InMemorySessionService()
```

```
memory_service = InMemoryMemoryService() # Use in-memory for demo
```

```
runner = Runner(
```

```
    # Start with the info capture agent
```

```
    agent=info_capture_agent,
```

```
    app_name=APP_NAME,  
  
    session_service=session_service,  
  
    memory_service=memory_service # Provide the memory service to the Runner  
)  
  
# --- Scenario ---  
  
# Turn 1: Capture some information in a session  
  
print("--- Turn 1: Capturing Information ---")
```

```
session1_id = "session_info"

session1 = await runner.session_service.create_session(app_name=APP_NAME,
user_id=USER_ID, session_id=session1_id)

user_input1 = Content(parts=[Part(text="My favorite project is Project Alpha.")],
role="user")

# Run the agent

final_response_text = "(No final response)"

async for event in runner.run_async(user_id=USER_ID, session_id=session1_id,
new_message=user_input1):

    if event.is_final_response() and event.content and event.content.parts:
```

```
    final_response_text = event.content.parts[0].text

print(f"Agent 1 Response: {final_response_text}")

# Get the completed session

completed_session1 = await runner.session_service.get_session(app_name=APP_NAME,
user_id=USER_ID, session_id=session1_id)

# Add this session's content to the Memory Service

print("\n--- Adding Session 1 to Memory ---")

memory_service = await memory_service.add_session_to_memory(completed_session1)
```

```
print("Session added to memory.")

# Turn 2: In a *new* (or same) session, ask a question requiring memory

print("\n--- Turn 2: Recalling Information ---")

session2_id = "session_recall" # Can be same or different session ID

session2 = await runner.session_service.create_session(app_name=APP_NAME,
user_id=USER_ID, session_id=session2_id)

# Switch runner to the recall agent

runner.agent = memory_recall_agent
```

```
user_input2 = Content(parts=[Part(text="What is my favorite project?")],  
role="user")  
  
# Run the recall agent  
  
print("Running MemoryRecallAgent...")  
  
final_response_text_2 = "(No final response)"  
  
async for event in runner.run_async(user_id=USER_ID, session_id=session2_id,  
new_message=user_input2):  
  
    print(f"  Event: {event.author} - Type: {'Text' if event.content and  
event.content.parts and event.content.parts[0].text else ''}")  
  
    f"'FuncCall' if event.get_function_calls() else ''}"  
  
    f"'FuncResp' if event.get_function_responses() else ''}")
```

```
if event.is_final_response() and event.content and event.content.parts:
```

```
    final_response_text_2 = event.content.parts[0].text
```

```
    print(f"Agent 2 Final Response: {final_response_text_2}")
```

```
    break # Stop after final response
```

```
# Expected Event Sequence for Turn 2:
```

```
# 1. User sends "What is my favorite project?"
```

```
# 2. Agent (LLM) decides to call `load_memory` tool with a query like "favorite
project".
```

```
# 3. Runner executes the `load_memory` tool, which calls
`memory_service.search_memory`.
```

```
# 4. `InMemoryMemoryService` finds the relevant text ("My favorite project is Project Alpha.") from session1.
```

```
# 5. Tool returns this text in a FunctionResponse event.
```

```
# 6. Agent (LLM) receives the function response, processes the retrieved text.
```

```
# 7. Agent generates the final answer (e.g., "Your favorite project is Project Alpha."). 
```

[Previous State](#)

[Next](#)
[Callbacks: Observe, Customize, and Control Agent Behavior](#)

[Material for MkDocs](#)

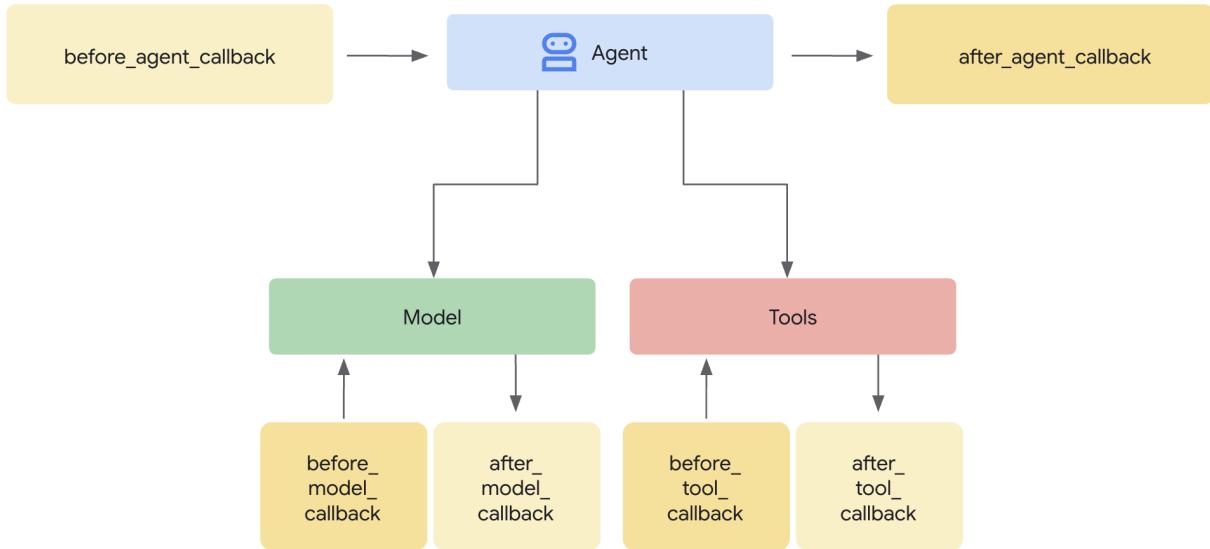
Callbacks: Observe, Customize, and Control Agent Behavior

Introduction: What are Callbacks and Why Use Them?

Callbacks are a cornerstone feature of ADK, providing a powerful mechanism to hook into an agent's execution process. They allow you to observe, customize, and even control the agent's behavior at specific, predefined points without modifying the core ADK framework code.

What are they? In essence, callbacks are standard functions that you define. You then associate these functions with an agent when you create it. The ADK framework automatically calls your functions at key stages, letting you observe or intervene. Think of it like checkpoints during the agent's process:

- **Before the agent starts its main work on a request, and after it finishes:** When you ask an agent to do something (e.g., answer a question), it runs its internal logic to figure out the response.
- The `Before Agent` callback executes *right before* this main work begins for that specific request.
- The `After Agent` callback executes *right after* the agent has finished all its steps for that request and has prepared the final result, but just before the result is returned.
- This "main work" encompasses the agent's *entire* process for handling that single request. This might involve deciding to call an LLM, actually calling the LLM, deciding to use a tool, using the tool, processing the results, and finally putting together the answer. These callbacks essentially wrap the whole sequence from receiving the input to producing the final output for that one interaction.
- **Before sending a request to, or after receiving a response from, the Large Language Model (LLM):** These callbacks (`Before Model`, `After Model`) allow you to inspect or modify the data going to and coming from the LLM specifically.
- **Before executing a tool (like a Python function or another agent) or after it finishes:** Similarly, `Before Tool` and `After Tool` callbacks give you control points specifically around the execution of tools invoked by the agent.



Why use them? Callbacks unlock significant flexibility and enable advanced agent capabilities:

- **Observe & Debug:** Log detailed information at critical steps for monitoring and troubleshooting.
- **Customize & Control:** Modify data flowing through the agent (like LLM requests or tool results) or even bypass certain steps entirely based on your logic.
- **Implement Guardrails:** Enforce safety rules, validate inputs/outputs, or prevent disallowed operations.
- **Manage State:** Read or dynamically update the agent's session state during execution.
- **Integrate & Enhance:** Trigger external actions (API calls, notifications) or add features like caching.

How are they added:

Code

The Callback Mechanism: Interception and Control

When the ADK framework encounters a point where a callback can run (e.g., just before calling the LLM), it checks if you provided a corresponding callback function for that agent. If you did, the framework executes your function.

Context is Key: Your callback function isn't called in isolation. The framework provides special **context objects** (`CallbackContext` or `ToolContext`) as arguments. These objects contain vital information about the current state of the agent's execution, including the invocation details, session state, and potentially references to services like artifacts or memory. You use these context objects to understand the situation and interact with the framework. (See the dedicated "Context Objects" section for full details).

Controlling the Flow (The Core Mechanism): The most powerful aspect of callbacks lies in how their **return value** influences the agent's subsequent actions. This is how you intercept and control the execution flow:

1. `return None (Allow Default Behavior):`

- The specific return type can vary depending on the language. In Java, the equivalent return type is `Optional.empty()`. Refer to the API documentation for language specific guidance.
- This is the standard way to signal that your callback has finished its work (e.g., logging, inspection, minor modifications to *mutable* input arguments like `llm_request`) and that the ADK agent should **proceed with its normal operation**.
- For `before_*` callbacks (`before_agent`, `before_model`, `before_tool`), returning `None` means the next step in the sequence (running the agent logic, calling the LLM, executing the tool) will occur.
- For `after_*` callbacks (`after_agent`, `after_model`, `after_tool`), returning `None` means the result just produced by the preceding step (the agent's output, the LLM's response, the tool's result) will be used as is.

2. `return <Specific Object> (Override Default Behavior):`

- Returning a *specific type of object* (instead of `None`) is how you **override** the ADK agent's default behavior. The framework will use the object you return and *skip* the step that would normally follow or *replace* the result that was just generated.
- `before_agent_callback → types.Content`: Skips the agent's main execution logic (`_run_async_impl` / `_run_live_impl`). The returned `Content` object is immediately treated as the agent's final output

- for this turn. Useful for handling simple requests directly or enforcing access control.
- `before_model_callback` → `LlmResponse`: Skips the call to the external Large Language Model. The returned `LlmResponse` object is processed as if it were the actual response from the LLM. Ideal for implementing input guardrails, prompt validation, or serving cached responses.
 - `before_tool_callback` → `dict or Map`: Skips the execution of the actual tool function (or sub-agent). The returned `dict` is used as the result of the tool call, which is then typically passed back to the LLM. Perfect for validating tool arguments, applying policy restrictions, or returning mocked/cached tool results.
 - `after_agent_callback` → `types.Content`: *Replaces* the `Content` that the agent's run logic just produced.
 - `after_model_callback` → `LlmResponse`: *Replaces* the `LlmResponse` received from the LLM. Useful for sanitizing outputs, adding standard disclaimers, or modifying the LLM's response structure.
 - `after_tool_callback` → `dict or Map`: *Replaces* the `dict` result returned by the tool. Allows for post-processing or standardization of tool outputs before they are sent back to the LLM.

Conceptual Code Example (Guardrail):

This example demonstrates the common pattern for a guardrail using `before_model_callback`.

Code

Python

Java

```
from google.adk.agents import LlmAgent

from google.adk.agents.callback_context import CallbackContext
```

```
from google.adk.models import LlmResponse, LlmRequest
```

```
from google.adk.runners import Runner
```

```
from typing import Optional
```

```
from google.genai import types
```

```
from google.adk.sessions import InMemorySessionService
```

```
GEMINI_2_FLASH="gemini-2.0-flash"
```

```
# --- Define the Callback Function ---  
  
def simple_before_model_modifier(  
    callback_context: CallbackContext, llm_request: LlmRequest  
) -> Optional[LlmResponse]:  
  
    """Inspects/modifies the LLM request or skips the call."""  
  
    agent_name = callback_context.agent_name  
  
    print(f"[Callback] Before model call for agent: {agent_name}")  
  
    # Inspect the last user message in the request contents
```

```
last_user_message = ""

if llm_request.contents and llm_request.contents[-1].role == 'user':


    if llm_request.contents[-1].parts:


        last_user_message = llm_request.contents[-1].parts[0].text


print(f"[Callback] Inspecting last user message: '{last_user_message}'")

# --- Modification Example ---

# Add a prefix to the system instruction
```

```
original_instruction = llm_request.config.system_instruction or
types.Content(role="system", parts=[])

prefix = "[Modified by Callback] "

# Ensure system_instruction is Content and parts list exists

if not isinstance(original_instruction, types.Content):

    # Handle case where it might be a string (though config expects Content)

    original_instruction = types.Content(role="system",
parts=[types.Part(text=str(original_instruction))])

if not original_instruction.parts:

    original_instruction.parts.append(types.Part(text="")) # Add an empty part
if none exist
```

```
# Modify the text of the first part

modified_text = prefix + (original_instruction.parts[0].text or "")

original_instruction.parts[0].text = modified_text

llm_request.config.system_instruction = original_instruction

print(f"[Callback] Modified system instruction to: '{modified_text}'")

# --- Skip Example ---

# Check if the last user message contains "BLOCK"
```

```
    if "BLOCK" in last_user_message.upper():

        print("[Callback] 'BLOCK' keyword found. Skipping LLM call.")

        # Return an LlmResponse to skip the actual LLM call

        return LlmResponse(
            content=types.Content(
                role="model",
                parts=[types.Part(text="LLM call was blocked by
before_model_callback.")],
            )
        )
```

```
    )

else:

    print("[Callback] Proceeding with LLM call.")

# Return None to allow the (modified) request to go to the LLM

return None

# Create LlmAgent and Assign Callback

my_llm_agent = LlmAgent(
```

```
        name="ModelCallbackAgent",  
  
        model=GEMINI_2_FLASH,  
  
        instruction="You are a helpful assistant.", # Base instruction  
  
        description="An LLM agent demonstrating before_model_callback",  
  
        before_model_callback=simple_before_model_modifier # Assign the function  
here  
    )  
  
APP_NAME = "guardrail_app"
```

```
USER_ID = "user_1"

SESSION_ID = "session_001"

# Session and Runner

session_service = InMemorySessionService()

session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)

runner = Runner(agent=my_llm_agent, app_name=APP_NAME,
session_service=session_service)
```

```
# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_message=content)

    for event in events:

        if event.is_final_response():

            final_response = event.content.parts[0].text
```

```
    print("Agent Response: ", final_response)

call_agent("callback example")
```

By understanding this mechanism of returning `None` versus returning specific objects, you can precisely control the agent's execution path, making callbacks an essential tool for building sophisticated and reliable agents with ADK.

Types of Callbacks

The framework provides different types of callbacks that trigger at various stages of an agent's execution. Understanding when each callback fires and what context it receives is key to using them effectively.

Agent Lifecycle Callbacks

These callbacks are available on *any* agent that inherits from `BaseAgent` (including `LlmAgent`, `SequentialAgent`, `ParallelAgent`, `LoopAgent`, etc).

Note

The specific method names or return types may vary slightly by SDK language (e.g., return `None` in Python, return `Optional.empty()` or `Maybe.empty()` in Java). Refer to the language-specific API documentation for details.

Before Agent Callback

When: Called *immediately before* the agent's `_run_async_impl` (or `_run_live_impl`) method is executed. It runs after the agent's `InvocationContext` is created but *before* its core logic begins.

Purpose: Ideal for setting up resources or state needed only for this specific agent's run, performing validation checks on the session state (`callback_context.state`) before execution starts, logging the entry point of the agent's activity, or potentially modifying the invocation context before the core logic uses it.

Code

Python

Java

```
# # --- Setup Instructions ---
```

```
# # 1. Install the ADK package:
```

```
# !pip install google-adk
```

```
# # Make sure to restart kernel if using colab/jupyter notebooks
```

```
# # 2. Set up your Gemini API Key:
```

```
# # - Get a key from Google AI Studio: https://aistudio.google.com/app/apikey
```

```
# # - Set it as an environment variable:
```

```
# import os
```

```
# os.environ["GOOGLE_API_KEY"] = "YOUR_API_KEY_HERE" # <--- REPLACE with your  
actual key
```

```
# # Or learn about other authentication methods (like Vertex AI):
```

```
# # https://google.github.io/adk-docs/agents/models/

# ADK Imports

from google.adk.agents import LlmAgent

from google.adk.agents.callback_context import CallbackContext

from google.adk.runners import InMemoryRunner # Use InMemoryRunner

from google.genai import types # For types.Content

from typing import Optional
```

```
# Define the model - Use the specific model name requested
```

```
GEMINI_2_FLASH="gemini-2.0-flash"
```

```
# --- 1. Define the Callback Function ---
```

```
def check_if_agent_should_run(callback_context: CallbackContext) ->
Optional[types.Content]:
```

```
    """
```

```
    Logs entry and checks 'skip_llm_agent' in session state.
```

```
If True, returns Content to skip the agent's execution.
```

```
If False or not present, returns None to allow execution.
```

```
    ...
```

```
    agent_name = callback_context.agent_name
```

```
    invocation_id = callback_context.invocation_id
```

```
    current_state = callback_context.state.to_dict()
```

```
    print(f"\n[Callback] Entering agent: {agent_name} (Inv: {invocation_id})")
```

```
    print(f"[Callback] Current State: {current_state}")
```

```
# Check the condition in session state dictionary

if current_state.get("skip_llm_agent", False):

    print(f"[Callback] State condition 'skip_llm_agent=True' met: Skipping
agent {agent_name}.")"

# Return Content to skip the agent's run

return types.Content(
    parts=[types.Part(text=f"Agent {agent_name} skipped by
before_agent_callback due to state.")],
    role="model" # Assign model role to the overriding response
)
```

```
    else:

        print(f"[Callback] State condition not met: Proceeding with agent
{agent_name}.")

    # Return None to allow the LlmAgent's normal execution

    return None

# --- 2. Setup Agent with Callback ---

llm_agent_with_before_cb = LlmAgent(
    name="MyControlledAgent",
```

```
    model=GEMINI_2_FLASH,  
  
    instruction="You are a concise assistant.",  
  
    description="An LLM agent demonstrating stateful before_agent_callback",  
  
    before_agent_callback=check_if_agent_should_run # Assign the callback  
  
)  
  
# --- 3. Setup Runner and Sessions using InMemoryRunner ---  
  
async def main():  
  
    app_name = "before_agent_demo"
```

```
user_id = "test_user"

session_id_run = "session_will_run"

session_id_skip = "session_will_skip"

# Use InMemoryRunner - it includes InMemorySessionService

runner = InMemoryRunner(agent=llm_agent_with_before_cb, app_name=app_name)

# Get the bundled session service to create sessions

session_service = runner.session_service
```

```
# Create session 1: Agent will run (default empty state)

    session_service.create_session(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id_run
    )

    # No initial state means 'skip_llm_agent' will be False in the callback
    check

)
```

```
# Create session 2: Agent will be skipped (state has skip_llm_agent=True)

    session_service.create_session(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id_skip,
        state={"skip_llm_agent": True} # Set the state flag here
    )
```

```
# --- Scenario 1: Run where callback allows agent execution ---  
  
    print("\n" + "="*20 + f" SCENARIO 1: Running Agent on Session  
'{session_id_run}' (Should Proceed) " + "="*20)  
  
    async for event in runner.run_async(  
  
        user_id=user_id,  
  
        session_id=session_id_run,  
  
        new_message=types.Content(role="user", parts=[types.Part(text="Hello,  
please respond.")])  
  
    ):  
  
        # Print final output (either from LLM or callback override)
```

```
    if event.is_final_response() and event.content:  
  
        print(f"Final Output: [{event.author}]  
{event.content.parts[0].text.strip()}")  
  
    elif event.is_error():  
  
        print(f"Error Event: {event.error_details}")  
  
# --- Scenario 2: Run where callback intercepts and skips agent ---  
  
    print("\n" + "="*20 + f" SCENARIO 2: Running Agent on Session  
'{session_id_skip}' (Should Skip) " + "="*20)  
  
async for event in runner.run_async()
```

```
    user_id=user_id,  
  
    session_id=session_id_skip,  
  
    new_message=types.Content(role="user", parts=[types.Part(text="This message  
won't reach the LLM.")])  
):  
  
# Print final output (either from LLM or callback override)  
  
if event.is_final_response() and event.content:  
  
    print(f"Final Output: [{event.author}]  
{event.content.parts[0].text.strip()}")  
  
elif event.is_error():
```

```
    print(f"Error Event: {event.error_details}")

# --- 4. Execute ---

# In a Python script:

# import asyncio

# if __name__ == "__main__":
#     # Make sure GOOGLE_API_KEY environment variable is set if not using Vertex
#     AI auth

#     # Or ensure Application Default Credentials (ADC) are configured for Vertex
#     AI
```

```
#     asyncio.run(main())

# In a Jupyter Notebook or similar environment:

await main()
```

Note on the `before_agent_callback` Example:

- **What it Shows:** This example demonstrates the `before_agent_callback`. This callback runs *right before* the agent's main processing logic starts for a given request.
- **How it Works:** The callback function (`check_if_agent_should_run`) looks at a flag (`skip_llm_agent`) in the session's state.
 - a. If the flag is `True`, the callback returns a `types.Content` object. This tells the ADK framework to **skip** the agent's main execution entirely and use the callback's returned content as the final response.
 - b. If the flag is `False` (or not set), the callback returns `None` or an empty object. This tells the ADK framework to **proceed** with the agent's normal execution (calling the LLM in this case).
- **Expected Outcome:** You'll see two scenarios:
 - a. In the session *with* the `skip_llm_agent : True` state, the agent's LLM call is bypassed, and the output comes directly from the callback ("Agent... skipped...").
 - b. In the session *without* that state flag, the callback allows the agent to run, and you see the actual response from the LLM (e.g., "Hello!").
- **Understanding Callbacks:** This highlights how `before_ callbacks` act as **gatekeepers**, allowing you to intercept execution *before* a major step and potentially prevent it based on checks (like state, input validation, permissions).

After Agent Callback

When: Called *immediately after* the agent's `_run_async_impl` (or `_run_live_impl`) method successfully completes. It does *not* run if the agent was skipped due to `before_agent_callback` returning content or if `end_invocation` was set during the agent's run.

Purpose: Useful for cleanup tasks, post-execution validation, logging the completion of an agent's activity, modifying final state, or augmenting/replacing the agent's final output.

Code

Python

Java

```
# # --- Setup Instructions ---
# # 1. Install the ADK package:
# !pip install google-adk
# # Make sure to restart kernel if using colab/jupyter notebooks
```

```
# # 2. Set up your Gemini API Key:
```

```
# #     - Get a key from Google AI Studio: https://aistudio.google.com/app/apikey
```

```
# #     - Set it as an environment variable:
```

```
# import os
```

```
# os.environ[ "GOOGLE_API_KEY" ] = "YOUR_API_KEY_HERE" # <--- REPLACE with your  
actual key
```

```
# # Or learn about other authentication methods (like Vertex AI):
```

```
# # https://google.github.io/adk-docs/agents/models/
```

```
# ADK Imports

from google.adk.agents import LlmAgent

from google.adk.agents.callback_context import CallbackContext

from google.adk.runners import InMemoryRunner # Use InMemoryRunner

from google.genai import types # For types.Content

from typing import Optional

# Define the model - Use the specific model name requested
```

```
GEMINI_2_FLASH="gemini-2.0-flash"
```

```
# --- 1. Define the Callback Function ---
```

```
def modify_output_after_agent(callback_context: CallbackContext) ->
Optional[types.Content]:
```

```
    ...
```

```
Logs exit from an agent and checks 'add_concluding_note' in session state.
```

```
If True, returns new Content to *replace* the agent's original output.
```

```
If False or not present, returns None, allowing the agent's original output to
be used.
```

```
    ...
```

```
    agent_name = callback_context.agent_name
```

```
    invocation_id = callback_context.invocation_id
```

```
    current_state = callback_context.state.to_dict()
```

```
    print(f"\n[Callback] Exiting agent: {agent_name} (Inv: {invocation_id})")
```

```
    print(f"[Callback] Current State: {current_state}")
```

```
# Example: Check state to decide whether to modify the final output

if current_state.get("add_concluding_note", False):

    print(f"[Callback] State condition 'add_concluding_note=True' met:
Replacing agent {agent_name}'s output.")

# Return Content to *replace* the agent's own output

return types.Content(
    parts=[types.Part(text=f"Concluding note added by after_agent_callback,
replacing original output.")],
    role="model" # Assign model role to the overriding response
)
```

```
    else:

        print(f"[Callback] State condition not met: Using agent {agent_name}'s
original output.")

    # Return None - the agent's output produced just before this callback will
be used.

    return None

# --- 2. Setup Agent with Callback ---

llm_agent_with_after_cb = LlmAgent(
    name="MySimpleAgentWithAfter",
```

```
    model=GEMINI_2_FLASH,  
  
    instruction="You are a simple agent. Just say 'Processing complete!' ",  
  
    description="An LLM agent demonstrating after_agent_callback for output  
modification",  
  
    after_agent_callback=modify_output_after_agent # Assign the callback here  
  
)
```

```
# --- 3. Setup Runner and Sessions using InMemoryRunner ---
```

```
async def main():  
  
    app_name = "after_agent_demo"
```

```
user_id = "test_user_after"

session_id_normal = "session_run_normally"

session_id_modify = "session_modify_output"

# Use InMemoryRunner - it includes InMemorySessionService

runner = InMemoryRunner(agent=llm_agent_with_after_cb, app_name=app_name)

# Get the bundled session service to create sessions

session_service = runner.session_service
```

```
# Create session 1: Agent output will be used as is (default empty state)

    session_service.create_session(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id_normal
        # No initial state means 'add_concluding_note' will be False in the
        # callback check
    )

# print(f"Session '{session_id_normal}' created with default state.")
```

```
# Create session 2: Agent output will be replaced by the callback

    session_service.create_session(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id_modify,
        state={"add_concluding_note": True} # Set the state flag here
    )
```

```
# print(f"Session '{session_id_modify}' created with
state={{'add_concluding_note': True}}.)"

# --- Scenario 1: Run where callback allows agent's original output ---

print("\n" + "="*20 + f" SCENARIO 1: Running Agent on Session
'{session_id_normal}' (Should Use Original Output) " + "="*20)

async for event in runner.run_async(
    user_id=user_id,
    session_id=session_id_normal,
```

```
    new_message=types.Content(role="user", parts=[types.Part(text="Process this
please.")])  
  
):  
  
# Print final output (either from LLM or callback override)  
  
if event.is_final_response() and event.content:  
  
    print(f"Final Output: [{event.author}]
{event.content.parts[0].text.strip()}")  
  
elif event.is_error():  
  
    print(f"Error Event: {event.error_details}")
```

```
# --- Scenario 2: Run where callback replaces the agent's output ---  
  
    print("\n" + "="*20 + f" SCENARIO 2: Running Agent on Session  
'{session_id_modify}' (Should Replace Output) " + "="*20)  
  
    async for event in runner.run_async(  
  
        user_id=user_id,  
  
        session_id=session_id_modify,  
  
        new_message=types.Content(role="user", parts=[types.Part(text="Process this  
and add note.")])  
  
    ):  
  
        # Print final output (either from LLM or callback override)
```

```
    if event.is_final_response() and event.content:

        print(f"Final Output: [{event.author}]
{event.content.parts[0].text.strip()}")

    elif event.is_error():

        print(f"Error Event: {event.error_details}")

# --- 4. Execute ---

# In a Python script:

# import asyncio

# if __name__ == "__main__":

```

```
#      # Make sure GOOGLE_API_KEY environment variable is set if not using Vertex
#      AI auth

#      # Or ensure Application Default Credentials (ADC) are configured for Vertex
#      AI

#      asyncio.run(main())

# In a Jupyter Notebook or similar environment:

await main()
```

Note on the `after_agent_callback` Example:

- **What it Shows:** This example demonstrates the `after_agent_callback`. This callback runs *right after* the agent's main processing logic has finished and produced its result, but *before* that result is finalized and returned.
- **How it Works:** The callback function (`modify_output_after_agent`) checks a flag (`add_concluding_note`) in the session's state.

- a. If the flag is `True`, the callback returns a `new types.Content` object. This tells the ADK framework to **replace** the agent's original output with the content returned by the callback.
- b. If the flag is `False` (or not set), the callback returns `None` or an empty object. This tells the ADK framework to **use** the original output generated by the agent.
- **Expected Outcome:** You'll see two scenarios:
 - a. In the session *without* the `add_concluding_note: True` state, the callback allows the agent's original output ("Processing complete!") to be used.
 - b. In the session *with* that state flag, the callback intercepts the agent's original output and replaces it with its own message ("Concluding note added...").
- **Understanding Callbacks:** This highlights how `after_` callbacks allow **post-processing** or **modification**. You can inspect the result of a step (the agent's run) and decide whether to let it pass through, change it, or completely replace it based on your logic.

LLM Interaction Callbacks

These callbacks are specific to `LlmAgent` and provide hooks around the interaction with the Large Language Model.

Before Model Callback

When: Called just before the `generate_content_async` (or equivalent) request is sent to the LLM within an `LlmAgent`'s flow.

Purpose: Allows inspection and modification of the request going to the LLM. Use cases include adding dynamic instructions, injecting few-shot examples based on state, modifying model config, implementing guardrails (like profanity filters), or implementing request-level caching.

Return Value Effect:

If the callback returns `None` (or a `Maybe.empty()` object in Java), the LLM continues its normal

workflow. If the callback returns an `LlmResponse` object, then the call to the LLM is **skipped**. The returned `LlmResponse` is used directly as if it came from the model. This is powerful for implementing guardrails or caching.

Code

Python

Java

```
from google.adk.agents import LlmAgent

from google.adk.agents.callback_context import CallbackContext

from google.adk.models import LlmResponse, LlmRequest

from google.adk.runners import Runner

from typing import Optional

from google.genai import types
```

```
from google.adk.sessions import InMemorySessionService
```

```
GEMINI_2_FLASH="gemini-2.0-flash"
```

```
# --- Define the Callback Function ---
```

```
def simple_before_model_modifier(
```

```
    callback_context: CallbackContext, llm_request: LlmRequest
```

```
) -> Optional[LlmResponse]:
```

```
"""Inspects/modifies the LLM request or skips the call."""
```

```
agent_name = callback_context.agent_name
```

```
print(f"[Callback] Before model call for agent: {agent_name}")
```

```
# Inspect the last user message in the request contents
```

```
last_user_message = ""
```

```
if llm_request.contents and llm_request.contents[-1].role == 'user':
```

```
    if llm_request.contents[-1].parts:
```

```
        last_user_message = llm_request.contents[-1].parts[0].text
```

```
print(f"[Callback] Inspecting last user message: '{last_user_message}'")  
  
# --- Modification Example ---  
  
# Add a prefix to the system instruction  
  
original_instruction = llm_request.config.system_instruction or  
types.Content(role="system", parts=[])  
  
prefix = "[Modified by Callback] "  
  
# Ensure system_instruction is Content and parts list exists  
  
if not isinstance(original_instruction, types.Content):
```

```
# Handle case where it might be a string (though config expects Content)

    original_instruction = types.Content(role="system",
parts=[types.Part(text=str(original_instruction))])

if not original_instruction.parts:

    original_instruction.parts.append(types.Part(text="")) # Add an empty part
if none exist

# Modify the text of the first part

modified_text = prefix + (original_instruction.parts[0].text or "")

original_instruction.parts[0].text = modified_text
```

```
    llm_request.config.system_instruction = original_instruction

    print(f"[Callback] Modified system instruction to: '{modified_text}'")

# --- Skip Example ---

# Check if the last user message contains "BLOCK"

if "BLOCK" in last_user_message.upper():

    print("[Callback] 'BLOCK' keyword found. Skipping LLM call.")

# Return an LlmResponse to skip the actual LLM call

return LlmResponse()
```

```
        content=types.Content(  
  
            role="model",  
  
            parts=[types.Part(text="LLM call was blocked by  
before_model_callback.")],  
  
        )  
  
    )  
  
else:  
  
    print("[Callback] Proceeding with LLM call.")  
  
    # Return None to allow the (modified) request to go to the LLM
```

```
    return None
```

```
# Create LlmAgent and Assign Callback
```

```
my_llm_agent = LlmAgent(
```

```
    name="ModelCallbackAgent",
```

```
    model=GEMINI_2_FLASH,
```

```
    instruction="You are a helpful assistant.", # Base instruction
```

```
    description="An LLM agent demonstrating before_model_callback",
```

```
    before_model_callback=simple_before_model_modifier # Assign the function  
here
```

```
)
```

```
APP_NAME = "guardrail_app"
```

```
USER_ID = "user_1"
```

```
SESSION_ID = "session_001"
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()

session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
                                         session_id=SESSION_ID)

runner = Runner(agent=my_llm_agent, app_name=APP_NAME,
                session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])
```

```
events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_message=content)

for event in events:

    if event.is_final_response():

        final_response = event.content.parts[0].text

        print("Agent Response: ", final_response)

call_agent("callback example")
```

After Model Callback

When: Called just after a response (`LlmResponse`) is received from the LLM, before it's processed further by the invoking agent.

Purpose: Allows inspection or modification of the raw LLM response. Use cases include

- logging model outputs,
- reformatting responses,
- censoring sensitive information generated by the model,
- parsing structured data from the LLM response and storing it in `callback_context.state`
- or handling specific error codes.

Code

Python

Java

```
from google.adk.agents import LlmAgent

from google.adk.agents.callback_context import CallbackContext

from google.adk.runners import Runner

from typing import Optional

from google.genai import types
```

```
from google.adk.sessions import InMemorySessionService

from google.adk.models import LlmResponse

GEMINI_2_FLASH="gemini-2.0-flash"

# --- Define the Callback Function ---

def simple_after_model_modifier(
    callback_context: CallbackContext, llm_response: LlmResponse
```

```
) -> Optional[LLmResponse]:  
  
    """Inspects/modifies the LLM response after it's received."""  
  
    agent_name = callback_context.agent_name  
  
    print(f"[Callback] After model call for agent: {agent_name}")  
  
    # --- Inspection ---  
  
    original_text = ""  
  
    if llm_response.content and llm_response.content.parts:  
  
        # Assuming simple text response for this example
```

```
if llm_response.content.parts[0].text:

    original_text = llm_response.content.parts[0].text

    print(f"[Callback] Inspected original response text:
'{original_text[:100]}...'"") # Log snippet

elif llm_response.content.parts[0].function_call:

    print(f"[Callback] Inspected response: Contains function call
'{llm_response.content.parts[0].function_call.name}'. No text modification.")

return None # Don't modify tool calls in this example

else:

    print("[Callback] Inspected response: No text content found.")
```

```
        return None

    elif llm_response.error_message:

        print(f"[Callback] Inspected response: Contains error
'{llm_response.error_message}'. No modification.")

        return None

    else:

        print("[Callback] Inspected response: Empty LlmResponse.")

        return None # Nothing to modify
```

```
# --- Modification Example ---

# Replace "joke" with "funny story" (case-insensitive)

search_term = "joke"

replace_term = "funny story"

if search_term in original_text.lower():

    print(f"[Callback] Found '{search_term}'. Modifying response.")

modified_text = original_text.replace(search_term, replace_term)

modified_text = modified_text.replace(search_term.capitalize(),
replace_term.capitalize()) # Handle capitalization
```

```
# Create a NEW LlmResponse with the modified content

# Deep copy parts to avoid modifying original if other callbacks exist

modified_parts = [copy.deepcopy(part) for part in
llm_response.content.parts]

modified_parts[0].text = modified_text # Update the text in the copied part

new_response = LlmResponse(
    content=types.Content(role="model", parts=modified_parts),

    # Copy other relevant fields if necessary, e.g., grounding_metadata
```

```
        grounding_metadata=llm_response.grounding_metadata

    )

print(f"[Callback] Returning modified response.")

return new_response # Return the modified response

else:

    print(f"[Callback] '{search_term}' not found. Passing original response
through.")

# Return None to use the original llm_response

return None
```

```
# Create LlmAgent and Assign Callback

my_llm_agent = LlmAgent(
    name="AfterModelCallbackAgent",
    model=GEMINI_2_FLASH,
    instruction="You are a helpful assistant.",
    description="An LLM agent demonstrating after_model_callback",
    after_model_callback=simple_after_model_modifier # Assign the function here
```

```
)
```

```
APP_NAME = "guardrail_app"
```

```
USER_ID = "user_1"
```

```
SESSION_ID = "session_001"
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()
```

```
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)
```

```
runner = Runner(agent=my_llm_agent, app_name=APP_NAME,
session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_message=content)
```

```
for event in events:

    if event.is_final_response():

        final_response = event.content.parts[0].text

        print("Agent Response: ", final_response)

call_agent("callback example")
```

Tool Execution Callbacks

These callbacks are also specific to `LlmAgent` and trigger around the execution of tools (including `FunctionTool`, `AgentTool`, etc.) that the LLM might request.

Before Tool Callback

When: Called just before a specific tool's `run_async` method is invoked, after the LLM has generated a function call for it.

Purpose: Allows inspection and modification of tool arguments, performing authorization checks before execution, logging tool usage attempts, or implementing tool-level caching.

Return Value Effect:

1. If the callback returns `None` (or a `Maybe.empty()` object in Java), the tool's `run_async` method is executed with the (potentially modified) `args`.
2. If a dictionary (or `Map` in Java) is returned, the tool's `run_async` method is **skipped**. The returned dictionary is used directly as the result of the tool call. This is useful for caching or overriding tool behavior.

Code

Python

Java

```
from google.adk.agents import LlmAgent
```

```
from google.adk.runners import Runner
```

```
from typing import Optional
```

```
from google.genai import types
```

```
from google.adk.sessions import InMemorySessionService

from google.adk.tools import FunctionTool

from google.adk.tools.tool_context import ToolContext

from google.adk.tools.base_tool import BaseTool

from typing import Dict, Any

GEMINI_2_FLASH="gemini-2.0-flash"
```

```
def get_capital_city(country: str) -> str:

    """Retrieves the capital city of a given country."""

    print(f"--- Tool 'get_capital_city' executing with country: {country} ---")

    country_capitals = {

        "united states": "Washington, D.C.",

        "canada": "Ottawa",

        "france": "Paris",

        "germany": "Berlin",
```

```
    }

    return country_capitals.get(country.lower(), f"Capital not found for
{country}")

capital_tool = FunctionTool(func=get_capital_city)

def simple_before_tool_modifier(
    tool: BaseTool, args: Dict[str, Any], tool_context: ToolContext
) -> Optional[Dict]:
    """Inspects/modifies tool args or skips the tool call."""

```

```
agent_name = tool_context.agent_name

tool_name = tool.name

print(f"[Callback] Before tool call for tool '{tool_name}' in agent
'{agent_name}'")

print(f"[Callback] Original args: {args}")

if tool_name == 'get_capital_city' and args.get('country', '').lower() ==
'canada':


    print("[Callback] Detected 'Canada'. Modifying args to 'France'.")  

    args['country'] = 'France'
```

```
    print(f"[Callback] Modified args: {args}")
```

```
    return None
```

```
# If the tool is 'get_capital_city' and country is 'BLOCK'
```

```
if tool_name == 'get_capital_city' and args.get('country', '').upper() ==  
'BLOCK':
```

```
    print("[Callback] Detected 'BLOCK'. Skipping tool execution.")
```

```
    return {"result": "Tool execution was blocked by before_tool_callback."}
```

```
    print("[Callback] Proceeding with original or previously modified args.")

    return None

my_llm_agent = LlmAgent(
    name="ToolCallbackAgent",
    model=GEMINI_2_FLASH,
    instruction="You are an agent that can find capital cities. Use the
get_capital_city tool.",
    description="An LLM agent demonstrating before_tool_callback",
    tools=[capital_tool],
```

```
    before_tool_callback=simple_before_tool_modifier
```

```
)
```

```
APP_NAME = "guardrail_app"
```

```
USER_ID = "user_1"
```

```
SESSION_ID = "session_001"
```

```
# Session and Runner
```

```
session_service = InMemorySessionService()

session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
                                         session_id=SESSION_ID)

runner = Runner(agent=my_llm_agent, app_name=APP_NAME,
                session_service=session_service)

# Agent Interaction

def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])
```

```
events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_message=content)

for event in events:

    if event.is_final_response():

        final_response = event.content.parts[0].text

        print("Agent Response: ", final_response)

call_agent("callback example")
```

After Tool Callback

When: Called just after the tool's `run_async` method completes successfully.

Purpose: Allows inspection and modification of the tool's result before it's sent back to the LLM (potentially after summarization). Useful for logging tool results, post-processing or formatting results, or saving specific parts of the result to the session state.

Return Value Effect:

1. If the callback returns `None` (or a `Maybe.empty()` object in Java), the original `tool_response` is used.
2. If a new dictionary is returned, it **replaces** the original `tool_response`. This allows modifying or filtering the result seen by the LLM.

Code

Python

Java

```
# Copyright 2025 Google LLC

#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# 
```

```
#      http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
from google.adk.agents import LlmAgent
```

```
from google.adk.runners import Runner
```

```
from typing import Optional
```

```
from google.genai import types

from google.adk.sessions import InMemorySessionService

from google.adk.tools import FunctionTool

from google.adk.tools.tool_context import ToolContext

from google.adk.tools.base_tool import BaseTool

from typing import Dict, Any

from copy import deepcopy

GEMINI_2_FLASH="gemini-2.0-flash"

# --- Define a Simple Tool Function (Same as before) ---
```

```
def get_capital_city(country: str) -> str:

    """Retrieves the capital city of a given country."""

    print(f"--- Tool 'get_capital_city' executing with country: {country} ---")

    country_capitals = {

        "united states": "Washington, D.C.",

        "canada": "Ottawa",

        "france": "Paris",

        "germany": "Berlin",

    }

    return {"result": country_capitals.get(country.lower(), f"Capital not found for {country}")}
```

```
# --- Wrap the function into a Tool ---

capital_tool = FunctionTool(func=get_capital_city)

# --- Define the Callback Function ---

def simple_after_tool_modifier(
    tool: BaseTool, args: Dict[str, Any], tool_context: ToolContext, tool_response: Dict
) -> Optional[Dict]:
    """Inspects/modifies the tool result after execution."""

    agent_name = tool_context.agent_name

    tool_name = tool.name
```

```
    print(f"[Callback] After tool call for tool '{tool_name}' in agent  
' {agent_name}'")  
  
    print(f"[Callback] Args used: {args}")  
  
    print(f"[Callback] Original tool_response: {tool_response}")  
  
# Default structure for function tool results is {"result": <return_value>}  
  
original_result_value = tool_response.get("result", "")  
  
# original_result_value = tool_response  
  
# --- Modification Example ---  
  
# If the tool was 'get_capital_city' and result is 'Washington, D.C.'  
  
if tool_name == 'get_capital_city' and original_result_value == "Washington,  
D.C.":
```

```
    print("[Callback] Detected 'Washington, D.C.'. Modifying tool response.")
```

```
# IMPORTANT: Create a new dictionary or modify a copy
```

```
modified_response = deepcopy(tool_response)
```

```
    modified_response["result"] = f"{original_result_value} (Note: This is the  
capital of the USA)."
```

```
    modified_response["note_added_by_callback"] = True # Add extra info if  
needed
```

```
    print(f"[Callback] Modified tool_response: {modified_response}")
```

```
return modified_response # Return the modified dictionary
```

```
    print("[Callback] Passing original tool response through.")

    # Return None to use the original tool_response

    return None

# Create LlmAgent and Assign Callback

my_llm_agent = LlmAgent(
    name="AfterToolCallbackAgent",
    model=GEMINI_2_FLASH,
    instruction="You are an agent that finds capital cities using the
get_capital_city tool. Report the result clearly.",
    description="An LLM agent demonstrating after_tool_callback",
```

```
    tools=[capital_tool], # Add the tool

    after_tool_callback=simple_after_tool_modifier # Assign the callback

)

APP_NAME = "guardrail_app"

USER_ID = "user_1"

SESSION_ID = "session_001"

# Session and Runner

session_service = InMemorySessionService()

session = await session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)
```

```
runner = Runner(agent=my_llm_agent, app_name=APP_NAME,
session_service=session_service)

# Agent Interaction

async def call_agent(query):

    content = types.Content(role='user', parts=[types.Part(text=query)])

    events = runner.run_async(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    async for event in events:

        if event.is_final_response():
```

```
final_response = event.content.parts[0].text

print("Agent Response: ", final_response)

await call_agent("united states")
```

[Previous](#)

[Callbacks: Observe, Customize, and Control Agent Behavior](#)

[Next](#)

[Callback patterns](#)

[Material for MkDocs](#)

Design Patterns and Best Practices for Callbacks

Callbacks offer powerful hooks into the agent lifecycle. Here are common design patterns illustrating how to leverage them effectively in ADK, followed by best practices for implementation.

Design Patterns

These patterns demonstrate typical ways to enhance or control agent behavior using callbacks:

1. Guardrails & Policy Enforcement

- **Pattern:** Intercept requests before they reach the LLM or tools to enforce rules.
- **How:** Use `before_model_callback` to inspect the `LlmRequest` prompt or `before_tool_callback` to inspect tool arguments. If a policy violation is detected (e.g., forbidden topics, profanity), return a predefined response (`LlmResponse` or `dict/ Map`) to block the operation and optionally update `context.state` to log the violation.
- **Example:** A `before_model_callback` checks `llm_request.contents` for sensitive keywords and returns a standard "Cannot process this request" `LlmResponse` if found, preventing the LLM call.

2. Dynamic State Management

- **Pattern:** Read from and write to session state within callbacks to make agent behavior context-aware and pass data between steps.
- **How:** Access `callback_context.state` or `tool_context.state`. Modifications (`state['key'] = value`) are automatically tracked in the subsequent `Event.actions.state_delta` for persistence by the `SessionService`.
- **Example:** An `after_tool_callback` saves a `transaction_id` from the tool's result to `tool_context.state['last_transaction_id']`. A later `before_agent_callback` might read `state['user_tier']` to customize the agent's greeting.

3. Logging and Monitoring

- **Pattern:** Add detailed logging at specific lifecycle points for observability and debugging.

- **How:** Implement callbacks (e.g., `before_agent_callback`, `after_tool_callback`, `after_model_callback`) to print or send structured logs containing information like agent name, tool name, invocation ID, and relevant data from the context or arguments.
- **Example:** Log messages like `INFO: [Invocation: e-123] Before Tool: search_api - Args: {'query': 'ADK'}`.

4. Caching

- **Pattern:** Avoid redundant LLM calls or tool executions by caching results.
- **How:** In `before_model_callback` or `before_tool_callback`, generate a cache key based on the request/arguments. Check `context.state` (or an external cache) for this key. If found, return the cached `LlmResponse` or result directly, skipping the actual operation. If not found, allow the operation to proceed and use the corresponding `after_callback` (`after_model_callback`, `after_tool_callback`) to store the new result in the cache using the key.
- **Example:** `before_tool_callback` for `get_stock_price(symbol)` checks `state[f"cache:stock:{symbol}"]`. If present, returns the cached price; otherwise, allows the API call and `after_tool_callback` saves the result to the state key.

5. Request/Response Modification

- **Pattern:** Alter data just before it's sent to the LLM/tool or just after it's received.
- **How:**
 - `before_model_callback`: Modify `llm_request` (e.g., add system instructions based on `state`).
 - `after_model_callback`: Modify the returned `LlmResponse` (e.g., format text, filter content).
 - `before_tool_callback`: Modify the tool `args` dictionary (or Map in Java).
 - `after_tool_callback`: Modify the `tool_response` dictionary (or Map in Java).
- **Example:** `before_model_callback` appends "User language preference: Spanish" to `llm_request.config.system_instruction` if `context.state['lang'] == 'es'`.

6. Conditional Skipping of Steps

- **Pattern:** Prevent standard operations (agent run, LLM call, tool execution) based on certain conditions.
- **How:** Return a value from a `before_` callback (Content from `before_agent_callback`, `LlmResponse` from `before_model_callback`, `dict` from `before_tool_callback`). The framework interprets this returned value as the result for that step, skipping the normal execution.
- **Example:** `before_tool_callback` checks `tool_context.state['api_quota_exceeded']`. If `True`, it returns `{'error': 'API quota exceeded'}`, preventing the actual tool function from running.

7. Tool-Specific Actions (Authentication & Summarization Control)

- **Pattern:** Handle actions specific to the tool lifecycle, primarily authentication and controlling LLM summarization of tool results.
- **How:** Use `ToolContext` within tool callbacks (`before_tool_callback`, `after_tool_callback`).
 - **Authentication:** Call `tool_context.request_credential(auth_config)` in `before_tool_callback` if credentials are required but not found (e.g., via `tool_context.get_auth_response` or state check). This initiates the auth flow.
 - **Summarization:** Set `tool_context.actions.skip_summarization = True` if the raw dictionary output of the tool should be passed back to the LLM or potentially displayed directly, bypassing the default LLM summarization step.
- **Example:** A `before_tool_callback` for a secure API checks for an auth token in state; if missing, it calls `request_credential`. An `after_tool_callback` for a tool returning structured JSON might set `skip_summarization = True`.

8. Artifact Handling

- **Pattern:** Save or load session-related files or large data blobs during the agent lifecycle.
- **How:** Use `callback_context.save_artifact` / `await tool_context.save_artifact` to store data (e.g., generated reports, logs, intermediate data). Use `load_artifact` to retrieve previously stored artifacts. Changes are tracked via `Event.actions.artifact_delta`.
- **Example:** An `after_tool_callback` for a "generate_report" tool saves the output file using `await tool_context.save_artifact("report.pdf", report_part)`. A `before_agent_callback` might load a configuration artifact using `callback_context.load_artifact("agent_config.json")`.

Best Practices for Callbacks

- **Keep Focused:** Design each callback for a single, well-defined purpose (e.g., just logging, just validation). Avoid monolithic callbacks.
- **Mind Performance:** Callbacks execute synchronously within the agent's processing loop. Avoid long-running or blocking operations (network calls, heavy computation). Offload if necessary, but be aware this adds complexity.
- **Handle Errors Gracefully:** Use `try...except/ catch` blocks within your callback functions. Log errors appropriately and decide if the agent invocation should halt or attempt recovery. Don't let callback errors crash the entire process.
- **Manage State Carefully:**
 - Be deliberate about reading from and writing to `context.state`. Changes are immediately visible within the *current* invocation and persisted at the end of the event processing.
 - Use specific state keys rather than modifying broad structures to avoid unintended side effects.
 - Consider using state prefixes (`State.APP_PREFIX`, `State.USER_PREFIX`, `State.TEMP_PREFIX`) for clarity, especially with persistent `SessionService` implementations.
- **Consider Idempotency:** If a callback performs actions with external side effects (e.g., incrementing an external counter), design it to be idempotent (safe to run multiple times with the same input) if possible, to handle potential retries in the framework or your application.

- **Test Thoroughly:** Unit test your callback functions using mock context objects. Perform integration tests to ensure callbacks function correctly within the full agent flow.
- **Ensure Clarity:** Use descriptive names for your callback functions. Add clear docstrings explaining their purpose, when they run, and any side effects (especially state modifications).
- **Use Correct Context Type:** Always use the specific context type provided (`CallbackContext` for agent/model, `ToolContext` for tools) to ensure access to the appropriate methods and properties.

By applying these patterns and best practices, you can effectively use callbacks to create more robust, observable, and customized agent behaviors in ADK.