

[Skip to  
content](#)



[Create](#)

[Home](#)

[Competitions](#)

[Datasets](#)

[Models](#)

[Benchmarks](#)

[Game Arena](#)

[Code](#)

[Discussions](#)

[Learn](#)

[More](#)

[View Active Events](#)

# **5-Day AI Agents Intensive Course with Google**



# Welcome to our 5-Day AI Agents Intensive Course with Google!

## What is the 5-Day AI Agents Intensive?

This 5-day online course was crafted by Google's ML researchers and engineers to help developers explore the foundations and practical applications of AI agents. You'll learn the core components – models, tools, orchestration, memory and evaluation. Finally, you'll discover how agents move beyond LLM prototypes to become production-ready systems.

Each day blends conceptual deep dives with hands-on examples, codelabs, and live discussions. By the end, you'll be ready to build, evaluate, and deploy agents that solve real-world problems.

**This page serves as the central hub for all course materials, updates and resources.**



## How the Course Works:

The course is designed to be flexible and interactive, allowing you to learn at your own pace while benefiting from live sessions and community engagement.

- **Daily Content Release:** New assignments will be posted each day on this Kaggle course page - your primary source for all course materials.
- **Assignment Notifications (Fast Follow):** Given the large number of participants, links to all assignment materials including whitepapers, codelabs and podcast episodes — will also be shared on the Kaggle Discord and sent via follow-up email shortly after being posted.
- **Support & Discussion:** Join the discussion on our dedicated Discord channels to connect with other learners, ask questions, and get support from Google researchers and engineers who will be monitoring the channels throughout the week.
- **Daily Livestreams:** Join Kanchana Patlolla and Anant Nawalgaria live each day starting Monday, November 10th at 11 AM PT / 8 PM CET / 12:30 AM IST on Kaggle's YouTube channel. They will be joined by special guests from Google, NVIDIA, Cohere and more. After each session, recordings will be shared on Discord.
- **Course Completion:** To get the most out of this intensive, we recommend completing all course materials including whitepapers, podcasts and codelabs before starting the capstone project. Assignments are provided for your learning, so assignment submissions are not required and you can complete them at your own pace.
- **Capstone Project:** On the final day, you'll have the chance to apply everything you've learned by building your own AI agent. By participating in the capstone project, you will earn a badge on Kaggle.

- **Community:** We want this community to be positive and supportive. Please follow Kaggle's community guidelines found [here](#).



## Setup Instructions

To ensure you are ready for the course, please complete these essential setup steps:

- Kaggle account: Sign up for a Kaggle account and learn how Notebooks work. Make sure to phone verify your account, it's necessary for the course's codelabs.  
AI Studio account: Sign up for an AI Studio account and ensure you can generate an API key.  
Kaggle Discord: Sign up for a Discord account and join us on the Kaggle Discord server. We have the following channels dedicated to this event:

#5dgai-announcements: find official course announcements and livestream recordings.

#5dgai-introductions: introduce yourself and meet other participants from around the world.

#5dgai-question-forum: Discord forum-style channel for asking questions and discussions about the assignments.

#5dgai-general-chat: a general channel to discuss course materials and network with other participants.

Please note that if you would like to post on other channels on the Kaggle discord you will need to link your Kaggle account to discord here:

<https://kaggle.com/discord/confirmation>.



## Day 1 (Introduction to Agents)

Today's whitepaper introduces AI agents. It presents a taxonomy of agent capabilities, emphasizes the need for an Agent Ops discipline for reliability and governance, and discusses the importance of agent interoperability and security through identity and constrained policies.

In today's codelabs, you'll be building your first AI agent and your first multi-agent system, using Agent Development Kit (ADK), powered by Gemini, and giving it the ability to use Google Search to answer questions with up-to-date information. In the second codelab, the focus will be on multi-agent systems, where you'll learn how to create teams of specialized agents and explore different architectural patterns.

## Day 1 Assignments

1. Complete the Unit 1 – “Introduction to Agents”:

- Listen to the summary podcast episode for this unit, created by NotebookLM.  
To complement the podcast, read the “Introduction to Agents” whitepaper  
Complete these codelabs on Kaggle:

Build your first agent using Gemini and ADK.

Build your first multi-agent systems using ADK.

Make sure you phone verify your Kaggle account before starting, it's necessary for the codelabs.

We also have a troubleshooting guide for the codelabs. Be sure to check there for solutions to common problems.

Want to have an interactive conversation? Try adding the whitepapers to NotebookLM.



## **Day 2 (Agent Tools & Interoperability with Model Context Protocol (MCP))**

Today's whitepaper focuses on external tools functions that allow an agent to perform actions or retrieve real-time data beyond its training set and introduces best practices for designing effective tools. You'll learn about MCP, highlighting its architectural components, communication layer, risks and enterprise readiness gaps.

In today's codelabs, you will create custom tools for your agents by turning your own Python functions into actions your agent can perform. You'll also use MCP and implement long-running operations where an agent can pause tool calls while waiting for human approval, before resuming.

## Day 2 Assignments

Complete Unit 2 - “Agent Tools & Interoperability with Model Context Protocol (MCP)”:

- Listen to the summary podcast episode for this unit, created by NotebookLM.  
To complement the podcast, read the “Agent Tools & Interoperability with Model Context Protocol (MCP)” whitepaper.  
Complete these codelabs on Kaggle:

Explore new ways to add tools to extend what your agents can do.

Explore best practices for tools, including using MCP and long-running operations.

Want to have an interactive conversation? Try adding the whitepapers to NotebookLM.



## **Day 3 (Context Engineering: Sessions & Memory)**

This whitepaper explores context engineering as the practice of dynamically assembling and managing information within an agent's context window to create stateful and personalized AI experiences. It defines Sessions as the container for a single, immediate conversation's history, and Memory as the long-term persistence mechanism.

In the codelabs, you will learn how to make agents stateful by managing conversation history through context engineering in ADK, and working memory within a session, allowing your agent to remember context and have coherent, multi-turn conversations. In the second notebook, you'll give your agent long-term memory that persists across different sessions.

### **Day 3 Assignment**

Complete Unit 3 - “Context Engineering: Sessions & Memory”:

- Listen to the summary podcast episode for this unit, created by NotebookLM.  
To complement the podcast, read the “Context Engineering: Sessions & Memory whitepaper”.  
Complete these codelabs on Kaggle:

Build stateful agents and perform context engineering.

Explore how to use memory with your agent.

Want to have an interactive conversation? Try adding the whitepapers to NotebookLM.



## Day 4 (Agent Quality)

This whitepaper addresses the challenge of assuring quality in AI agents by introducing a holistic evaluation framework. The necessary technical foundation for this is Observability, built on three pillars: Logs (the diary), Traces (the narrative), and Metrics (the health report), enabling a continuous feedback loop using scalable methods like LLM-as-a-Judge and Human-in-the-Loop (HITL) evaluation.

For today's codelabs, you'll learn how to use logs, traces, and metrics to get full visibility into your agent's decision-making process, allowing you to debug failures and understand why your agent behaves the way it does. In the

second codelab, you'll learn how to evaluate your agents to score your agent's response quality and tool usage.

## Day 4 Assignment

Complete Unit 4 - “Agent Quality”:

- Listen to the summary podcast episode for this unit, created by NotebookLM.  
To complement the podcast, read the Agent Quality whitepaper.  
Complete these codelabs on Kaggle:

Implement observability to help you debug your agents.

Evaluate your agents.



## Day 5 (Prototype to Production)

This whitepaper provides a technical guide to the operational lifecycle of AI agents, focusing on deployment, scaling and productionization. It explores the challenges of transitioning agentic systems from prototypes to enterprise-grade solutions, with special attention to Agent2Agent (A2A) Protocol.

For today's codelabs, you'll learn how to build systems of multiple, independent agents that can communicate and collaborate using A2A Protocol. You'll also learn how to take your agent from your local machine to a production-ready, scalable service, by deploying your agent to Vertex AI Agent Engine on Google Cloud.

## **Final Assignment**

Complete Unit 5 - “Prototype to Production”:

- Listen to the summary podcast episode for this unit.  
To complement the podcast, read the “Prototype to Production” whitepaper.  
Complete these codelabs on Kaggle:

Explore how to use A2A Protocol to have agents interact with each other.  
[Optional] Deploy your agent to Agent Engine on Google Cloud.



## **Final Reminders and Announcements**

Congrats on completing the 5-day AI Agents Intensive course!

-  **Capstone Project:** You'll create an AI agents project showcasing a use case that leverages some of the key capabilities learned throughout this course. The top 12 winners will receive Kaggle swag and have their work featured on Kaggle's social media platforms. More details about the Capstone Project, including the evaluation and submission process will be shared in an email later today. Participation in the Capstone Project is optional.
-  **Kaggle badge and certificate:** Due to high demand, participants in the Capstone Project are eligible to earn a badge and certificate on their Kaggle profile. These will be added to all eligible profiles by the end of December 2025.
-  **Look out for the Kaggle Learn Guide:** We are actively working to transition this content into a convenient Kaggle Learn Guide. We will be sharing this updated resource with you as soon as it is ready.
-  **GEAR:** If you'd like to continue building on what you learned in the Agents Intensive, Google's upcoming Gemini Enterprise Agent Ready (GEAR) initiative offers a deeper dive into learning, building and deploying AI agents. GEAR launches in early 2026 - learn more [here](#).

**Check out the Agents Intensive - Capstone Project [here](#).**

Copyright 2025 Google LLC.

```
# @title 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  
#  
# https://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.
```



## Your First AI Agent: From Prompt to Action

Welcome to the Kaggle 5-day Agents course!

This notebook is your first step into building AI agents. An agent can do more than just respond to a prompt — it can **take actions** to find information or get things done.

In this notebook, you'll:

-  Install [Agent Development Kit \(ADK\)](#)
-  Configure your API key to use the Gemini model
-  Build your first simple agent
-  Run your agent and watch it use a tool (like Google Search) to answer a question

## !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

**Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

**X Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the **Copy & Edit** button in the top-right corner.

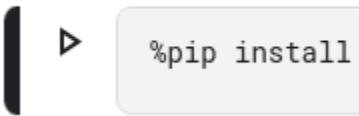


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the  Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

## Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires authentication.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

## 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

## 3. Authenticate in the notebook

Run the cell below to complete authentication.

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Gemini API key setup complete.")
except Exception as e:
    print(
        f"⚠️ Authentication Error: Please make sure you have added
'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

✅ Gemini API key setup complete.

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from google.adk.agents import Agent
from google.adk.models.google_llm import Gemini
from google.adk.runners import InMemoryRunner
from google.adk.tools import google_search
from google.genai import types

print("✅ ADK components imported successfully.")
```

✅ ADK components imported successfully.

#### 1.4: Helper functions

We'll define some helper functions. If you are running this outside the Kaggle environment, you don't need to do this.

```
# Define helper functions that will be reused throughout the notebook

from IPython.core.display import display, HTML
from jupyter_server.serverapp import list_running_servers

# Gets the proxied URL in the Kaggle Notebooks environment
def get_adk_proxy_url():
    PROXY_HOST = "https://kkb-production.jupyter-proxy.kaggle.net"
    ADK_PORT = "8000"

    servers = list(list_running_servers())
    if not servers:
        raise Exception("No running Jupyter servers found.")

    baseURL = servers[0]["base_url"]
```

```

try:
    path_parts = baseURL.split("/")
    kernel = path_parts[2]
    token = path_parts[3]
except IndexError:
    raise Exception(f"Could not parse kernel/token from base URL: {baseURL}")

url_prefix = f"/k/{kernel}/{token}/proxy/proxy/{ADK_PORT}"
url = f"{PROXY_HOST}{url_prefix}"

styled_html = f"""
<div style="padding: 15px; border: 2px solid #f0ad4e; border-radius: 8px; background-color: #fef9f0; margin: 20px 0;">
    <div style="font-family: sans-serif; margin-bottom: 12px; color: #333; font-size: 1.1em;">
        <strong>⚠️ IMPORTANT: Action Required</strong>
    </div>
    <div style="font-family: sans-serif; margin-bottom: 15px; color: #333; line-height: 1.5;">
        The ADK web UI is <strong>not running yet</strong>. You must start it in the next cell.
        <ol style="margin-top: 10px; padding-left: 20px;">
            <li style="margin-bottom: 5px;"><strong>Run the next cell</strong> (the one with <code>!adk web ...</code>) to start the ADK web UI.</li>
            <li style="margin-bottom: 5px;">Wait for that cell to show it is "Running" (it will not "complete").</li>
            <li>Once it's running, <strong>return to this button</strong> and click it to open the UI.</li>
        </ol>
        <em style="font-size: 0.9em; color: #555;">(If you click the button before running the next cell, you will get a 500 error.)</em>
    </div>
    <a href='{url}' target='_blank' style="display: inline-block; background-color: #1a73e8; color: white; padding: 10px 20px; text-decoration: none; border-radius: 25px; font-family: sans-serif; font-weight: 500;">

```

```
        box-shadow: 0 2px 5px rgba(0,0,0,0.2); transition: all 0.2s  
        ease; ">  
        Open ADK Web UI (after running cell below) ^  
    </a>  
  </div>  
  """  
  
display(HTML(styled_html))  
  
return url_prefix  
  
print("✅ Helper functions defined.")
```

✅ Helper functions defined.

## 1.5: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config=types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1, # Initial delay before first retry (in seconds)  
    http_status_codes=[429, 500, 503, 504] # Retry on these HTTP errors  
)
```

---

## 🤖 Section 2: Your first AI Agent with ADK

## 2.1 What is an AI Agent?

You've probably used an LLM like Gemini before, where you give it a prompt and it gives you a text response.

Prompt -> LLM -> Text

An AI Agent takes this one step further. An agent can think, take actions, and observe the results of those actions to give you a better answer.

Prompt -> Agent -> Thought -> Action -> Observation -> Final Answer

In this notebook, we'll build an agent that can take the action of searching Google. Let's see the difference!

## 2.2 Define your agent

Now, let's build our agent. We'll configure an Agent by setting its key properties, which tell it what to do and how to operate.

To learn more, check out the documentation related to [agents in ADK](#).

These are the main properties we'll set:

- **name and description:** A simple name and description to identify our agent.
- **model:** The specific LLM that will power the agent's reasoning. We'll use "gemini-2.5-flash-lite".
- **instruction:** The agent's guiding prompt. This tells the agent what its goal is and how to behave.
- **tools:** A list of [tools](#) that the agent can use. To start, we'll give it the google\_search tool, which lets it find up-to-date information online.

```
root_agent = Agent(  
    name="helpful_assistant",  
    model=Gemini(  
        model="gemini-2.5-flash-lite",  
        retry_options=retry_config  
    ),
```

```
        description="A simple agent that can answer general questions.",
        instruction="You are a helpful assistant. Use Google Search for
current info or if unsure.",
        tools=[google_search],
)

print("✅ Root Agent defined.")
```

✅ Root Agent defined.

## 2.3 Run your agent

Now it's time to bring your agent to life and send it a query. To do this, you need a [Runner](#), which is the central component within ADK that acts as the orchestrator. It manages the conversation, sends our messages to the agent, and handles its responses.

### a. Create an InMemoryRunner and tell it to use our root\_agent:

```
runner = InMemoryRunner(agent=root_agent)

print("✅ Runner created.")
```

✅ Runner created.

👉 Note that we are using the Python Runner directly in this notebook. You can also run agents using ADK command-line tools such as `adk run`, `adk web`, or `adk api_server`. To learn more, check out the documentation related to [runtime in ADK](#).

### b. Now you can call the `.run_debug()` method to send our prompt and get an answer.

👉 This method abstracts the process of session creation and maintenance and is used in prototyping. We'll explore "what sessions are and how to create them" on Day 3.

```
response = await runner.run_debug(  
    "What is Agent Development Kit from Google? What languages is the SDK  
available in?"  
)
```

```
### Created new session: debug_session_id
```

User > What is Agent Development Kit from Google? What languages is the SDK available in?

helpful\_assistant > The Agent Development Kit (ADK) from Google is a flexible, modular, and open-source framework designed to simplify the development, deployment, and orchestration of AI agents and multi-agent systems. It applies software development principles to AI agent creation, making it feel more like traditional software development and allowing for robust debugging, versioning, and deployment. The ADK is optimized for the Gemini models and the Google ecosystem but is also model-agnostic and compatible with other frameworks.

The SDK is available in the following languages:

- \* \*\*Python\*\*
- \* \*\*Java\*\*
- \* \*\*Go\*\*

You can see a summary of ADK and its available languages in the response.

## 2.4 How does it work?

The agent performed a Google Search to get the latest information about ADK, and it knew to use this tool because:

1. The agent inspects and is aware of which tools it has available to use.

2. The agent's instructions specify the use of the search tool to get current information or if it is unsure of an answer.

The best way to see the full, detailed trace of the agent's thoughts and actions is in the **ADK web UI**, which we'll set up later in this notebook.

And we'll cover more detailed workflows for logging and observability later in the course.

## 2.5 Your Turn!

This is your chance to see the agent in action. Ask it a question that requires current information.

Try one of these, or make up your own:

- What's the weather in London?
- Who won the last soccer world cup?
- What new movies are showing in theaters now?

```
response = await runner.run_debug("What's the weather in London?")
```

```
### Continue session: debug_session_id
```

User > What's the weather in London?

helpful\_assistant > The weather in London, UK is currently clear with a temperature of 50°F (10°C) and 89% humidity. There is a 0% chance of rain today.

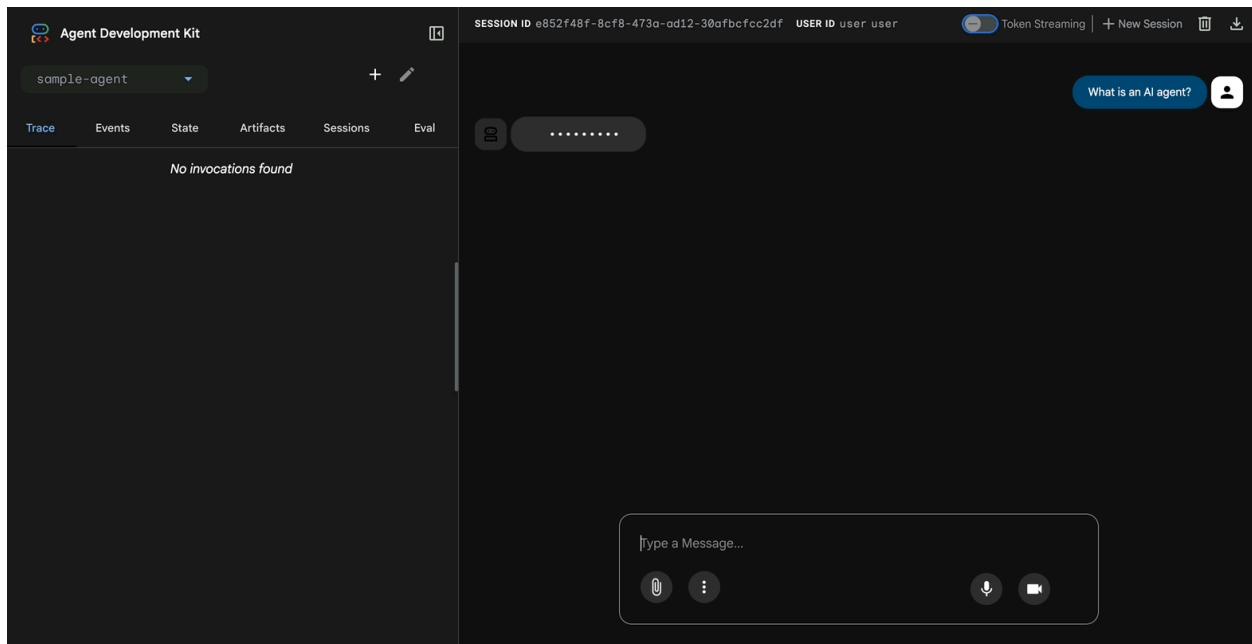
Looking ahead, the forecast for the next few days includes chances of rain, with temperatures generally ranging between the low 40s°F and high 50s°F. Some sources indicate light rain and cloudy conditions for the upcoming week.

---

## Section 3: Try the ADK Web Interface

### Overview

ADK includes a built-in web interface for interactively chatting with, testing, and debugging your agents.



To use the ADK web UI, you'll need to create an agent with Python files using the `adk create` command.

Run the command below to generate a `sample-agent` folder that contains all the necessary files, including `agent.py` for your code, an `.env` file with your API key pre-configured, and an `__init__.py` file:

```
!adk create sample-agent --model gemini-2.5-flash-lite --api_key  
$GOOGLE_API_KEY
```

```
Agent created in /kaggle/working/sample-agent:
```

```
- .env  
- __init__.py  
- agent.py
```

Get your custom URL to access the ADK web UI in the Kaggle Notebooks environment:

```
url_prefix = get_adk_proxy_url()
```

### **IMPORTANT: Action Required**

The ADK web UI is **not running yet**. You must start it in the next cell.

1. **Run the next cell** (the one with `!adk web ...`) to start the ADK web UI.
2. Wait for that cell to show it is "Running" (it will not "complete").
3. Once it's running, **return to this button** and click it to open the UI.

*(If you click the button before running the next cell, you will get a 500 error.)*

[Open ADK Web UI \(after running cell below\) ↗](#)

Now we can run ADK web:

```
!adk web --url_prefix {url_prefix}
```

```
/usr/local/lib/python3.11/dist-packages/google/adk/cli/fast_api.py:130:  
UserWarning: [EXPERIMENTAL] InMemoryCredentialService: This feature is
```

experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.

```
credential_service = InMemoryCredentialService()
/usr/local/lib/python3.11/dist-packages/google/adk/auth/credential_service
/in_memory_credential_service.py:33: UserWarning: [EXPERIMENTAL]
BaseCredentialService: This feature is experimental and may change or be
removed in future versions without notice. It may introduce breaking
changes at any time.

super().__init__()

INFO:      Started server process [93]
INFO:      Waiting for application startup.

+-----+
| ADK Web Server started
|
|
|
| For local testing, access at http://127.0.0.1:8000.
|
+-----+
| ADK Web Server shutting down...
|
+-----+  
INFO:      Application shutdown complete.
INFO:      Finished server process [93]
```

Aborted!

Now you can access the ADK dev UI using the link above.

Once you open the link, you'll see the ADK web interface where you can ask your ADK agent questions.

Note: This sample agent does not have any tools enabled (like Google Search). It is a basic agent designed specifically to let you explore the UI features.

**!! IMPORTANT: DO NOT SHARE THE PROXY LINK** with anyone - treat it as sensitive data as it contains your authentication token in the URL.

---



## Congratulations!

You've built and run your first agent with ADK! You've just seen the core concept of agent development in action.

The big takeaway is that your agent didn't just *respond*—it **reasoned** that it needed more information and then **acted** by using a tool. This ability to take action is the foundation of all agent-based AI.



### Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.



### Learn More

Refer to the following documentation to learn more:

- [ADK Documentation](#)
- [ADK Quickstart for Python](#)

- [ADK Agents Overview](#)
- [ADK Tools Overview](#)

## ⌚ Next Steps

Ready for the next challenge? Continue to the next notebook to learn how to **architect multi-agent systems.**

---

Authors
<a href="#">Kristopher Overholt</a>

*Copyright 2025 Google LLC.*

```
# @title 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
#
# https://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.
```

## Multi-Agent Systems & Workflow Patterns

Welcome to the Kaggle 5-day Agents course!

In the previous notebook, you built a **single agent** that could take action. Now, you'll learn how to scale up by building **agent teams**.

Just like a team of people, you can create specialized agents that collaborate to solve complex problems. This is called a **multi-agent system**, and it's one of the most powerful concepts in AI agent development.

In this notebook, you'll:

- Learn when to use multi-agent systems in [Agent Development Kit \(ADK\)](#)
- Build your first system using an LLM as a "manager"
- Learn three core workflow patterns (Sequential, Parallel, and Loop) to coordinate your agent teams

## !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. [See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

## 1. Verify Your Account (Required)

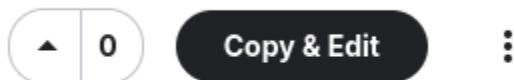
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

## 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy` and `Edit` button in the top-right corner.



This creates a private copy of the notebook just for you.

## 3. Run Code Cells

Once you have your copy, you can run code.

Click the Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

## 4. If You Get Stuck

To restart: Select `Factory reset` from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

## Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires authentication.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

#### 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

#### 3. Authenticate in the notebook

Run the cell below to complete authentication.

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✓ Gemini API key setup complete.")
except Exception as e:
    print(
        f"⚠️ Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

✓ Gemini API key setup complete.

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from google.adk.agents import Agent, SequentialAgent, ParallelAgent,
LoopAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import InMemoryRunner
from google.adk.tools import AgentTool, FunctionTool, google_search
from google.genai import types

print("✓ ADK components imported successfully.")
```

✓ ADK components imported successfully.

## 1.4: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config=types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```

## 🤔 Section 2: Why Multi-Agent Systems? + Your First Multi-Agent

### The Problem: The "Do-It-All" Agent

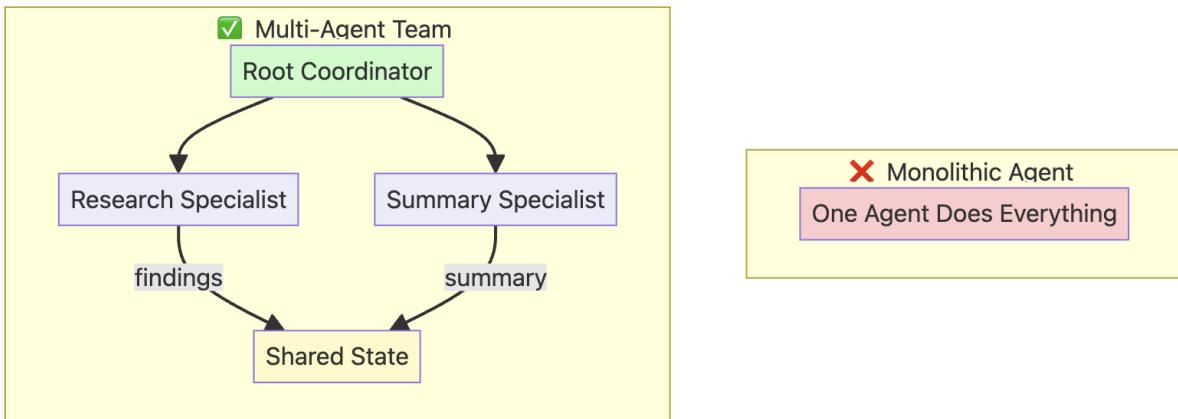
Single agents can do a lot. But what happens when the task gets complex? A single "monolithic" agent that tries to do research, writing, editing, and fact-checking all at once becomes a problem. Its instruction prompt gets long and confusing. It's hard to debug (which part failed?), difficult to maintain, and often produces unreliable results.

### The Solution: A Team of Specialists

Instead of one "do-it-all" agent, we can build a **multi-agent system**. This is a team of simple, specialized agents that collaborate, just like a real-world team. Each agent has one clear job (e.g., one agent *only* does research, another *only* writes). This makes them easier to build, easier to test, and much more powerful and reliable when working together.

To learn more, check out the documentation related to [LLM agents in ADK](#).

### Architecture: Single Agent vs Multi-Agent Team



## 2.1 Example: Research & Summarization System

Let's build a system with two specialized agents:

1. **Research Agent** - Searches for information using Google Search
2. **Summarizer Agent** - Creates concise summaries from research findings

```
# Research Agent: Its job is to use the google_search tool and present
# findings.
research_agent = Agent(
    name="ResearchAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a specialized research agent. Your only job is
    to use the
        google_search tool to find 2-3 pieces of relevant information on the
    given topic and present the findings with citations.""",
    tools=[google_search],
    output_key="research_findings", # The result of this agent will be
    stored in the session state with this key.
)

print("✓ research_agent created.")
```

✓ research\_agent created.

```
# Summarizer Agent: Its job is to summarize the text it receives.
summarizer_agent = Agent(
    name="SummarizerAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # The instruction is modified to request a bulleted list for a clear
    output format.
    instruction="""Read the provided research findings:
{research_findings}
Create a concise summary as a bulleted list with 3-5 key points.""",
    output_key="final_summary",
)
print("✓ summarizer_agent created.")
```

✓ summarizer\_agent created.

Refer to the ADK documentation for more information on [guiding agents with clear and specific instructions](#).

Then we bring the agents together under a root agent, or coordinator:

```
# Root Coordinator: Orchestrates the workflow by calling the sub-agents as
tools.
root_agent = Agent(
    name="ResearchCoordinator",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
),
```

```
# This instruction tells the root agent HOW to use its tools (which are
# the other agents).
instruction="""You are a research coordinator. Your goal is to answer
the user's query by orchestrating a workflow.
1. First, you MUST call the `ResearchAgent` tool to find relevant
information on the topic provided by the user.
2. Next, after receiving the research findings, you MUST call the
`SummarizerAgent` tool to create a concise summary.
3. Finally, present the final summary clearly to the user as your
response."""
# We wrap the sub-agents in `AgentTool` to make them callable tools for
# the root agent.
tools=[AgentTool(research_agent), AgentTool(summarizer_agent)],
)
print("✅ root_agent created.")
```

```
✅ root_agent created.
```

Here we're using `AgentTool` to wrap the sub-agents to make them callable tools for the root agent.

We'll explore `AgentTool` in-detail on Day 2.

Let's run the agent and ask it about a topic:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "What are the latest advancements in quantum computing and what do
    they mean for AI?"
)
```

```
### Created new session: debug_session_id
```

User > What are the latest advancements in quantum computing and what do  
they mean for AI?

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

ResearchCoordinator > Quantum computing is poised to revolutionize AI by offering unprecedented computational power and efficiency. Key advancements include enhanced processing speeds through qubits and superposition, enabling faster AI model training and the ability to solve complex problems currently intractable for classical computers. This convergence promises breakthroughs in areas like drug discovery, materials science, and personalized medicine. While major tech companies are heavily investing in the field, challenges such as qubit stability and data processing limitations remain.

You've just built your first multi-agent system! You used a single "coordinator" agent to manage the workflow, which is a powerful and flexible pattern.

!! However, **relying on an LLM's instructions to control the order can sometimes be unpredictable**. Next, we'll explore a different pattern that gives you guaranteed, step-by-step execution.

---

## ➡️ Section 3: Sequential Workflows - The Assembly Line

**The Problem: Unpredictable Order**

The previous multi-agent system worked, but it relied on a **detailed instruction prompt** to force the LLM to run steps in order. This can be unreliable. A complex LLM might decide to skip a step, run them in the wrong order, or get "stuck," making the process unpredictable.

### The Solution: A Fixed Pipeline

When you need tasks to happen in a **guaranteed, specific order**, you can use a `SequentialAgent`. This agent acts like an assembly line, running each sub-agent in the exact order you list them. The output of one agent automatically becomes the input for the next, creating a predictable and reliable workflow.

**Use Sequential when:** Order matters, you need a linear pipeline, or each step builds on the previous one.

To learn more, check out the documentation related to [sequential agents in ADK](#).

### Architecture: Blog Post Creation Pipeline



### 3.1 Example: Blog Post Creation with Sequential Agents

Let's build a system with three specialized agents:

1. **Outline Agent** - Creates a blog outline for a given topic
2. **Writer Agent** - Writes a blog post
3. **Editor Agent** - Edits a blog post draft for clarity and structure

```
# Outline Agent: Creates the initial blog post outline.
outline_agent = Agent(
    name="OutlineAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Create a blog outline for the given topic with:
```

```
1. A catchy headline
2. An introduction hook
3. 3-5 main sections with 2-3 bullet points for each
4. A concluding thought""",
output_key="blog_outline", # The result of this agent will be stored
in the session state with this key.
)

print("✓ outline_agent created.")
```

```
✓ outline_agent created.
```

```
# Writer Agent: Writes the full blog post based on the outline from the
previous agent.
writer_agent = Agent(
    name="WriterAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # The `'{blog_outline}'` placeholder automatically injects the state
    value from the previous agent's output.
    instruction="""Following this outline strictly: {blog_outline}
    Write a brief, 200 to 300-word blog post with an engaging and
    informative tone.""",
    output_key="blog_draft", # The result of this agent will be stored
    with this key.
)

print("✓ writer_agent created.")
```

```
✓ writer_agent created.
```

```
# Editor Agent: Edits and polishes the draft from the writer agent.
```

```
editor_agent = Agent(  
    name="EditorAgent",  
    model=Gemini(  
        model="gemini-2.5-flash-lite",  
        retry_options=retry_config  
    ),  
    # This agent receives the '{blog_draft}' from the writer agent's  
    output.  
    instruction="""Edit this draft: {blog_draft}  
    Your task is to polish the text by fixing any grammatical errors,  
    improving the flow and sentence structure, and enhancing overall  
    clarity.""" ,  
    output_key="final_blog", # This is the final output of the entire  
    pipeline.  
)  
  
print("✅ editor_agent created.")
```

```
✅ editor_agent created.
```

Then we bring the agents together under a sequential agent, which runs the agents in the order that they are listed:

```
root_agent = SequentialAgent(  
    name="BlogPipeline",  
    sub_agents=[outline_agent, writer_agent, editor_agent],  
)  
  
print("✅ Sequential Agent created.")
```

```
✅ Sequential Agent created.
```

Let's run the agent and give it a topic to write a blog post about:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Write a blog post about the benefits of multi-agent systems for
    software developers"
)
```

```
### Created new session: debug_session_id

User > Write a blog post about the benefits of multi-agent systems for
software developers
OutlineAgent > Hello! OutlineAgent here, ready to help you craft a
compelling blog post outline.
```

Here's a structure for your blog post on the benefits of multi-agent systems for software developers:

```
## **Headline:** Supercharge Your Code: Why Multi-Agent Systems Are Your
Next Developer Superpower
```

\*\*Introduction Hook:\*\* Imagine a team of tireless, intelligent assistants working in parallel, tackling complex tasks, debugging with uncanny speed, and optimizing your code in ways you never thought possible. This isn't science fiction anymore; it's the reality of multi-agent systems (MAS) and they're poised to revolutionize how software developers work.

---

```
### **Main Section 1: Enhanced Problem Solving and Complexity Management**
```

- \* \*\*Divide and Conquer, Smarter:\*\* MAS can break down large, intricate problems into smaller, manageable sub-problems, assigning them to specialized agents. This distributed approach makes tackling highly complex systems significantly more feasible and efficient.
- \* \*\*Intelligent Delegation:\*\* Developers can leverage agents to handle routine, repetitive, or computationally intensive tasks, freeing up their cognitive bandwidth for higher-level design, creativity, and strategic thinking.

- \* \*\*Adaptive and Robust Solutions:\*\* Agents can communicate, coordinate, and adapt their strategies in real-time, leading to more resilient and flexible software that can better handle unexpected changes or failures.

### ### \*\*Main Section 2: Accelerated Development Cycles and Increased Productivity\*\*

- \* \*\*Automated Testing and Debugging:\*\* Agents can be trained to autonomously identify, isolate, and even suggest fixes for bugs, drastically reducing the time spent on manual testing and debugging.
- \* \*\*Code Generation and Optimization:\*\* MAS can assist in generating boilerplate code, refactoring existing code, and identifying areas for performance optimization, leading to faster development timelines.
- \* \*\*Continuous Learning and Improvement:\*\* Agents can learn from past interactions and data, continuously improving their performance and providing developers with better insights and assistance over time.

### ### \*\*Main Section 3: Fostering Collaboration and Innovation\*\*

- \* \*\*Simulating Complex Interactions:\*\* MAS can be used to model and simulate real-world scenarios with multiple interacting entities, providing developers with a powerful tool for understanding system behavior and user interactions before deployment.
- \* \*\*Exploration of Novel Architectures:\*\* The agent-based paradigm encourages thinking about software in terms of decentralized, autonomous units, potentially leading to the development of entirely new and innovative software architectures.
- \* \*\*Democratizing Advanced Capabilities:\*\* MAS can abstract away complex algorithms and techniques, making advanced capabilities like AI and machine learning more accessible and easier to integrate into traditional software development.

---

\*\*Concluding Thought:\*\* Multi-agent systems are more than just a new tool; they represent a paradigm shift in how we approach software development. By embracing their power for problem-solving, productivity, and innovation, developers can unlock unprecedented levels of efficiency and creativity, ultimately building better software, faster.

## WriterAgent > ## Supercharge Your Code: Why Multi-Agent Systems Are Your Next Developer Superpower

Imagine a team of tireless, intelligent assistants working in parallel, tackling complex tasks, debugging with uncanny speed, and optimizing your code in ways you never thought possible. This isn't science fiction anymore; it's the reality of multi-agent systems (MAS) and they're poised to revolutionize how software developers work.

At their core, MAS offer a powerful approach to \*\*enhanced problem-solving and complexity management\*\*. Think of it as a smarter "divide and conquer." These systems can break down massive, intricate problems into smaller, manageable sub-problems, assigning them to specialized agents. This distributed approach makes tackling highly complex systems significantly more feasible and efficient. Developers can intelligently delegate routine, repetitive, or computationally intensive tasks to these agents, freeing up their own cognitive bandwidth for higher-level design, creativity, and strategic thinking. Furthermore, agents can communicate and adapt their strategies in real-time, leading to more resilient and flexible software.

The impact on \*\*accelerated development cycles and increased productivity\*\* is profound. MAS can automate testing and debugging, autonomously identifying, isolating, and even suggesting fixes for bugs, drastically reducing manual effort. They can also assist in generating boilerplate code, refactoring existing code, and identifying performance bottlenecks, all contributing to faster development timelines. Crucially, these agents can learn from past interactions, continuously improving their performance and providing developers with ever-better insights.

Beyond efficiency, MAS fosters \*\*collaboration and innovation\*\*. They can simulate complex interactions, offering a powerful tool for understanding system behavior before deployment. The agent-based paradigm encourages exploration of novel architectures, thinking in terms of decentralized, autonomous units. This also democratizes advanced capabilities like AI and machine learning, making them more accessible for integration.

Multi-agent systems are more than just a new tool; they represent a paradigm shift. By embracing their power, developers can unlock

unprecedented levels of efficiency and creativity, ultimately building better software, faster.

EditorAgent > ## Supercharge Your Code: Why Multi-Agent Systems Are Your Next Developer Superpower

Imagine a team of tireless, intelligent assistants working in parallel, tackling complex tasks, debugging with uncanny speed, and optimizing your code in ways you never thought possible. This isn't science fiction anymore; it's the reality of multi-agent systems (MAS), and they're poised to revolutionize how software developers work.

At their core, MAS offer a powerful approach to \*\*enhanced problem-solving and complexity management\*\*. Think of it as a smarter "divide and conquer." These systems can break down massive, intricate problems into smaller, manageable sub-problems, assigning them to specialized agents. This distributed approach makes tackling highly complex systems significantly more feasible and efficient. Developers can intelligently delegate routine, repetitive, or computationally intensive tasks to these agents, freeing up their own cognitive bandwidth for higher-level design, creativity, and strategic thinking. Furthermore, agents can communicate and adapt their strategies in real-time, leading to more resilient and flexible software.

The impact on \*\*accelerated development cycles and increased productivity\*\* is profound. MAS can automate testing and debugging, autonomously identifying, isolating, and even suggesting fixes for bugs, drastically reducing manual effort. They can also assist in generating boilerplate code, refactoring existing code, and identifying performance bottlenecks, all contributing to faster development timelines. Crucially, these agents can learn from past interactions, continuously improving their performance and providing developers with ever-better insights.

Beyond efficiency, MAS fosters \*\*collaboration and innovation\*\*. They can simulate complex interactions, offering a powerful tool for understanding system behavior before deployment. The agent-based paradigm encourages the exploration of novel architectures, promoting a mindset of decentralized, autonomous units. This also democratizes advanced capabilities like AI and machine learning, making them more accessible for integration.

Multi-agent systems are more than just a new tool; they represent a paradigm shift. By embracing their power, developers can unlock unprecedented levels of efficiency and creativity, ultimately building better software, faster.

 Great job! You've now created a reliable "assembly line" using a sequential agent, where each step runs in a predictable order.

**This is perfect for tasks that build on each other, but it's slow if the tasks are independent.**  
Next, we'll look at how to run multiple agents at the same time to speed up your workflow.

---

## Section 4: Parallel Workflows - Independent Researchers

### The Problem: The Bottleneck

The previous sequential agent is great, but it's an assembly line. Each step must wait for the previous one to finish. What if you have several tasks that are **not dependent** on each other? For example, researching three *different* topics. Running them in sequence would be slow and inefficient, creating a bottleneck where each task waits unnecessarily.

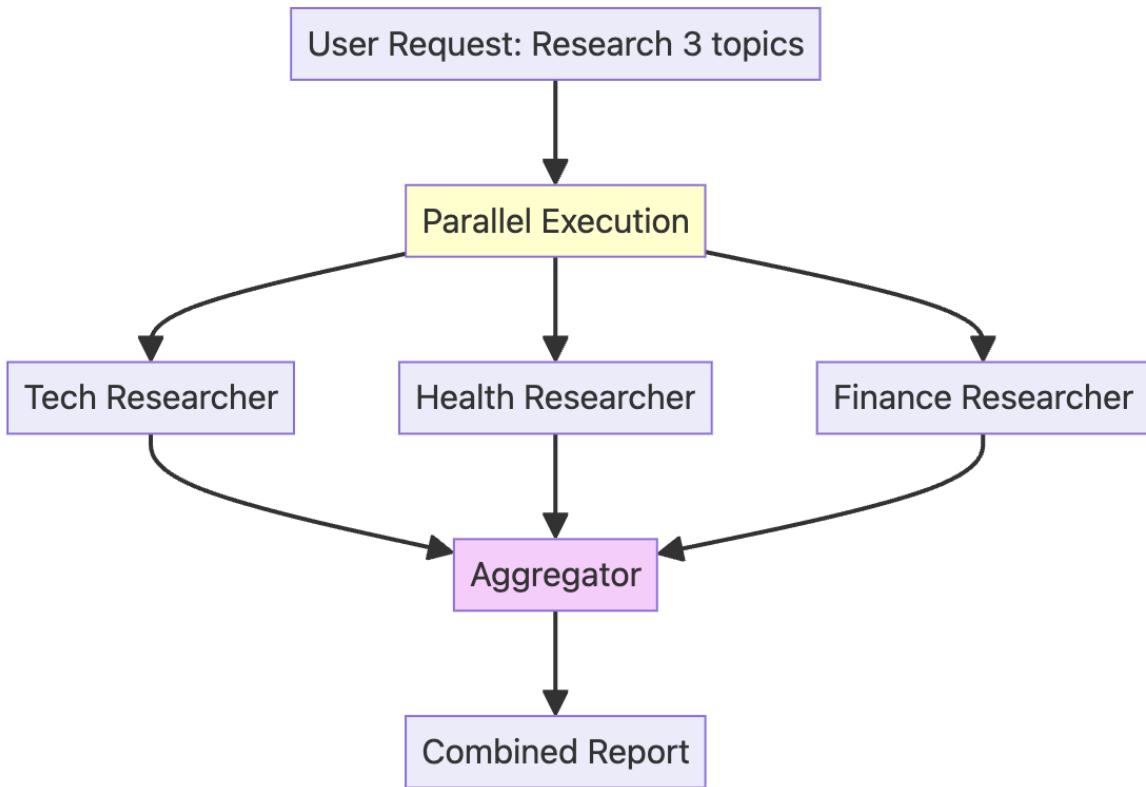
### The Solution: Concurrent Execution

When you have independent tasks, you can run them all at the same time using a `ParallelAgent`. This agent executes all of its sub-agents concurrently, dramatically speeding up the workflow. Once all parallel tasks are complete, you can then pass their combined results to a final 'aggregator' step.

**Use Parallel when:** Tasks are independent, speed matters, and you can execute concurrently.

To learn more, check out the documentation related to [parallel agents in ADK](#).

### Architecture: Multi-Topic Research



#### 4.1 Example: Parallel Multi-Topic Research

Let's build a system with four agents:

1. **Tech Researcher** - Researches AI/ML news and trends
2. **Health Researcher** - Researches recent medical news and trends
3. **Finance Researcher** - Researches finance and fintech news and trends
4. **Aggregator Agent** - Combines all research findings into a single summary

```

# Tech Researcher: Focuses on AI and ML trends.
tech_researcher = Agent(
    name="TechResearcher",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Research the latest AI/ML trends. Include 3 key
developments,
  
```

```
the main companies involved, and the potential impact. Keep the report  
very concise (100 words). """,  
    tools=[google_search],  
    output_key="tech_research", # The result of this agent will be stored  
in the session state with this key.  
)  
  
print("✅ tech_researcher created.")
```

✅ tech\_researcher created.

```
# Health Researcher: Focuses on medical breakthroughs.  
health_researcher = Agent(  
    name="HealthResearcher",  
    model=Gemini(  
        model="gemini-2.5-flash-lite",  
        retry_options=retry_config  
,  
        instruction="""Research recent medical breakthroughs. Include 3  
significant advances,  
their practical applications, and estimated timelines. Keep the report  
concise (100 words).""",  
        tools=[google_search],  
        output_key="health_research", # The result will be stored with this  
key.  
)  
  
print("✅ health_researcher created.")
```

✅ health\_researcher created.

```
# Finance Researcher: Focuses on fintech trends.  
finance_researcher = Agent(  
    name="FinanceResearcher",
```

```
model=Gemini(
    model="gemini-2.5-flash-lite",
    retry_options=retry_config
),
instruction="""Research current fintech trends. Include 3 key trends,
their market implications, and the future outlook. Keep the report concise
(100 words).""",
tools=[google_search],
output_key="finance_research", # The result will be stored with this
key.
)

print("✅ finance_researcher created.")
```

```
✅ finance_researcher created.
```

```
# The AggregatorAgent runs *after* the parallel step to synthesize the
results.

aggregator_agent = Agent(
    name="AggregatorAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
),
    # It uses placeholders to inject the outputs from the parallel agents,
    which are now in the session state.
    instruction="""Combine these three research findings into a single
executive summary:

**Technology Trends:**\n{tech_research}\n\n**Health Breakthroughs:**\n{health_research}\n\n**Finance Innovations:**\n{finance_research}"""
)
```

```
Your summary should highlight common themes, surprising connections,  
and the most important key takeaways from all three reports. The final  
summary should be around 200 words."",  
    output_key="executive_summary", # This will be the final output of the  
entire system.  
)  
  
print("✅ aggregator_agent created.")
```

✅ aggregator\_agent created.

👉 Then we bring the agents together under a parallel agent, which is itself nested inside of a sequential agent.

This design ensures that the research agents run first in parallel, then once all of their research is complete, the aggregator agent brings together all of the research findings into a single report:

```
# The ParallelAgent runs all its sub-agents simultaneously.  
parallel_research_team = ParallelAgent(  
    name="ParallelResearchTeam",  
    sub_agents=[tech_researcher, health_researcher, finance_researcher],  
)  
  
# This SequentialAgent defines the high-level workflow: run the parallel  
team first, then run the aggregator.  
root_agent = SequentialAgent(  
    name="ResearchSystem",  
    sub_agents=[parallel_research_team, aggregator_agent],  
)  
  
print("✅ Parallel and Sequential Agents created.")
```

✅ Parallel and Sequential Agents created.

Let's run the agent and give it a prompt to research the given topics:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Run the daily executive briefing on Tech, Health, and Finance"
)
```

```
### Created new session: debug_session_id
```

User > Run the daily executive briefing on Tech, Health, and Finance  
FinanceResearcher > Here's your daily executive briefing for Monday, November 10, 2025:

**Technology:** Tech giants are exploring space for data centers due to escalating AI energy demands, with startups launching GPU satellites for orbital computing tests. Apple is reportedly paying Google \$1 billion annually for AI model integration, aiming to develop its own model by late 2026. Harmonic Inc. announced its CFO will host investor meetings at the Needham Tech Week Conference.

**Health:** The FDA is initiating the removal of broad "black box" warnings from hormone replacement therapy (HRT) products for menopause, aiming to restore gold-standard science to women's health. The Pan American Health Organization reported the reestablishment of endemic measles transmission in the Americas, a setback that is reversible with political commitment and sustained vaccination.

**Finance:** Key fintech trends include the rise of embedded finance, where financial services are integrated into non-financial platforms, expanding accessibility and customer convenience. Decentralized Finance (DeFi) continues to grow, offering alternative financial systems and attracting new types of investors. Open banking initiatives are also gaining momentum, fostering innovation and competition by enabling third-party developers to build applications and services around financial institutions. The market implications of these trends point towards a more integrated, accessible, and innovative financial landscape. The future outlook suggests a continued blurring of lines between technology and

finance, with increased personalization and data-driven financial solutions.

TechResearcher > \*\*AI/ML Trends: Key Developments, Companies, and Impact\*\*

**\*\*1. Generative AI Expansion:\*\*** Generative AI is rapidly advancing beyond text to create diverse content like images, video, and music. Companies like Google (Imagen, Muse) and Stability AI (Stable Diffusion) are leading this charge. The impact includes revolutionizing content creation in marketing and art, while also raising ethical concerns about bias and misinformation.

**\*\*2. Rise of Agentic and Explainable AI:\*\*** AI agents are gaining autonomy in complex tasks, impacting logistics and customer support.

Simultaneously, there's a growing demand for Explainable AI (XAI) to ensure transparency and trust, particularly in sensitive sectors like healthcare and finance. Major tech firms like Google and Microsoft are investing heavily in these areas, aiming for responsible AI development.

**\*\*3. AI-Driven Automation and Personalization:\*\*** AI is increasingly used for end-to-end automation, from manufacturing processes to customer service. Companies like Amazon leverage AI for inventory management and personalized recommendations. This trend enhances efficiency and reduces costs but also raises concerns about job displacement and the need for workforce adaptation and upskilling.

HealthResearcher > \*\*Health:\*\* mRNA vaccine technology, initially prominent for COVID-19, is now being explored for other diseases like shingles and cancer, offering faster development and improved efficacy. Lab-grown blood has successfully undergone initial human trials, presenting a potential revolution for treating blood disorders. CRISPR gene-editing offers precise DNA alteration to correct genetic diseases, with early applications in inherited disorders. These advances are largely available or in advanced clinical trials, with widespread impact expected within the next 1-5 years.

**\*\*Technology:\*\*** AI and machine learning continue to advance, powering innovations from virtual assistants to predictive analytics and personalized medicine. 5G networks are expanding, enabling faster connectivity crucial for autonomous vehicles and smart cities. Extended Reality (XR), encompassing VR and AR, is transforming gaming, education, and remote collaboration, with market growth projected significantly by

2025. These technologies are already integrated, with ongoing development and expansion expected over the next 1-3 years.

**\*\*Finance:\*\*** AI and machine learning are central to FinTech, enhancing risk decisioning and user experience, with market growth projected to double by 2029. Embedded finance is streamlining transactions by integrating financial services into non-financial platforms, with sector growth to reach \$138 billion by 2026. Blockchain and DeFi are maturing, offering secure and transparent financial solutions, with new use cases emerging in various sectors. These financial innovations are currently being implemented and are expected to see significant growth and adoption within the next 1-3 years.

AggregatorAgent > ## Executive Summary: AI, Health Advancements, and Financial Innovation Converge

**\*\*Key Takeaways:\*\*** Artificial Intelligence (AI) is the dominant cross-sectoral driver, powering advancements in content creation, automation, personalization, and increasingly autonomous agents. This is profoundly impacting technology, finance, and healthcare. In health, mRNA technology, lab-grown blood, and CRISPR gene editing promise revolutionary treatments for diverse diseases, with widespread impact expected within 1-5 years. Concurrently, financial services are being transformed by embedded finance and Decentralized Finance (DeFi), fostering greater accessibility and integration with technology, with significant growth projected in the next 1-3 years.

**\*\*Surprising Connections:\*\*** The escalating energy demands of AI are prompting tech giants to explore space-based data centers, highlighting the symbiotic, and sometimes extreme, relationship between AI development and infrastructure. Apple's substantial investment in Google's AI models underscores the immense value and competitive landscape of AI integration.

**\*\*Most Important Themes:\*\***

- \*   **\*\*AI as an Enabler:\*\*** AI is a foundational technology fueling innovation across all sectors, from generative content to personalized medicine and streamlined financial transactions.
- \*   **\*\*Transformative Health Solutions:\*\*** Breakthroughs in mRNA, regenerative medicine, and gene editing are poised to reshape healthcare within the next few years.

- \* \*\*Integrated Financial Ecosystem:\*\* Embedded finance and DeFi are blurring the lines between traditional finance and technology, creating more accessible and user-centric financial solutions.
- \* \*\*Ethical and Infrastructural Challenges:\*\* The rapid AI expansion necessitates addressing ethical concerns like bias and misinformation, alongside significant infrastructural demands, including energy consumption.

🎉 Great! You've seen how parallel agents can dramatically speed up workflows by running independent tasks concurrently.

So far, all our workflows run from start to finish and then stop. **But what if you need to review and improve an output multiple times?** Next, we'll build a workflow that can loop and refine its own work.

---

## ⌚ Section 5: Loop Workflows - The Refinement Cycle

### The Problem: One-Shot Quality

All the workflows we've seen so far run from start to finish. The `SequentialAgent` and `ParallelAgent` produce their final output and then stop. This 'one-shot' approach isn't good for tasks that require refinement and quality control. What if the first draft of our story is bad? We have no way to review it and ask for a rewrite.

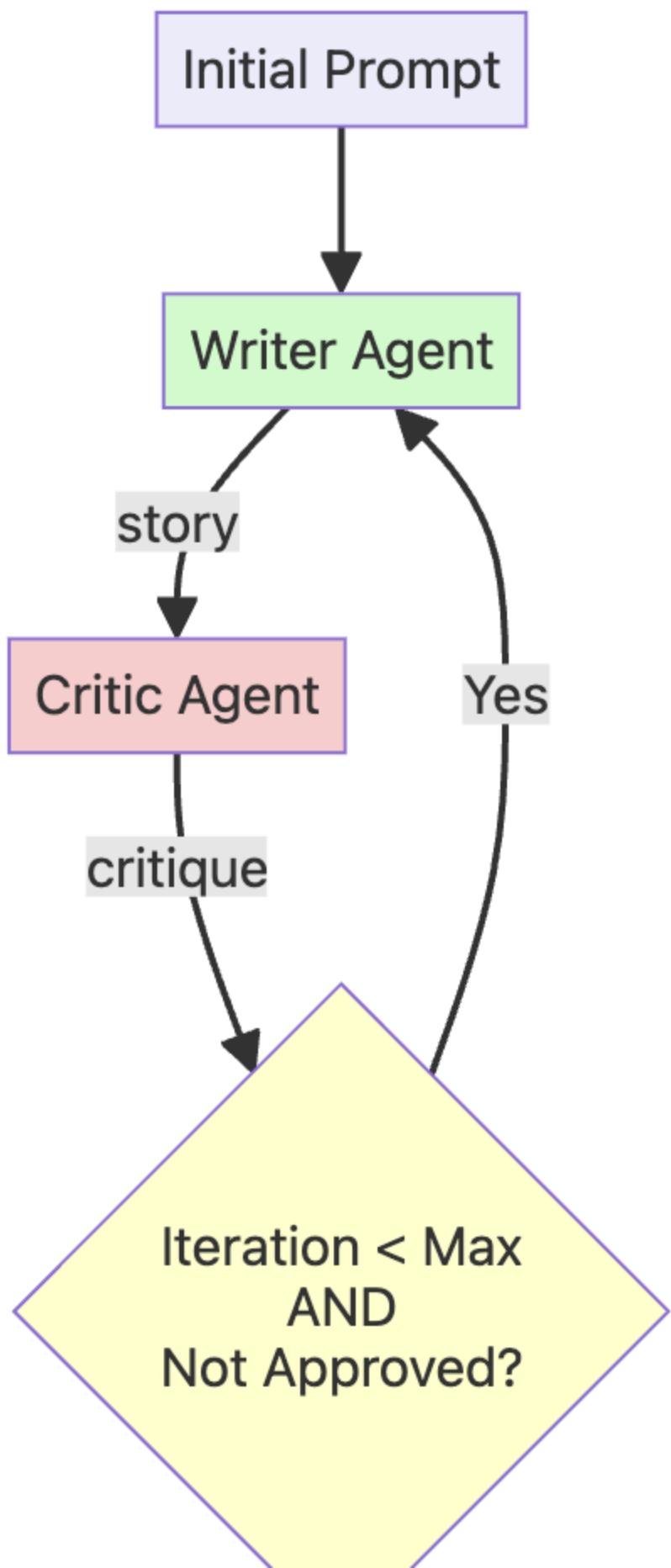
### The Solution: Iterative Refinement

When a task needs to be improved through cycles of feedback and revision, you can use a `LoopAgent`. A `LoopAgent` runs a set of sub-agents repeatedly *until a specific condition is met or a maximum number of iterations is reached*. This creates a refinement cycle, allowing the agent system to improve its own work over and over.

**Use Loop when:** Iterative improvement is needed, quality refinement matters, or you need repeated cycles.

To learn more, check out the documentation related to [loop agents in ADK](#).

#### **Architecture: Story Writing & Critique Loop**



## 5.1 Example: Iterative Story Refinement

Let's build a system with two agents:

1. **Writer Agent** - Writes a draft of a short story
2. **Critic Agent** - Reviews and critiques the short story to suggest improvements

```
# This agent runs ONCE at the beginning to create the first draft.
initial_writer_agent = Agent(
    name="InitialWriterAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Based on the user's prompt, write the first draft of a
short story (around 100-150 words).
Output only the story text, with no introduction or explanation.""",
    output_key="current_story", # Stores the first draft in the state.
)
print("✓ initial_writer_agent created.")
```

✓ initial\_writer\_agent created.

```
# This agent's only job is to provide feedback or the approval signal. It
has no tools.
critic_agent = Agent(
    name="CriticAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a constructive story critic. Review the story
provided below.
Story: {current_story}
```

Evaluate the story's plot, characters, and pacing.

```
- If the story is well-written and complete, you MUST respond with the
exact phrase: "APPROVED"
- Otherwise, provide 2-3 specific, actionable suggestions for
improvement."",
    output_key="critique", # Stores the feedback in the state.
)

print("✓ critic_agent created.")
```

```
✓ critic_agent created.
```

Now, we need a way for the loop to actually stop based on the critic's feedback. The LoopAgent itself doesn't automatically know that "APPROVED" means "stop."

We need an agent to give it an explicit signal to terminate the loop.

We do this in two parts:

1. A simple Python function that the LoopAgent understands as an "exit" signal.
2. An agent that can call that function when the right condition is met.

First, you'll define the `exit_loop` function:

```
# This is the function that the RefinerAgent will call to exit the loop.
def exit_loop():
    """Call this function ONLY when the critique is 'APPROVED', indicating
    the story is finished and no more changes are needed."""
    return {"status": "approved", "message": "Story approved. Exiting
refinement loop."}

print("✓ exit_loop function created.")
```

```
✓ exit_loop function created.
```

✓ exit\_loop function created.

To let an agent call this Python function, we wrap it in a FunctionTool. Then, we create a RefinerAgent that has this tool.

👉 **Notice its instructions:** this agent is the "brain" of the loop. It reads the {critique} from the CriticAgent and decides whether to (1) call the exit\_loop tool or (2) rewrite the story.

```
# This agent refines the story based on critique OR calls the exit_loop
# function.
refiner_agent = Agent(
    name="RefinerAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a story refiner. You have a story draft and
critique.

Story Draft: {current_story}
Critique: {critique}

Your task is to analyze the critique.
- IF the critique is EXACTLY "APPROVED", you MUST call the `exit_loop` function and nothing else.
- OTHERWISE, rewrite the story draft to fully incorporate the feedback from the critique."""
    output_key="current_story", # It overwrites the story with the new,
    refined version.
    tools=[
        FunctionTool(exit_loop)
    ], # The tool is now correctly initialized with the function
    reference.
)

print("✓ refiner_agent created.")
```

 refiner\_agent created.

Then we bring the agents together under a loop agent, which is itself nested inside of a sequential agent.

This design ensures that the system first produces an initial story draft, then the refinement loop runs up to the specified number of max\_iterations:

```
# The LoopAgent contains the agents that will run repeatedly: Critic ->
Refiner.
story_refinement_loop = LoopAgent(
    name="StoryRefinementLoop",
    sub_agents=[critic_agent, refiner_agent],
    max_iterations=2, # Prevents infinite loops
)

# The root agent is a SequentialAgent that defines the overall workflow:
Initial Write -> Refinement Loop.
root_agent = SequentialAgent(
    name="StoryPipeline",
    sub_agents=[initial_writer_agent, story_refinement_loop],
)

print("✅ Loop and Sequential Agents created.")
```

 Loop and Sequential Agents created.

Let's run the agent and give it a topic to write a short story about:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug()
```

"Write a short story about a lighthouse keeper who discovers a mysterious, glowing map"

)

```
### Created new session: debug_session_id
```

User > Write a short story about a lighthouse keeper who discovers a mysterious, glowing map

InitialWriterAgent > Elias polished the Fresnel lens until it gleamed, his only companion the ceaseless roar of the waves. For thirty years, the lighthouse had been his world. Tonight, however, a faint shimmer beneath a loose floorboard caught his eye. Prying it open, he found a rolled parchment, brittle with age. As he unfurled it, an ethereal blue light pulsed from intricate markings that shifted and reformed. It wasn't ink, but some phosphorescent substance. The lines traced no known coast, no familiar sea lanes. Instead, they depicted a labyrinth of submerged caverns and celestial alignments, culminating in a single, radiant star marked "X." Elias, a man of routine, felt a tremor of an unknown adventure stir within him.

CriticAgent > The story is promising, but could benefit from a bit more depth. Here are a few suggestions:

1. **\*\*Show, Don't Just Tell Elias's Routine and Personality:\*\*** While it's stated Elias is a man of routine and the lighthouse has been his world for thirty years, we don't *\*see\** much of this routine or feel the weight of his isolation. Consider adding a brief scene or detail that illustrates his monotonous life before the discovery, making the "tremor of an unknown adventure" feel more impactful by contrast. For example, describe the ritual of his polishing, the specific sounds he hears that are now

familiar, or a brief thought about the unchanging nature of his existence.

2. **\*\*Deepen the Mystery of the Map:\*\*** The description of the map is intriguing, but could be even more evocative. Instead of stating it's "not ink, but some phosphorescent substance," describe *\*how\** it glows. Does it flicker? Does it have a scent? Are the shifting markings subtle or dynamic? Similarly, the "celestial alignments" could be made more specific – are they constellations, planets, or something more abstract? This would enhance the sense of ancient, otherworldly knowledge.

3. \*\*Foreshadow the "Unknown Adventure":\*\* The story ends with Elias feeling a "tremor of an unknown adventure." To make this more compelling, you could hint at what this adventure might entail or the immediate consequence of his discovery. Does the light from the map cast strange shadows? Does the roar of the waves seem to change? A subtle hint of what lies beyond the lighthouse could propel the reader forward, making them eager to know what Elias will do next.

RefinerAgent > Elias meticulously polished the great Fresnel lens, the rhythmic sweep of his cloth a familiar ritual against the ceaseless roar of the waves. Thirty years he'd kept this solitary vigil, the lighthouse his unyielding world. Each night, the same: the lamp lit, the logbook filled, the lonely descent to his quarters. The salt-laced wind whistling through the tower's stones was his only confidante, the crash of surf against the rocks his only music.

But tonight, the monotony fractured. A faint shimmer, aberrant and out of place, caught his eye beneath a loose floorboard near his cot. His gnarled fingers, accustomed to the predictable weight of tools and lamp oil, pried at the aged wood. It groaned in protest, revealing a hollow space. Inside, a rolled parchment, brittle and yellowed, lay waiting.

As Elias unfurled it, an ethereal blue light bloomed, pulsing with a soft, internal rhythm. It wasn't ink that formed the intricate markings, but some phosphorescent substance that seemed to writhe and reform as he watched, like captured starlight. The lines traced no known coast, no charted sea lanes. Instead, they depicted a labyrinth of submerged caverns, their passages winding like the intestines of some colossal sea beast. Strange celestial alignments, not of constellations he recognized but of alien planetary bodies, hung suspended above the treacherous geography. The map culminated in a single, radiant star, marked with a stark, definitive "X."

The phosphorescence intensified, casting dancing blue shadows across Elias's weathered face. The steady roar of the waves outside seemed to deepen, to take on a resonant, almost expectant tone. The parchment felt cool to the touch, yet it radiated a subtle energy that made the hairs on his arm stand on end. Elias, a man whose life had been defined by the unwavering beam of his lighthouse and the predictable turn of the tides, felt a tremor, profound and unfamiliar, stir within him. It was the tremor of an unknown adventure, whispering from the depths of the sea and the far

reaches of the cosmos, promising a world far stranger than the one he had always known.

CriticAgent > APPROVED

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

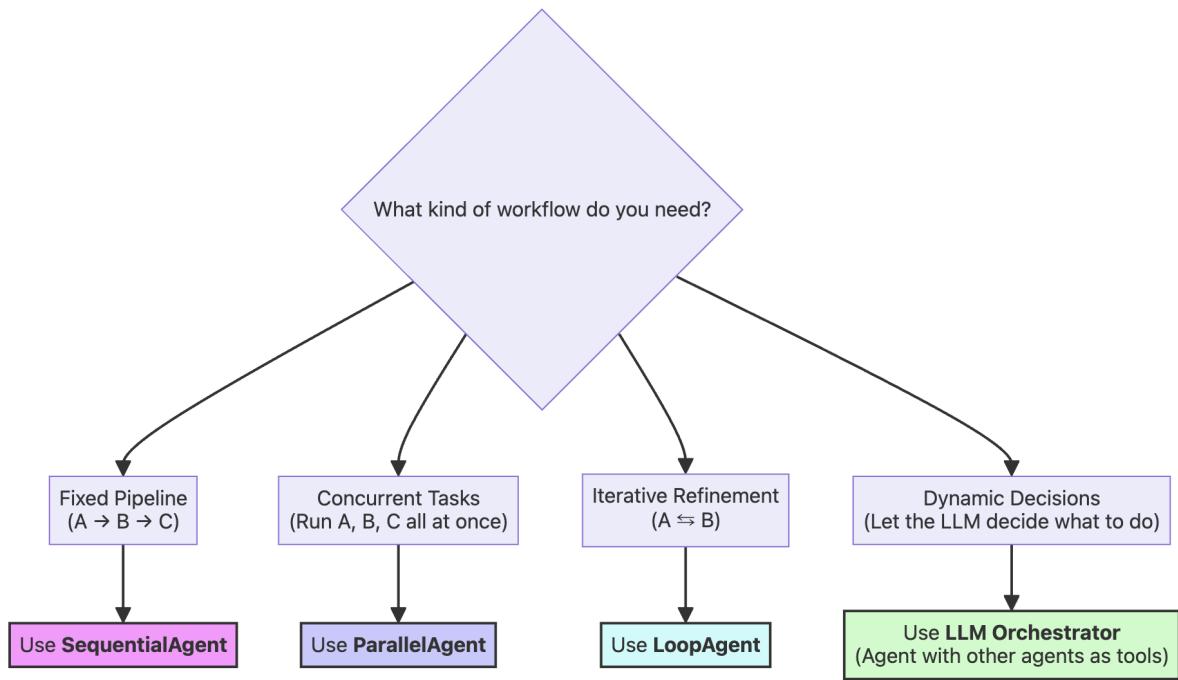
You've now implemented a loop agent, creating a sophisticated system that can iteratively review and improve its own output. This is a key pattern for ensuring high-quality results.

You now have a complete toolkit of workflow patterns. Let's put it all together and review how to choose the right one for your use case.

---

## Section 6: Summary - Choosing the Right Pattern

Decision Tree: Which Workflow Pattern?



## Quick Reference Table

Pattern	When to Use	Example	Key Feature
LLM-based (sub_agents)	Dynamic orchestration needed	Research + Summarize	LLM decides what to call
Sequential	Order matters, linear pipeline	Outline → Write → Edit	Deterministic order
Parallel	Independent tasks, speed matters	Multi-topic research	Concurrent execution

Loop	Iterative improvement needed	Writer + Critic refinement	Repeated cycles
------	------------------------------	----------------------------	-----------------

---

## Congratulations! You're Now an Agent Orchestrator

In this notebook, you made the leap from a single agent to a **multi-agent system**.

You saw **why** a team of specialists is easier to build and debug than one "do-it-all" agent. Most importantly, you learned how to be the **director** of that team.

You used SequentialAgent, ParallelAgent, and LoopAgent to create deterministic workflows, and you even used an LLM as a 'manager' to make dynamic decisions. You also mastered the "plumbing" by using output\_key to pass state between agents and make them collaborative.

### Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

### Learn More

Refer to the following documentation to learn more:

- [Agents in ADK](#)
- [Sequential Agents in ADK](#)
- [Parallel Agents in ADK](#)
- [Loop Agents in ADK](#)
- [Custom Agents in ADK](#)

## Next Steps

Ready for the next challenge? Stay tuned for Day 2 notebooks where we'll learn how to create **Custom Functions**, use **MCP Tools** and manage **Long-Running operations!**

---

Authors
<a href="#"><u>Kristopher Overholt</u></a>

*Copyright 2025 Google LLC.*

```
# @title 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
#
# https://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.
```

## Agent Tools

Welcome to Day-2 of the Kaggle 5-day Agents course!

In Day-1, you learned how to create agents with built-in tools like Google Search. You also learned how to orchestrate multi-agent systems. Now let's unlock the full power of agent tools by building custom logic, delegating to specialist agents, and handling real-world complexities.

## Why do Agents need Tools?

### The Problem

Without tools, the agent's knowledge is frozen in time — it can't access today's news or your company's inventory. It has no connection to the outside world, so the agent can't take actions for you.

**The Solution:** Tools are what transform your isolated LLM into a capable agent that can actually help you get things done.

In this notebook, you'll:

-  Turn your Python functions into Agent tools
-  Build an Agent and use it **as a tool** in another agent
-  **Build your first multi-tool agent**
-  Explore the different tool types in ADK

### !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

 Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

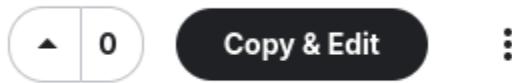
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy` and `Edit` button in the top-right corner.

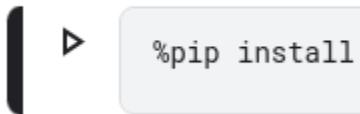


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

## Section 1: Setup

Before we go into today's concepts, follow the steps below to set up the environment.

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires an API key.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

#### 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

#### 3. Authenticate in the notebook

Run the cell below to access the GOOGLE\_API\_KEY you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✓ Setup and authentication complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

```
✓ Setup and authentication complete.
```

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from google.genai import types

from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import InMemoryRunner
from google.adk.sessions import InMemorySessionService
from google.adk.tools import google_search, AgentTool, ToolContext
from google.adk.code_executors import BuiltInCodeExecutor

print("✓ ADK components imported successfully.")
```

 ADK components imported successfully.

## 1.4: Helper functions

Helper function that prints the generated Python code and results from the code execution tool:

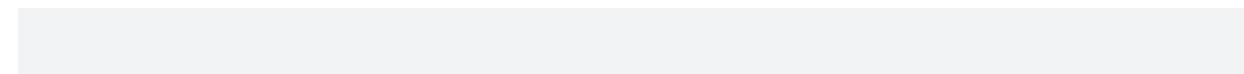
```
def show_python_code_and_result(response):
    for i in range(len(response)):
        # Check if the response contains a valid function call result from
        # the code executor
        if (
            (response[i].content.parts)
            and (response[i].content.parts[0])
            and (response[i].content.parts[0].function_response)
            and (response[i].content.parts[0].function_response.response)
        ):
            response_code =
                response[i].content.parts[0].function_response.response
            if "result" in response_code and response_code["result"] != "None":
                if "tool_code" in response_code["result"]:
                    print(
                        "Generated Python Code >> ",
                        response_code["result"].replace("tool_code", ""),
                    )
                else:
                    print("Generated Python Response >> ",
                        response_code["result"])
            print("✓ Helper functions defined.")
```

 Helper functions defined.

## 1.5: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config = types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```



## Section 2: What are Custom Tools?

**Custom Tools** are tools you build yourself using your own code and business logic. Unlike built-in tools that come ready-made with ADK, custom tools give you complete control over functionality.

### When to use Custom Tools?

Built-in tools like Google Search are powerful, but **every business has unique requirements** that generic tools can't handle. Custom tools let you implement your specific business logic, connect to your systems, and solve domain-specific problems. ADK provides multiple custom tool types to handle these scenarios.

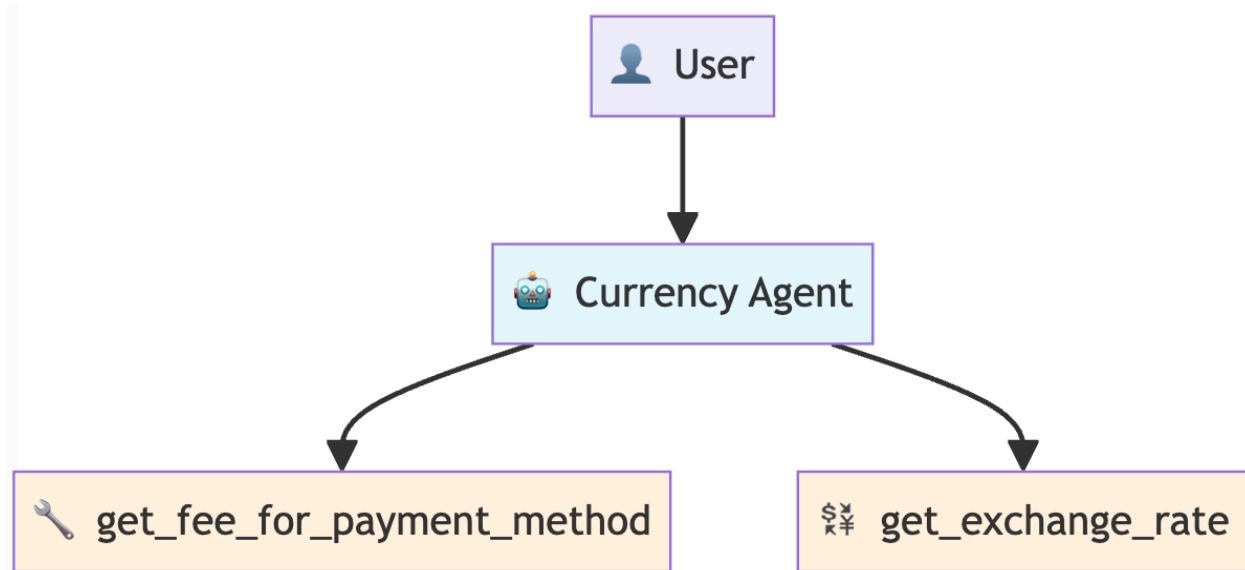
## 2.1: Building Custom Function Tools

### Example: Currency Converter Agent

This agent can convert currency from one denomination to another and calculates the fees to do the conversion. The agent has two custom tools and follows the workflow:

1. **Fee Lookup Tool** - Finds transaction fees for the conversion (mock)
2. **Exchange Rate Tool** - Gets currency conversion rates (mock)

3. **Calculation Step** - Calculates the total conversion cost including the fees



## 💡 2.2: How to define a Tool?

Any Python function can become an **agent tool** by following these simple guidelines:

1. Create a Python function
2. Follow the best practices listed below
3. Add your function to the agent's `tools=[ ]` list and ADK handles the rest automatically.

## 🏆 ADK Best Practices in Action

Notice how our tools follow ADK best practices:

- 1. Dictionary Returns:** Tools return `{"status": "success", "data": ...}` or `{"status": "error", "error_message": ...}`
- 2. Clear Docstrings:** LLMs use docstrings to understand when and how to use tools
- 3. Type Hints:** Enable ADK to generate proper schemas (`str`, `dict`, etc.)
- 4. Error Handling:** Structured error responses help LLMs handle failures gracefully

These patterns make your tools reliable and easy for LLMs to use correctly.

👉 Let's see this in action with our first tool:

```
# Pay attention to the docstring, type hints, and return value.
def get_fee_for_payment_method(method: str) -> dict:
    """Looks up the transaction fee percentage for a given payment method.

    This tool simulates looking up a company's internal fee structure based
    on
    the name of the payment method provided by the user.

    Args:
        method: The name of the payment method. It should be descriptive,
            e.g., "platinum credit card" or "bank transfer".

    Returns:
        Dictionary with status and fee information.
        Success: {"status": "success", "fee_percentage": 0.02}
        Error: {"status": "error", "error_message": "Payment method not
    found"}
    """

    # This simulates looking up a company's internal fee structure.
    fee_database = {
        "platinum credit card": 0.02,  # 2%
        "gold debit card": 0.035,  # 3.5%
        "bank transfer": 0.01,  # 1%
    }

    fee = fee_database.get(method.lower())
    if fee is not None:
        return {"status": "success", "fee_percentage": fee}
    else:
        return {
            "status": "error",
            "error_message": f"Payment method '{method}' not found",
        }

print("✅ Fee lookup function created")
print(f"🟡 Test: {get_fee_for_payment_method('platinum credit card')}")
```

```
✓ Fee lookup function created
█ Test: {'status': 'success', 'fee_percentage': 0.02}
```

Let's follow the same best practices to define our second tool `get_exchange_rate`.

```
def get_exchange_rate(base_currency: str, target_currency: str) -> dict:
    """Looks up and returns the exchange rate between two currencies.

    Args:
        base_currency: The ISO 4217 currency code of the currency you
                       are converting from (e.g., "USD").
        target_currency: The ISO 4217 currency code of the currency you
                         are converting to (e.g., "EUR").

    Returns:
        Dictionary with status and rate information.
        Success: {"status": "success", "rate": 0.93}
        Error: {"status": "error", "error_message": "Unsupported currency
pair"}
    """
    # Static data simulating a live exchange rate API
    # In production, this would call something like:
    requests.get("api.exchangerates.com")
    rate_database = {
        "usd": {
            "eur": 0.93, # Euro
            "jpy": 157.50, # Japanese Yen
            "inr": 83.58, # Indian Rupee
        }
    }

    # Input validation and processing
    base = base_currency.lower()
    target = target_currency.lower()
```

```

# Return structured result with status
rate = rate_database.get(base, {}).get(target)
if rate is not None:
    return {"status": "success", "rate": rate}
else:
    return {
        "status": "error",
        "error_message": f"Unsupported currency pair: {base_currency}/{target_currency}",
    }

print("✅ Exchange rate function created")
print(f"$€ Test: {get_exchange_rate('USD', 'EUR')}")

```

✅ Exchange rate function created  
\$€ Test: {'status': 'success', 'rate': 0.93}

Now let's create our currency agent. Pay attention to how the agent's instructions reference the tools:

#### Key Points:

- The tools=[ ] list tells the agent which functions it can use
- Instructions reference tools by their exact function names (e.g., get\_fee\_for\_payment\_method())
- The agent uses these names to decide when and how to call each tool

```

# Currency agent with custom function tools
currency_agent = LlmAgent(
    name="currency_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    instruction="""You are a smart currency conversion assistant.

```

For currency conversion requests:

1. Use `get\_fee\_for\_payment\_method()` to find transaction fees
2. Use `get\_exchange\_rate()` to get currency conversion rates
3. Check the "status" field in each tool's response for errors
4. Calculate the final amount after fees based on the output from `get\_fee\_for\_payment\_method` and `get\_exchange\_rate` methods and provide a clear breakdown.

5. First, state the final converted amount.

Then, explain how you got that result by showing the intermediate amounts. Your explanation must include: the fee percentage and its value in the original currency, the amount remaining after the fee, and the exchange rate used for the final conversion.

If any tool returns status "error", explain the issue to the user clearly.

```
"""
tools=[get_fee_for_payment_method, get_exchange_rate],
)

print("✅ Currency agent created with custom function tools")
print("🔧 Available tools:")
print("  • get_fee_for_payment_method - Looks up company fee structure")
print("  • get_exchange_rate - Gets current exchange rates")
```

 Currency agent created with custom function tools  
 Available tools:  
 • get\_fee\_for\_payment\_method - Looks up company fee structure  
 • get\_exchange\_rate - Gets current exchange rates

```
# Test the currency agent
currency_runner = InMemoryRunner(agent=currency_agent)
_ = await currency_runner.run_debug(
    "I want to convert 500 US Dollars to Euros using my Platinum Credit
Card. How much will I receive?"
)
```

```
### Created new session: debug_session_id

User > I want to convert 500 US Dollars to Euros using my Platinum Credit Card. How much will I receive?
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call', 'function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call', 'function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
```

```
currency_agent > You will receive 455.70 Euros.
```

This is based on the following:

The transaction fee for using a platinum credit card is 2%, which amounts to 10 USD.

After deducting the fee, the remaining amount is 490 USD.

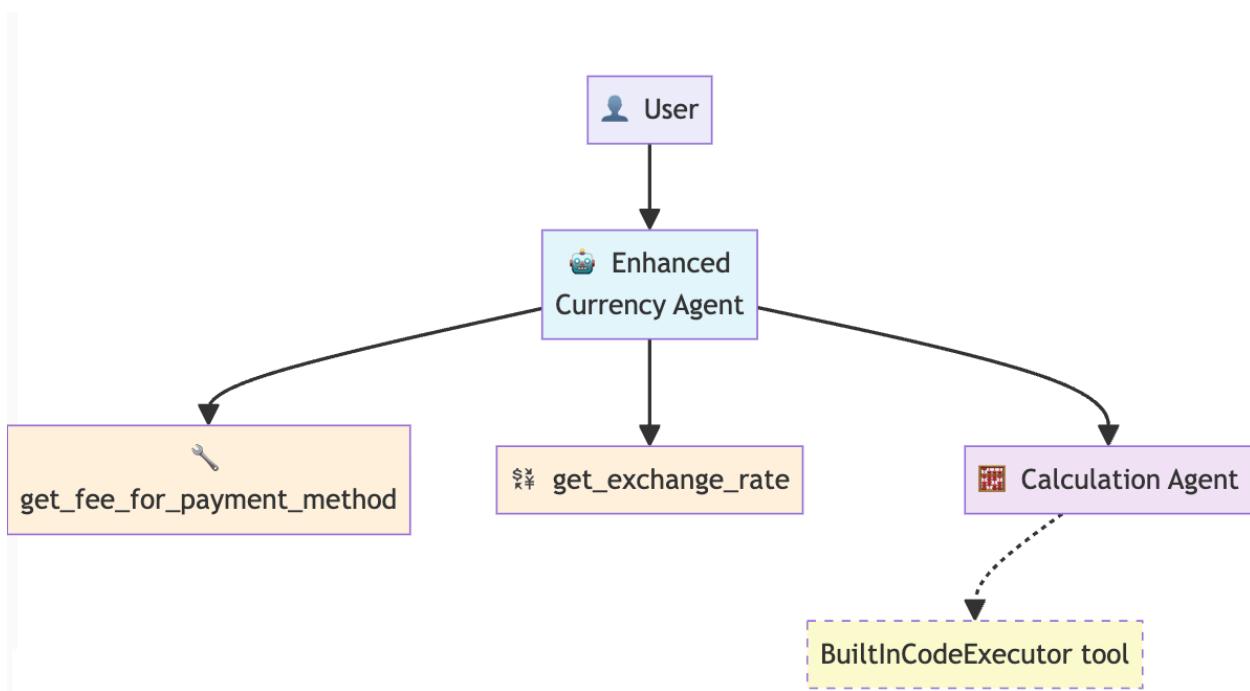
The exchange rate from USD to EUR is 0.93, so 490 USD is converted to 455.70 EUR.

**Excellent!** Our agent now uses custom business logic with structured responses.

## Section 3: Improving Agent Reliability with Code

The agent's instruction says "*calculate the final amount after fees*" but LLMs aren't always reliable at math. They might make calculation errors or use inconsistent formulas.

 **Solution:** Let's ask our agent to generate a Python code to do the math, and run it to give us the final result! Code execution is much more reliable than having the LLM try to do math in its head!



### 3.1 Built-in Code Executor

ADK has a built-in Code Executor capable of running code in a sandbox. **Note:** This uses Gemini's Code Execution capability.

Let's create a calculation\_agent which takes in a Python code and uses the BuiltInCodeExecutor to run it.

```
calculation_agent = LlmAgent(
    name="CalculationAgent",
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    instruction="""You are a specialized calculator that ONLY responds
with Python code. You are forbidden from providing any text, explanations,
or conversational responses.
```

Your task is to take a request for a calculation and translate it into a single block of Python code that calculates the answer.

#### \*\*RULES:\*\*

1. Your output MUST be ONLY a Python code block.
2. Do NOT write any text before or after the code block.

3. The Python code MUST calculate the result.
4. The Python code MUST print the final result to stdout.
5. You are PROHIBITED from performing the calculation yourself. Your only job is to generate the code that will perform the calculation.

Failure to follow these rules will result in an error.

```
"""  
code_executor=BuiltInCodeExecutor(), # Use the built-in Code Executor  
Tool. This gives the agent code execution capabilities  
)
```

### 3.2: Update the Agent's instruction and toolset

We'll do two key actions:

1. **Update the currency\_agent's instructions to generate Python code**
  - Original: "Calculate the final amount after fees" (vague math instructions)
  - Enhanced: "Generate a Python code to calculate the final amount .. and use the calculation\_agent to run the code and compute final amount"
1. **Add the calculation\_agent to the toolset**

ADK lets you use any agent as a tool using AgentTool.

  - Add AgentTool(agent=calculation\_agent) to the tools list
  - The specialist agent appears as a callable tool to the root agent

Let's see this in action:

```
enhanced_currency_agent = LlmAgent(  
    name="enhanced_currency_agent",  
    model=Gemini(model="gemini-2.5-flash-lite",  
    retry_options=retry_config),  
    # Updated instruction  
    instruction="""You are a smart currency conversion assistant. You must  
    strictly follow these steps and use the available tools.
```

For any currency conversion request:

1. Get Transaction Fee: Use the get\_fee\_for\_payment\_method() tool to determine the transaction fee.

2. Get Exchange Rate: Use the `get_exchange_rate()` tool to get the currency conversion rate.
3. Error Check: After each tool call, you must check the "status" field in the response. If the status is "error", you must stop and clearly explain the issue to the user.
4. Calculate Final Amount (CRITICAL): You are strictly prohibited from performing any arithmetic calculations yourself. You must use the `calculation_agent` tool to generate Python code that calculates the final converted amount. This code will use the fee information from step 1 and the exchange rate from step 2.
5. Provide Detailed Breakdown: In your summary, you must:
  - \* State the final converted amount.
  - \* Explain how the result was calculated, including:
    - \* The fee percentage and the fee amount in the original currency.
    - \* The amount remaining after deducting the fee.
    - \* The exchange rate applied.

```
"""
tools=[
    get_fee_for_payment_method,
    get_exchange_rate,
    AgentTool(agent=calculation_agent), # Using another agent as a
tool!
],
)

print("✅ Enhanced currency agent created")
print("⌚ New capability: Delegates calculations to specialist agent")
print("🔧 Tool types used:")
print("  • Function Tools (fees, rates)")
print("  • Agent Tool (calculation specialist)")
```

- 
- ✓ Enhanced currency agent created
  - ⌚ New capability: Delegates calculations to specialist agent
  - 🔧 Tool types used:
    - Function Tools (fees, rates)
    - Agent Tool (calculation specialist)

```
# Define a runner
enhanced_runner = InMemoryRunner(agent=enhanced_currency_agent)

# Test the enhanced agent
response = await enhanced_runner.run_debug(
    "Convert 1,250 USD to INR using a Bank Transfer. Show me the precise
calculation."
)

### Created new session: debug_session_id

User > Convert 1,250 USD to INR using a Bank Transfer. Show me the precise
calculation.

WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
```

enhanced\_currency\_agent > The final converted amount is 103199.10 INR.

Here's the breakdown of the calculation:

The transaction fee for a Bank Transfer is 1.0%, which amounts to 12.50 USD.

After deducting the fee, 1237.50 USD remains.

This amount was converted to INR using an exchange rate of 1 USD = 83.58 INR.

Therefore, the final converted amount is 103199.10 INR.

**Excellent!** Notice what happened:

- When the Currency agent calls the CalculationAgent, it passes in the generated Python code
- The CalculationAgent in turn used the BuiltInCodeExecutor to run the code and gave us precise calculations instead of LLM guesswork!

Now you can inspect the parts of the response that either generated Python code or that contain the Python code results, using the helper function that was defined near the beginning of this notebook:

```
show_python_code_and_result(response)
```

```
Generated Python Code >>
original_amount_usd = 1250
fee_percentage = 1.0
exchange_rate = 83.58

# Calculate the fee amount
fee_amount_usd = original_amount_usd * (fee_percentage / 100)

# Calculate the amount after deducting the fee
amount_after_fee_usd = original_amount_usd - fee_amount_usd

# Calculate the final converted amount in INR
final_amount_inr = amount_after_fee_usd * exchange_rate

print(f"Original Amount (USD): {original_amount_usd}")
print(f"Fee Percentage: {fee_percentage}%")
print(f"Fee Amount (USD): {fee_amount_usd:.2f}")
print(f"Amount after deducting fee (USD): {amount_after_fee_usd:.2f}")
print(f"Exchange Rate (INR/USD): {exchange_rate}")
print(f"Final Converted Amount (INR): {final_amount_inr:.2f}")
```

### 3.3: Agent Tools vs Sub-Agents: What's the Difference?

This is a common question! Both involve using multiple agents, but they work very differently:

#### **Agent Tools (what we're using):**

- Agent A calls Agent B as a tool
- Agent B's response goes **back to Agent A**
- Agent A stays in control and continues the conversation
- **Use case:** Delegation for specific tasks (like calculations)

#### **Sub-Agents (different pattern):**

- Agent A transfers control **completely to Agent B**
- Agent B takes over and handles all future user input
- Agent A is out of the loop
- **Use case:** Handoff to specialists (like customer support tiers)

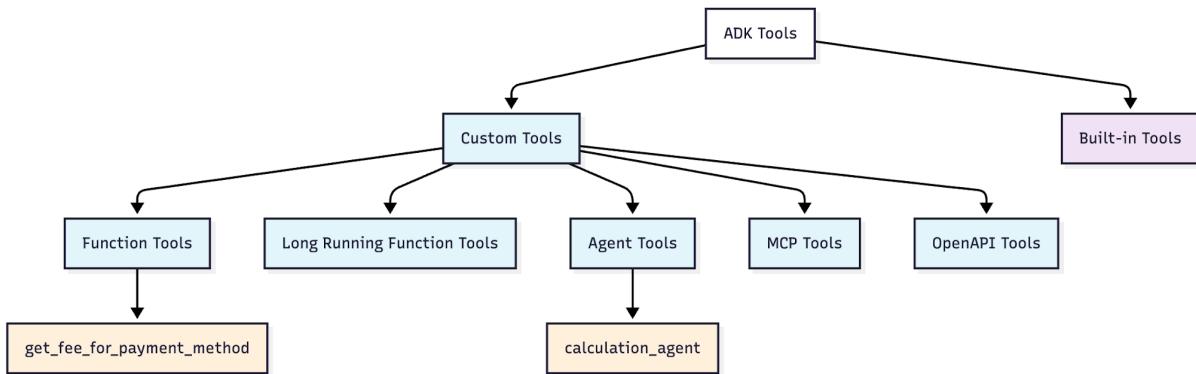
**In our currency example:** We want the currency agent to get calculation results and continue working with them, so we use **Agent Tools**, not sub-agents.

## Section 4: Complete Guide to ADK Tool Types

Now that you've seen tools in action, let's understand the complete ADK toolkit:

It's broadly divided into two categories: **Custom tools** and **Built-in tools**

### **1. Custom Tools**



**What:** Tools you build yourself for specific needs

**Advantage:** Complete control over functionality — you build exactly what your agent needs

### Function Tools ✓ (You've used these!)

- **What:** Python functions converted to agent tools
- **Examples:** `get_fee_for_payment_method`, `get_exchange_rate`
- **Advantage:** Turn any Python function into an agent tool instantly

### Long Running Function Tools

- **What:** Functions for operations that take significant time
- **Examples:** Human-in-the-loop approvals, file processing
- **Advantage:** Agents can start tasks and continue with other work while waiting

### Agent Tools ✓ (You've used these!)

- **What:** Other agents used as tools
- **Examples:** `AgentTool(agent=calculation_agent)`
- **Advantage:** Build specialist agents and reuse them across different systems

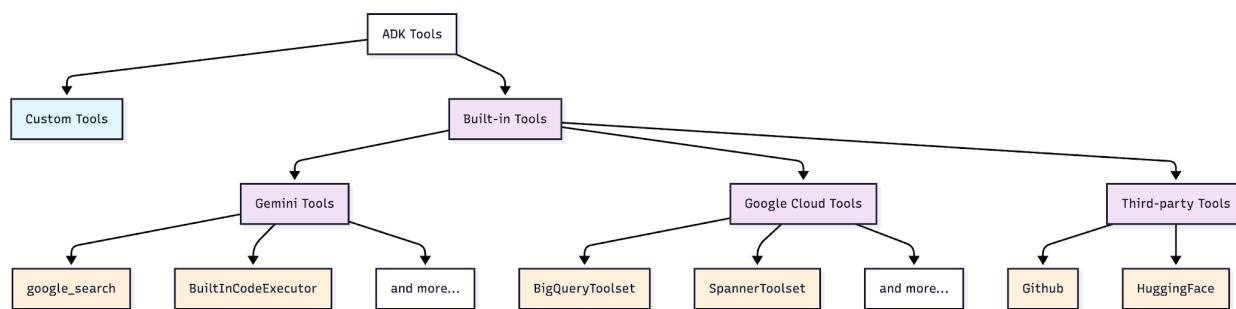
### MCP Tools

- **What:** Tools from Model Context Protocol servers
- **Examples:** Filesystem access, Google Maps, databases
- **Advantage:** Connect to any MCP-compatible service without custom integration

## OpenAPI Tools

- **What:** Tools automatically generated from API specifications
- **Examples:** REST API endpoints become callable tools
- **Advantage:** No manual coding — just provide an API spec and get working tools

## 2. Built-in Tools



**What:** Pre-built tools provided by ADK

**Advantage:** No development time — use immediately with zero setup

**Gemini Tools** ✓ (You've used these!)

- **What:** Tools that leverage Gemini's capabilities
- **Examples:** google\_search, BuiltInCodeExecutor
- **Advantage:** Reliable, tested tools that work out of the box

**Google Cloud Tools** [needs Google Cloud access]

- **What:** Tools for Google Cloud services and enterprise integration
- **Examples:** BigQueryToolset, SpannerToolset, APIHubToolset
- **Advantage:** Enterprise-grade database and API access with built-in security

## Third-party Tools

- **What:** Wrappers for existing tool ecosystems
- **Examples:** Hugging Face, Firecrawl, GitHub Tools
- **Advantage:** Reuse existing tool investments — no need to rebuild what already exists

## Congratulations!

You've successfully learned how to build agents that go beyond simple responses to take intelligent actions with custom tools. In this notebook, you learned:

1.  **Function Tools** - Converted Python functions into agent tools
2.  **Agent Tools** - Created specialist agents and used them as tools
3.  **Complete Toolkit** - Explored all ADK tool types and when to use them

### Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

## Learn More

Refer to the following documentation to learn more:

- [ADK Documentation](#)
- [ADK Tools Documentation](#)
- [ADK Custom Tools Guide](#)
- [ADK Function Tools](#)
- [ADK Plugins Overview](#)

## Next Steps

You've built the foundation of agent tool mastery.

Ready for the next challenge? Continue to the next notebook to learn about **tool patterns!**

Authors

[Laxmi Harikumar](#)

*Copyright 2025 Google LLC.*

```
# @title 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
#
# https://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.
```



## Agent Tool Patterns and Best Practices

**Welcome to Day-2 of the Kaggle 5-day Agents course!**

In the previous notebook, you learned how to add custom Python functions as tools to your agent. In this notebook, we'll take the next step: **consuming external MCP services** and handling **long-running operations**.

In this notebook, you'll learn how to:

- **Connect to external MCP servers**
- **Implement long-running operations** that can pause agent execution for external input
- **Build resumable workflows** that maintain state across conversation breaks

-  Understand when and how to use these patterns

## !! Please Read

  **Note:** **No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. [See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy` and `Edit` button in the top-right corner.

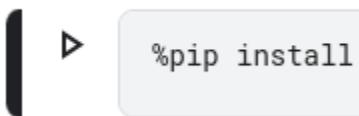


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

---

## Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

## 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires an API key.

### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

### 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

### 3. Authenticate in the notebook

Run the cell below to access the GOOGLE\_API\_KEY you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Setup and authentication complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
```

)

 Setup and authentication complete.

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit. This keeps your code organized and ensures we have access to the necessary building blocks.

```
import uuid
from google.genai import types

from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService

from google.adk.tools.mcp_tool.mcp_toolset import McpToolset
from google.adk.tools.tool_context import ToolContext
from google.adk.tools.mcp_tool.mcp_session_manager import
StdioConnectionParams
from mcp import StdioServerParameters

from google.adk.apps.app import App, ResumabilityConfig
from google.adk.tools.function_tool import FunctionTool

print("✅ ADK components imported successfully.")
```

 ADK components imported successfully.

### 1.4: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config = types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```

## Section 2: Model Context Protocol

So far, you have learned how to create custom functions for your agents. But connecting to external systems (GitHub, databases, Slack) requires writing and maintaining API clients.

**Model Context Protocol (MCP)** is an open standard that lets agents use community-built integrations. Instead of writing your own integrations and API clients, just connect to an existing MCP server.

MCP enables agents to:

- Access live, external data** from databases, APIs, and services without custom integration code
- Leverage community-built tools** with standardized interfaces
- Scale capabilities** by connecting to multiple specialized servers

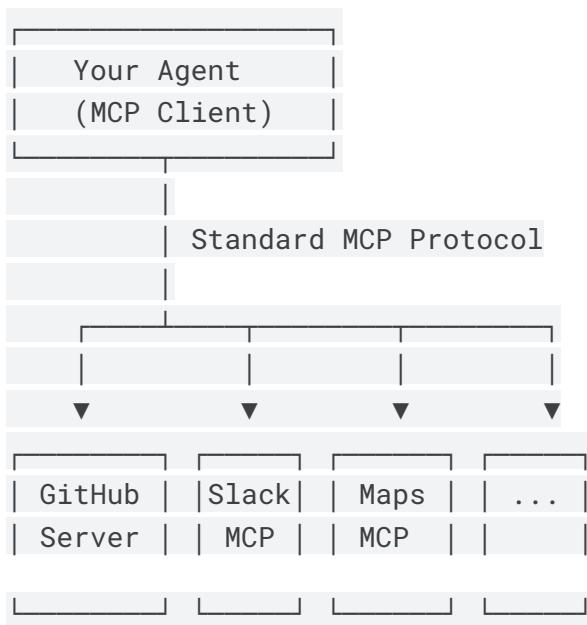
### 2.1: How MCP Works

MCP connects your agent (the **client**) to external **MCP servers** that provide tools:

- **MCP Server:** Provides specific tools (like image generation, database access)
- **MCP Client:** Your agent that uses those tools

- All servers work the same way - standardized interface

### Architecture:



## 2.2: Using MCP with Your Agent

The workflow is simple:

1. Choose an MCP Server and tool
2. Create the MCP Toolset (configure connection)
3. Add it to your agent
4. Run and test the agent

### Step 1: Choose MCP Server

For this demo, we'll use the [Everything MCP Server](#) - an npm package

(@modelcontextprotocol/server-everything) designed for testing MCP integrations.

It provides a `getTinyImage` tool that returns a simple test image (16x16 pixels, Base64-encoded).

**Find more servers:** [modelcontextprotocol.io/examples](http://modelcontextprotocol.io/examples)

**!! NOTE: This is a demo server to learn MCP.** In production, you'll use servers for Google Maps, Slack, Discord, etc.

## Step 2: Create the MCP Toolset

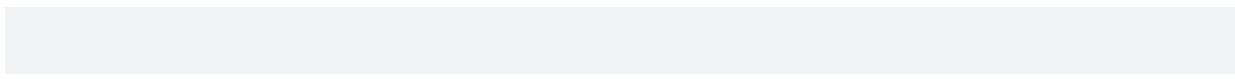
The McpToolset is used to integrate an ADK Agent with an MCP Server.

What the code does:

- Uses npx (Node package runner) to run the MCP server
- Connects to @modelcontextprotocol/server-everything
- Filters to only use the getTinyImage tool (the server has others, but we only need this one)

```
# MCP integration with Everything Server
mcp_image_server = McpToolset(
    connection_params=StdioConnectionParams(
        server_params=StdioServerParameters(
            command="npx", # Run MCP server via npx
            args=[
                "-y", # Argument for npx to auto-confirm install
                "@modelcontextprotocol/server-everything",
            ],
            tool_filter=[ "getTinyImage" ],
        ),
        timeout=30,
    )
)

print("✅ MCP Tool created")
```



✅ MCP Tool created

Behind the scenes:

1. **Server Launch:** ADK runs npx -y @modelcontextprotocol/server-everything
2. **Handshake:** Establishes stdio communication channel
3. **Tool Discovery:** Server tells ADK: "I provide getTinyImage" functionality
4. **Integration:** Tools appear in agent's tool list automatically

5. **Execution:** When agent calls `getTinyImage()`, ADK forwards to MCP server
6. **Response:** Server result is returned to agent seamlessly

**Why This Matters:** You get instant access to tools without writing integration code!

### Step 3: Add MCP tool to agent

Let's add the `mcp_server` to the agent's tool array and update the agent's instructions to handle requests to generate tiny images.

```
# Create image agent with MCP integration
image_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="image_agent",
    instruction="Use the MCP Tool to generate images for user queries",
    tools=[mcp_image_server],
)
```

Create the runner:

```
from google.adk.runners import InMemoryRunner

runner = InMemoryRunner(agent=image_agent)
```

### Step 4: Test the agent

Ask the agent to generate an image. Watch it use the MCP tool:

```
response = await runner.run_debug("Provide a sample tiny image",
verbose=True)
```

```
### Created new session: debug_session_id
```

User > Provide a sample tiny image

```
/usr/local/lib/python3.11/dist-packages/google/adk/tools/mcp_tool/mcp_tool.py:101: UserWarning: [EXPERIMENTAL] BaseAuthenticatedTool: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
```

```
super().__init__()  
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
```

```
image_agent > [Calling tool: getTinyImage({})]  
image_agent > [Tool result: {'content': [{'type': 'text', 'text': 'This is a tiny image:'}, {'type': 'image', 'data': 'iVBORw0KG...'}]}  
image_agent > This is a tiny image:  
The image above is the MCP tiny image.
```

### Display the image:

The server returns base64-encoded image data. Let's decode and display it:

```
from IPython.display import display, Image as IPIImage  
import base64  
  
for event in response:  
    if event.content and event.content.parts:  
        for part in event.content.parts:  
            if hasattr(part, "function_response") and  
part.function_response:  
                for item in part.function_response.response.get("content",  
[]):  
                    if item.get("type") == "image":  
  
display(IPIImage(data=base64.b64decode(item["data"])))
```



## 2.3: Extending to Other MCP Servers

The same pattern works for any MCP server - only the `connection_params` change. Here are some examples:

### 👉 Kaggle MCP Server - For dataset and notebook operations

Kaggle provides an MCP server that lets your agents interact with Kaggle datasets, notebooks, and competitions.

#### Connection example:

```
McpToolset(  
    connection_params=StdioConnectionParams(  
        server_params=StdioServerParameters(  
            command='npx',  
            args=[  
                '-y',  
                'mcp-remote',  
                'https://www.kaggle.com/mcp'  
            ],  
        ),  
        timeout=30,  
    )  
)
```

#### What it provides:

- 📊 Search and download Kaggle datasets
- 📃 Access notebook metadata
- 🏆 Query competition information etc.,

Learn more: [Kaggle MCP Documentation](#)

👉 GitHub MCP Server - For PR/Issue analysis

```
McpToolset(  
    connection_params=StreamableHTTPServerParams(  
        url="https://api.githubcopilot.com/mcp/",  
        headers={  
            "Authorization": f"Bearer {GITHUB_TOKEN}",  
            "X-MCP-Toolsets": "all",  
            "X-MCP-Readonly": "true"  
        },  
    ),  
)
```

More resources: [ADK Third-party Tools Documentation](#)

---

## ➡️ Section 3: Long-Running Operations (Human-in-the-Loop)

So far, all tools execute and return immediately:

User asks → Agent calls tool → Tool returns result → Agent responds

**But what if your tools are long-running or you need human approval before completing an action?**

Example: A shipping agent should ask for approval before placing a large order.

User asks → Agent calls tool → Tool PAUSES and asks human → Human approves → Tool completes → Agent responds

This is called a **Long-Running Operation (LRO)** - the tool needs to pause, wait for external input (human approval), then resume.

### When to use Long-Running Operations:

- **Financial transactions** requiring approval (transfers, purchases)
- **Bulk operations** (delete 1000 records - confirm first!)
- **Compliance checkpoints** (regulatory approval needed)
- **High-cost actions** (spin up 50 servers - are you sure?)
- **Irreversible operations** (permanently delete account)

### 3.1: What We're Building Today

Let's build a **shipping coordinator agent** with one tool that:

- Auto-approves small orders ( $\leq 5$  containers)
- **Pauses and asks for approval** on large orders ( $> 5$  containers)
- Completes or cancels based on the approval decision

This demonstrates the core long-running operation pattern: **pause → wait for human input → resume**.

### 3.2: The Shipping Tool with Approval Logic

Here's the complete function.

#### The ToolContext Parameter

Notice the function signature includes `tool_context: ToolContext`. ADK automatically provides this object when your tool runs. It gives you two key capabilities:

1. **Request approval:** Call `tool_context.request_confirmation()`
2. **Check approval status:** Read `tool_context.tool_confirmation`

```
LARGE_ORDER_THRESHOLD = 5
```

```
def place_shipping_order(
    num_containers: int, destination: str, tool_context: ToolContext
) -> dict:
    """Places a shipping order. Requires approval if ordering more than 5
    containers (LARGE_ORDER_THRESHOLD)."""

    Args:
        num_containers: Number of containers to ship
        destination: Shipping destination

    Returns:
        Dictionary with order status
    """
    #

    # SCENARIO 1: Small orders (<=5 containers) auto-approve
    if num_containers <= LARGE_ORDER_THRESHOLD:
        return {
            "status": "approved",
            "order_id": f"ORD-{num_containers}-AUTO",
            "num_containers": num_containers,
            "destination": destination,
            "message": f"Order auto-approved: {num_containers} containers
to {destination}",
        }

    #

    # SCENARIO 2: This is the first time this tool is called. Large orders
    # need human approval - PAUSE here.
    if not tool_context.tool_confirmation:
```

```

        tool_context.request_confirmation(
            hint=f"⚠ Large order: {num_containers} containers to
{destination}. Do you want to approve?",
            payload={"numContainers": num_containers, "destination": destination},
        )
        return { # This is sent to the Agent
            "status": "pending",
            "message": f"Order for {num_containers} containers requires
approval",
        }

#
-----#
-----#
-----#
# SCENARIO 3: The tool is called AGAIN and is now resuming. Handle
approval response - RESUME here.
if tool_context.tool_confirmation.confirmed:
    return {
        "status": "approved",
        "order_id": f"ORD-{num_containers}-HUMAN",
        "numContainers": num_containers,
        "destination": destination,
        "message": f"Order approved: {num_containers} containers to
{destination}",
    }
else:
    return {
        "status": "rejected",
        "message": f"Order rejected: {num_containers} containers to
{destination}",
    }

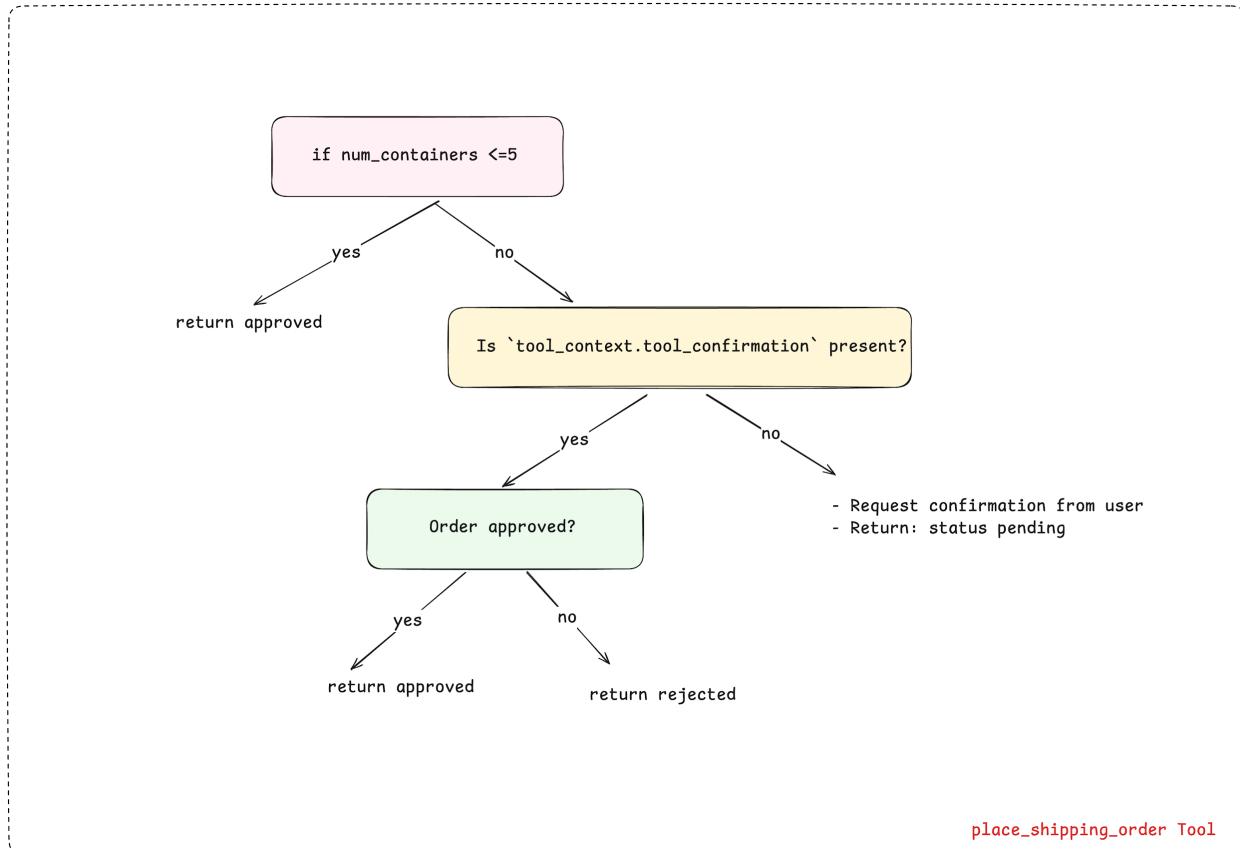
print("✅ Long-running functions created!")

```

✓ Long-running functions created!

### 3.3: Understanding the Code

Now that you've seen the complete function, let's break down how it works.



#### How the Three Scenarios Work

The tool handles three scenarios by checking `tool_context.tool_confirmation`:

**Scenario 1: Small order ( $\leq 5$  containers):** Returns immediately with auto-approved status.

- `tool_context.tool_confirmation` is never checked

**Scenario 2: Large order - FIRST CALL**

- Tool detects it's a first call: `if not tool_context.tool_confirmation:`
- Calls `request_confirmation()` to request human approval
- Returns `{'status': 'pending', ...}` immediately
- **ADK automatically creates adk\_request\_confirmation event**
- Agent execution pauses - waiting for human decision

### Scenario 3: Large order - RESUMED CALL

- Tool detects it's resuming: `if not tool_context.tool_confirmation:` is now False
- Checks human decision: `tool_context.tool_confirmation.confirmed`
- If True → Returns approved status
- If False → Returns rejected status

**Key insight:** Between the two calls, your workflow code (in Section 4) must detect the `adk_request_confirmation` event and resume with the approval decision.

## 3.4: Create the Agent, App and Runner

### Step 1: Create the agent

Add the tool to the Agent. The tool decides internally when to request approval based on the order size.

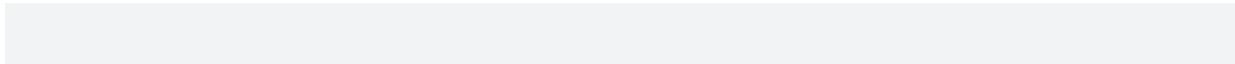
```
# Create shipping agent with pausable tool
shipping_agent = LlmAgent(
    name="shipping_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    instruction="""You are a shipping coordinator assistant.
```

When users request to ship containers:

1. Use the `place_shipping_order` tool with the number of containers and destination
2. If the order status is 'pending', inform the user that approval is required
3. After receiving the final result, provide a clear summary including:
  - Order status (approved/rejected)
  - Order ID (if available)
  - Number of containers and destination

```
4. Keep responses concise but informative
"""
    tools=[FunctionTool(func=place_shipping_order)],
)

print("✅ Shipping Agent created!")
```



```
✅ Shipping Agent created!
```

## Step 2: Wrap in resumable App

**The problem:** A regular LlmAgent is stateless - each call is independent with no memory of previous interactions. If a tool requests approval, the agent can't remember what it was doing.

**The solution:** Wrap your agent in an **App** with **resumability enabled**. The App adds a persistence layer that saves and restores state.

### What gets saved when a tool pauses:

- All conversation messages so far
- Which tool was called (place\_shipping\_order)
- Tool parameters (10 containers, Rotterdam)
- Where exactly it paused (waiting for approval)

When you resume, the App loads this saved state so the agent continues exactly where it left off - as if no time passed.

```
# Wrap the agent in a resumable app - THIS IS THE KEY FOR LONG-RUNNING
OPERATIONS!
shipping_app = App(
    name="shipping_coordinator",
    root_agent=shipping_agent,
    resumability_config=ResumabilityConfig(is_resumable=True),
)
```

```
print("✓ Resumable app created!")
```

✓ Resumable app created!

```
/tmp/ipykernel_48/3673777575.py:5: UserWarning: [EXPERIMENTAL]  
ResumabilityConfig: This feature is experimental and may change or be  
removed in future versions without notice. It may introduce breaking  
changes at any time.  
resumability_config=ResumabilityConfig(is_resumable=True),
```

### Step 3: Create Session and Runner with the App

Pass `app=shipping_app` instead of `agent=...` so the runner knows about resumability.

```
session_service = InMemorySessionService()  
  
# Create runner with the resumable app  
shipping_runner = Runner(  
    app=shipping_app, # Pass the app instead of the agent  
    session_service=session_service,  
)  
  
print("✓ Runner created!")
```

✓ Runner created!

---

✓ Recap: Your pausable shipping agent is now complete!

You've created:

1.  A tool that can pause for approval (`place_shipping_order`)
2.  An agent that uses this tool (`shipping_agent`)
3.  A resumable app that saves state (`shipping_app`)
4.  A runner that can handle pause/resume (`shipping_runner`)

**Next step:** Build the workflow code and test that our Agent detects pauses and handles approvals.

---

## Section 4: Building the Workflow

**!! Important:** The workflow code uses ADK concepts like Sessions, Runners, and Events. We'll cover what you need to know for long-running operations in this notebook. For deeper understanding, we will cover these topics in Day 3, or you can check out the [ADK docs](#) and this [video](#).

### 4.1: The Critical Part - Handling Events in Your Workflow

The agent won't automatically handle pause/resume. **Every long-running operation workflow requires you to:**

1. **Detect the pause:** Check if events contain `adk_request_confirmation`
2. **Get human decision:** In production, show UI and wait for user click. Here, we simulate it.
3. **Resume the agent:** Send the decision back with the saved `invocation_id`

### 4.2 Understand Key Technical Concepts

 **events** - ADK creates events as the agent executes. Tool calls, model responses, function results - all become events

 **adk\_request\_confirmation event** - This event is special - it signals "pause here!"

- Automatically created by ADK when your tool calls `request_confirmation()`
- Contains the `invocation_id`
- Your workflow must detect this event to know the agent paused

 **invocation\_id** - Every call to `run_async()` gets a unique `invocation_id` (like "abc123")

- When a tool pauses, you save this ID
- When resuming, pass the same ID so ADK knows which execution to continue
- Without it, ADK would start a NEW execution instead of resuming the paused one

### 4.3: Helper Functions to Process Events

These handle the event iteration logic for you.

**check\_for\_approval()** - Detects if the agent paused

- Loops through all events and looks for the special adk\_request\_confirmation event
- Returns approval\_id (identifies this specific request) and invocation\_id (identifies which execution to resume)
- Returns None if no pause detected

```
def check_for_approval(events):
    """Check if events contain an approval request.

    Returns:
        dict with approval details or None
    """
    for event in events:
        if event.content and event.content.parts:
            for part in event.content.parts:
                if (
                    part.function_call
                    and part.function_call.name ==
                    "adk_request_confirmation"
                ):
                    return {
                        "approval_id": part.function_call.id,
                        "invocation_id": event.invocation_id,
                    }
    return None
```

**print\_agent\_response()** - Displays agent text

- Simple helper to extract and print text from events

```
def print_agent_response(events):
    """Print agent's text responses from events."""
    for event in events:
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.text:
                    print(f"Agent > {part.text}")
```

`create_approval_response()` - Formats the human decision

- Takes the approval info and boolean decision (True/False) from the human
- Creates a FunctionResponse that ADK understands
- Wraps it in a Content object to send back to the agent

```
def create_approval_response(approval_info, approved):
    """Create approval response message."""
    confirmation_response = types.FunctionResponse(
        id=approval_info["approval_id"],
        name="adk_request_confirmation",
        response={"confirmed": approved},
    )
    return types.Content(
        role="user",
        parts=[types.Part(function_response=confirmation_response)]
    )

print("✓ Helper functions defined")
```

Helper functions defined

4.4: The Workflow Function - Let's tie it all together!

The `run_shipping_workflow()` function orchestrates the entire approval flow.

Look for the code explanation in the cell below.



```
async def run_shipping_workflow(query: str, auto_approve: bool = True):  
    """Runs a shipping workflow with approval handling.
```

*Args:*

```
    query: User's shipping request  
    auto_approve: Whether to auto-approve large orders (simulates human  
decision)  
    """
```

```
    print(f"\n{'='*60}")  
    print(f"User > {query}\n")
```

```
# Generate unique session ID  
session_id = f"order_{uuid.uuid4().hex[:8]}"  
  
# Create session  
await session_service.create_session(  
    app_name="shipping_coordinator", user_id="test_user",  
    session_id=session_id  
)
```

```
query_content = types.Content(role="user",  
parts=[types.Part(text=query)])  
events = []
```

```
#-----  
-----  
#-----  
-----
```

```

# STEP 1: Send initial request to the Agent. If num_containers > 5, the
Agent returns the special `adk_request_confirmation` event
    async for event in shipping_runner.run_async(
        user_id="test_user", session_id=session_id,
        new_message=query_content
    ):
        events.append(event)

#
-----#
-----#
# STEP 2: Loop through all the events generated and check if
`adk_request_confirmation` is present.
approval_info = check_for_approval(events)

#
-----#
-----#
# STEP 3: If the event is present, it's a large order - HANDLE APPROVAL
WORKFLOW
if approval_info:
    print(f"\n  Pausing for approval...")
    print(f"\n  Human Decision: {'APPROVE' ✅' if auto_approve else
'REJECT' ❌'}\n")

    # PATH A: Resume the agent by calling run_async() again with the
    approval decision
    async for event in shipping_runner.run_async(
        user_id="test_user",
        session_id=session_id,
        new_message=create_approval_response(
            approval_info, auto_approve
        ), # Send human decision here
        invocation_id=approval_info[

```

```

        "invocation_id"
    ], # Critical: same invocation_id tells ADK to RESUME
):
    if event.content and event.content.parts:
        for part in event.content.parts:
            if part.text:
                print(f"Agent > {part.text}")

#
-----
-----
#
-----
-----
else:
    # PATH B: If the `adk_request_confirmation` is not present - no
    # approval needed - order completed immediately.
    print_agent_response(events)

    print('='*60)

print("✅ Workflow function ready")

```

✅ Workflow function ready

## Code breakdown

### Step 1: Send initial request to the Agent

- Call `run_async()` to start agent execution
- Collect all events in a list for inspection

### Step 2: Detect Pause

- Call `check_for_approval(events)` to look for the special event: `adk_request_confirmation`
- Returns approval info (with `invocation_id`) if the special event is present; None if completed

### Step 3: Resume execution

PATH A:

- If the approval info is present, at this point the Agent *pauses* for human input.
- Once the Human input is available, call the agent again using `run_async()` and pass in the Human input.
- **Critical:** Same `invocation_id` (tells ADK to RESUME, not restart)
- Display agent's final response after resuming

PATH B:

- If the approval info is not present, then approval is not needed and the agent completes execution.

## 4.5: Demo: Testing the Workflow

Now, let's run our demos. Notice how much cleaner and easier to read they are. All the complex logic for pausing and resuming is now hidden away in our `run_workflow` helper function, allowing us to focus on the tasks we want the agent to perform.

**Note:** You may see warnings like `Warning: there are non-text parts in the response: [ 'function_call' ]` - this is normal and can be ignored. It just means the agent is calling tools in addition to generating text.

```
# Demo 1: It's a small order. Agent receives auto-approved status from tool
await run_shipping_workflow("Ship 3 containers to Singapore")

# Demo 2: Workflow simulates human decision: APPROVE ✅
await run_shipping_workflow("Ship 10 containers to Rotterdam",
auto_approve=True)

# Demo 3: Workflow simulates human decision: REJECT ❌
```

```
await run_shipping_workflow("Ship 8 containers to Los Angeles",
auto_approve=False)
```

```
User > Ship 3 containers to Singapore
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
```

```
Agent > 3 containers to Singapore have been shipped. The order has been
approved and the order ID is ORD-3-AUTO.
```

```
User > Ship 10 containers to Rotterdam
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
```

```
/usr/local/lib/python3.11/dist-packages/google/adk/tools/tool_context.py:9
2: UserWarning: [EXPERIMENTAL] ToolConfirmation: This feature is
experimental and may change or be removed in future versions without
notice. It may introduce breaking changes at any time.
```

```
    ToolConfirmation(
```

```
/usr/local/lib/python3.11/dist-packages/google/adk/agents/invocation_conte
xt.py:298: UserWarning: [EXPERIMENTAL] BaseAgentState: This feature is
experimental and may change or be removed in future versions without
notice. It may introduce breaking changes at any time.
```

```
self.agent_states[event.author] = BaseAgentState()
```

!! Pausing for approval...  
🤔 Human Decision: APPROVE ✓

Agent > Order approved. 10 containers will be shipped to Rotterdam. The order ID is ORD-10-HUMAN.

```
=====
```

```
=====
```

User > Ship 8 containers to Los Angeles

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

!! Pausing for approval...  
🤔 Human Decision: REJECT ✗

Agent > The order for 8 containers to Los Angeles has been rejected.

```
=====
```

#### 4.6: (Optional) Complete execution flow

Here's an example trace of the whole workflow.

**TL;DR:** Tool pauses at TIME 6, workflow detects it at TIME 8, resumes at TIME 10 with same invocation\_id="abc123".

### Detailed timeline:

Here's what happens step-by-step when you run `run_shipping_workflow("Ship 10 containers to Rotterdam", auto_approve=True)`:

```
TIME 1: User sends "Ship 10 containers to Rotterdam"
    ↓
TIME 2: Workflow calls shipping_runner.run_async(...)
    ADK assigns a unique invocation_id = "abc123"
    ↓
TIME 3: Agent receives user message, decides to use place_shipping_order
    tool
    ↓
TIME 4: ADK calls place_shipping_order(10, "Rotterdam", tool_context)
    ↓
TIME 5: Tool checks: num_containers (10) > 5
    Tool calls tool_context.request_confirmation(...)
    ↓
TIME 6: Tool returns {'status': 'pending', ...}
    ↓
TIME 7: ADK creates adk_request_confirmation event with
    invocation_id="abc123"
    ↓
TIME 8: Workflow detects the event via check_for_approval()
    Saves approval_id and invocation_id="abc123"
    ↓
TIME 9: Workflow gets human decision → True (approve)
    ↓
TIME 10: Workflow calls shipping_runner.run_async(...,
    invocation_id="abc123")
    Passes approval decision as FunctionResponse
    ↓
TIME 11: ADK sees invocation_id="abc123" - knows to RESUME (instead of
    starting new)
    Loads saved state from TIME 7
    ↓
TIME 12: ADK calls place_shipping_order again with same parameters
    But now tool_context.tool_confirmation.confirmed = True
    ↓
```

```
TIME 13: Tool returns {'status': 'approved', 'order_id': 'ORD-10-HUMAN',  
...}  
↓
```

```
TIME 14: Agent receives result and responds to user
```

**Key point:** The `invocation_id` is how ADK knows to resume the paused execution instead of starting a new one.

---



## Section 5: Summary - Key Patterns for Advanced Tools

In this notebook, you implemented two powerful, production-ready patterns for extending your agent's capabilities beyond simple functions.

Pattern	When to Use It	Key ADK Components
MCP Integration	You need to connect to <b>external, standardized services</b> (like time, databases, or file systems) without writing custom integration code.	McpToolset
Long-Running Operations	You need to <b>pause a workflow</b> to wait for an external event, most commonly for <b>human-in-the-loop</b> approvals or long background tasks or for compliance/security checkpoints.	ToolContext, request_confirmation, App, ResumabilityConfig



### Production Ready Concepts

You now understand how to build agents that:

-  **Scale**: Leverage community tools instead of building everything
-  **Handle Time**: Manage operations that span minutes, hours, or days
-  **Ensure Compliance**: Add human oversight to critical operations
-  **Maintain State**: Resume conversations exactly where they paused

**Start Simple**: Begin with custom tools → Add MCP services → Add long-running as needed

## Exercise: Build an Image Generation Agent with Cost Approval

**The scenario:**

Build an agent that generates images using the MCP server, but requires approval for "bulk" image generation:

- Single image request (1 image): Auto-approve, generate immediately
  - Bulk request (>1 image): Pause and ask for approval before generating multiple images
  - Explore different publicly available Image Generation MCP Servers
- 



Congratulations! You've Learned Agent Patterns and Best Practices

You've successfully learned how to build agents that handle complex, real-world workflows integrating external systems and spanning time.

### Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

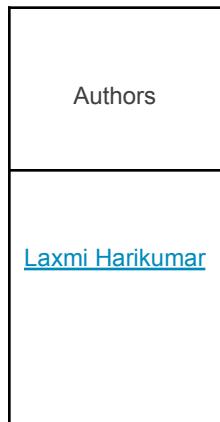
 [Learn More](#)

- [ADK Documentation](#)
- [MCP Tools Documentation](#)
- [Long-Running Operations Guide](#)
- [Model Context Protocol Specification](#)
- [The App and Runner](#)

## Next Steps

You've built the foundation for production-ready agent systems. Ready for the next challenge?

Continue to **Day 3** to learn about **State and Memory Management!**



*Copyright 2025 Google LLC.*

```
# @title 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
#
# https://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.
```



# Memory Management - Part 1 - Sessions

Welcome to Day 3 of the Kaggle 5-day Agents course!

In this notebook, you'll learn:

- What sessions are and how to use them in your agent
- How to build *stateful* agents with sessions and events
- How to persist sessions in a database
- Context management practices such as context compaction
- Best practices for sharing session State

## !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. [See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Get started with Kaggle Notebooks [\\_](#)

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

## 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the Copy and Edit button in the top-right corner.

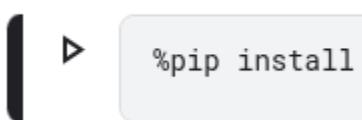


This creates a private copy of the notebook just for you.

## 3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶ Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

## 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

---

## Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires authentication.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

#### 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

#### 3. Authenticate in the notebook

Run the cell below to complete authentication.

```
import os
from kaggle_secrets import UserSecretsClient
```

```
try:  
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")  
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY  
    print("✓ Gemini API key setup complete.")  
except Exception as e:  
    print(  
        f"🔑 Authentication Error: Please make sure you have added  
'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"  
    )
```

✓ Gemini API key setup complete.

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from typing import Any, Dict  
  
from google.adk.agents import Agent, LlmAgent  
from google.adk.apps.app import App, EventsCompactionConfig  
from google.adk.models.google_llm import Gemini  
from google.adk.sessions import DatabaseSessionService  
from google.adk.sessions import InMemorySessionService  
from google.adk.runners import Runner  
from google.adk.tools.tool_context import ToolContext  
from google.genai import types  
  
print("✓ ADK components imported successfully.")
```

✓ ADK components imported successfully.

## 1.4: Helper functions

Helper function that manages a complete conversation session, handling session creation/retrieval, query processing, and response streaming. It supports both single queries and multiple queries in sequence.

Example:

```
>>> await run_session(runner, "What is the capital of France?",  
"geography-session")  
  
>>> await run_session(runner, ["Hello!", "What's my name?"],  
"user-intro-session")  
  
# Define helper functions that will be reused throughout the notebook  
async def run_session(  
    runner_instance: Runner,  
    user_queries: list[str] | str = None,  
    session_name: str = "default",  
):  
    print(f"\n *** Session: {session_name}")  
  
    # Get app name from the Runner  
    app_name = runner_instance.app_name  
  
    # Attempt to create a new session or retrieve an existing one  
    try:  
        session = await session_service.create_session(  
            app_name=app_name, user_id=USER_ID, session_id=session_name  
        )  
    except:  
        session = await session_service.get_session(  
            app_name=app_name, user_id=USER_ID, session_id=session_name  
        )  
  
    # Process queries if provided  
    if user_queries:
```

```

# Convert single query to list for uniform processing
if type(user_queries) == str:
    user_queries = [user_queries]

# Process each query in the list sequentially
for query in user_queries:
    print(f"\nUser > {query}")

# Convert the query string to the ADK Content format
query = types.Content(role="user",
parts=[types.Part(text=query)])

# Stream the agent's response asynchronously
async for event in runner_instance.run_async(
    user_id=USER_ID, session_id=session.id, new_message=query
):
    # Check if the event contains valid content
    if event.content and event.content.parts:
        # Filter out empty or "None" responses before printing
        if (
            event.content.parts[0].text != "None"
            and event.content.parts[0].text
        ):
            print(f"{MODEL_NAME} > ",
event.content.parts[0].text)
        else:
            print("No queries!")

print("✓ Helper functions defined.")

```

 Helper functions defined.

## 1.5: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config = types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```

---



## Section 2: Session Management

### 2.1 The Problem

At their core, Large Language Models are **inherently stateless**. Their awareness is confined to the information you provide in a single API call. This means an agent without proper context management will react to the current prompt without considering any previous history.

**?** **Why does this matter?** Imagine trying to have a meaningful conversation with someone who forgets everything you've said after each sentence. That's the challenge we face with raw LLMs!

In ADK, we use Sessions for **short term memory management** and Memory for **long term memory**. In the next notebook, you'll focus on Memory.

### 2.2 What is a Session?



#### Session

A session is a container for conversations. It encapsulates the conversation history in a chronological manner and also records all tool interactions and responses for a **single, continuous conversation**. A session is tied to a user and agent; it is not shared with other users. Similarly, a session history for an Agent is not shared with other Agents.

In ADK, a **Session** is comprised of two key components Events and State:

### **Session.Events:**

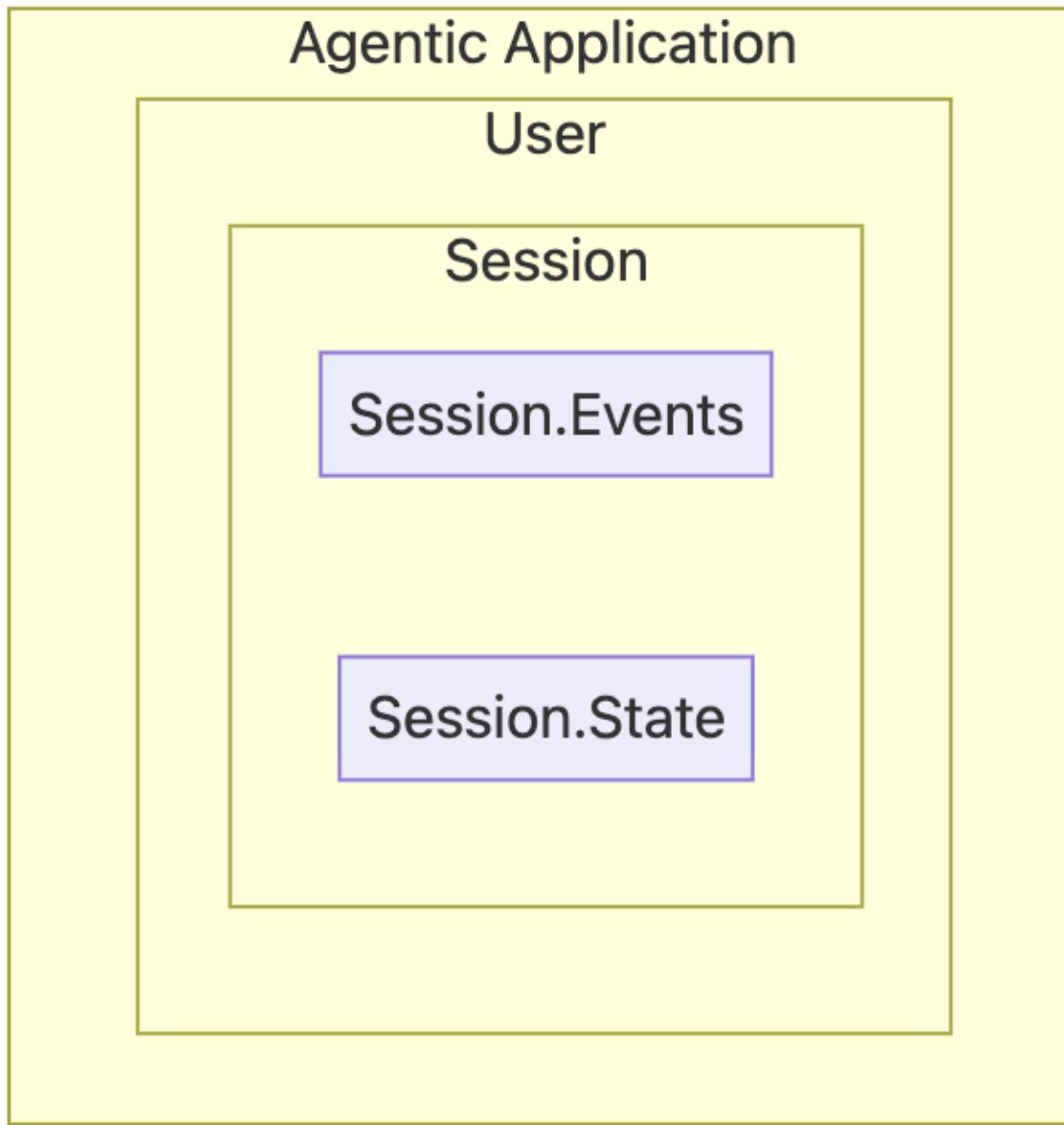
While a session is a container for conversations, Events are the building blocks of a conversation.

Example of Events:

- *User Input*: A message from the user (text, audio, image, etc.)
- *Agent Response*: The agent's reply to the user
- *Tool Call*: The agent's decision to use an external tool or API
- *Tool Output*: The data returned from a tool call, which the agent uses to continue its reasoning

### **Session.State:**

`session.state` is the Agent's scratchpad, where it stores and updates dynamic details needed during the conversation. Think of it as a global `{key, value}` pair storage which is available to all subagents and tools.



## 2.3 How to manage sessions?

An agentic application can have multiple users and each user may have multiple sessions with the application. To manage these sessions and events, ADK offers a **Session Manager** and **Runner**.

### 1. **SessionService**: The storage layer

- Manages creation, storage, and retrieval of session data
- Different implementations for different needs (memory, database, cloud)

## 2. **Runner**: The orchestration layer

- Manages the flow of information between user and agent
- Automatically maintains conversation history
- Handles the Context Engineering behind the scenes

Think of it like this:

- **Session** = A notebook 
- **Events** = Individual entries in a single page 
- **SessionService** = The filing cabinet storing notebooks 
- **Runner** = The assistant managing the conversation 

## 2.4 Implementing Our First Stateful Agent

Let's build our first stateful agent, that can remember and have constructive conversations.

ADK offers different types of sessions suitable for different needs. As a start, we'll start with a simple Session Management option (`InMemorySessionService`):

```
APP_NAME = "default" # Application
USER_ID = "default" # User
SESSION = "default" # Session

MODEL_NAME = "gemini-2.5-flash-lite"

# Step 1: Create the LLM Agent
root_agent = Agent(
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    name="text_chat_bot",
    description="A text chatbot", # Description of the agent's purpose
)

# Step 2: Set up Session Management
# InMemorySessionService stores conversations in RAM (temporary)
session_service = InMemorySessionService()

# Step 3: Create the Runner
```

```
runner = Runner(agent=root_agent, app_name=APP_NAME,
session_service=session_service)

print("✓ Stateful agent initialized!")
print(f" - Application: {APP_NAME}")
print(f" - User: {USER_ID}")
print(f" - Using: {session_service.__class__.__name__}")
```

```
✓ Stateful agent initialized!
- Application: default
- User: default
- Using: InMemorySessionService
```

## 2.5 Testing Our Stateful Agent

Now let's see the magic of sessions in action!

```
# Run a conversation with two queries in the same session
# Notice: Both queries are part of the SAME session, so context is
# maintained
await run_session(
    runner,
    [
        "Hi, I am Sam! What is the capital of United States?",
        "Hello! What is my name?", # This time, the agent should remember!
    ],
    "stateful-agentic-session",
)
```

```
### Session: stateful-agentic-session
```

```
User > Hi, I am Sam! What is the capital of United States?
gemini-2.5-flash-lite > Hi Sam! The capital of the United States is
Washington, D.C.
```

```
User > Hello! What is my name?  
gemini-2.5-flash-lite > Your name is Sam.
```

 **Success!** The agent remembered your name because both queries were part of the same session. The Runner automatically maintained the conversation history.

But there's a catch: InMemorySessionService is temporary. **Once the application stops, all conversation history is lost.**

### (Optional) 2.6 Testing Agent's forgetfulness

To verify that the agent forgets the conversation, **restart the kernel**. Then, **run ALL the previous cells in this notebook EXCEPT the run\_session in 2.5.**

Now run the cell below. You'll see that the agent doesn't remember anything from the previous conversation.

```
# Run this cell after restarting the kernel. All this history will be  
gone...  
await run_session(  
    runner,  
    ["What did I ask you about earlier?", "And remind me, what's my  
name?"],  
    "stateful-agentic-session",  
) # Note, we are using same session name
```

```
### Session: stateful-agentic-session
```

```
User > What did I ask you about earlier?  
gemini-2.5-flash-lite > You asked me what the capital of the United  
States is.
```

```
User > And remind me, what's my name?
```

gemini-2.5-flash-lite > Your name is Sam.

## The Problem

Session information is not persistent (i.e., meaningful conversations are lost). While this is advantageous in testing environments, **in the real world, a user should be able to refer from past and resume conversations.** To achieve this, we must persist information.

---



## Section 3: Persistent Sessions with `DatabaseSessionService`

While `InMemorySessionService` is great for prototyping, real-world applications need conversations to survive restarts, crashes, and deployments. Let's level up to persistent storage!

### 3.1 Choosing the Right SessionService

ADK provides different SessionService implementations for different needs:

Service	Use Case	Persistence	Best For
<code>InMemorySessionService</code>	Development & Testing	✗ Lost on restart	Quick prototypes
<code>DatabaseSessionService</code>	Self-managed apps	✓ Survives restarts	Small to medium apps

<b>Agent Engine Sessions</b>	Production on GCP	<input checked="" type="checkbox"/> Fully managed	Enterprise scale
------------------------------	-------------------	---	------------------

### 3.2 Implementing Persistent Sessions

Let's upgrade to DatabaseSessionService using SQLite. This gives us persistence without needing a separate database server for this demo.

Let's create a chatbot\_agent capable of having a conversation with the user.

```
# Step 1: Create the same agent (notice we use LlmAgent this time)
chatbot_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    name="text_chat_bot",
    description="A text chatbot with persistent memory",
)

# Step 2: Switch to DatabaseSessionService
# SQLite database will be created automatically
db_url = "sqlite:///my_agent_data.db" # Local SQLite file
session_service = DatabaseSessionService(db_url=db_url)

# Step 3: Create a new runner with persistent storage
runner = Runner(agent=chatbot_agent, app_name=APP_NAME,
session_service=session_service)

print("✅ Upgraded to persistent sessions!")
print(f"  - Database: my_agent_data.db")
print(f"  - Sessions will survive restarts!")
```

✅ Upgraded to persistent sessions!  
 - Database: my\_agent\_data.db  
 - Sessions will survive restarts!

### 3.3 Test Run 1: Verifying Persistence

In this first test run, we'll start a new conversation with the session ID `test-db-session-01`. We will first introduce our name as 'Sam' and then ask a question. In the second turn, we will ask the agent for our name.

Since we are using `DatabaseSessionService`, the agent should remember the name.

After the conversation, we'll inspect the `my_agent_data.db` SQLite database directly to see how the conversation events (the user queries and model responses) are stored.

```
await run_session(  
    runner,  
    ["Hi, I am Sam! What is the capital of the United States?", "Hello!  
What is my name?"],  
    "test-db-session-01",  
)
```

```
### Session: test-db-session-01
```

```
User > Hi, I am Sam! What is the capital of the United States?  
gemini-2.5-flash-lite > Hi Sam! The capital of the United States is  
Washington, D.C.
```

```
User > Hello! What is my name?  
gemini-2.5-flash-lite > Your name is Sam.
```

### 💡 (Optional) 3.4 Test Run 2: Resuming a Conversation

!! Now, let's repeat the test again, but this time, **let's stop this Kaggle Notebook's kernel and restart it again.**

1. Run all the previous cells in the notebook, **EXCEPT** the previous Section 3.3 (`run_session` cell).
2. Now, run the below cell with the **same session ID** (`test-db-session-01`).

We will ask a new question and then ask for our name again. **Because the session is loaded from the database, the agent should still remember** that our name is 'Sam' from the first test run. This demonstrates the power of persistent sessions.

```
await run_session(  
    runner,  
    ["What is the capital of India?", "Hello! What is my name?"],  
    "test-db-session-01",  
)
```

```
### Session: test-db-session-01
```

```
User > What is the capital of India?  
gemini-2.5-flash-lite > The capital of India is New Delhi.
```

```
User > Hello! What is my name?  
gemini-2.5-flash-lite > Your name is Sam.
```

### 3.5 Let's verify that the session data is isolated

As mentioned earlier, a session is private conversation between an Agent and a User (i.e., two sessions do not share information). Let's run our `run_session` with a different session name `test-db-session-02` to confirm this.

```
await run_session(  
    runner, ["Hello! What is my name?"], "test-db-session-02"  
) # Note, we are using new session name
```

```
### Session: test-db-session-02
```

```
User > Hello! What is my name?  
gemini-2.5-flash-lite > I do not have access to your personal  
information, including your name. I am a large language model, trained by  
Google.
```

### 3.6 How are the events stored in the Database?

Since we are using a sqlite DB to store information, let's have a quick peek to see how information is stored.

```
import sqlite3  
  
def check_data_in_db():  
    with sqlite3.connect("my_agent_data.db") as connection:  
        cursor = connection.cursor()  
        result = cursor.execute(  
            "select app_name, session_id, author, content from events"  
        )  
        print([_[0] for _ in result.description])  
        for each in result.fetchall():  
            print(each)  
  
check_data_in_db()
```

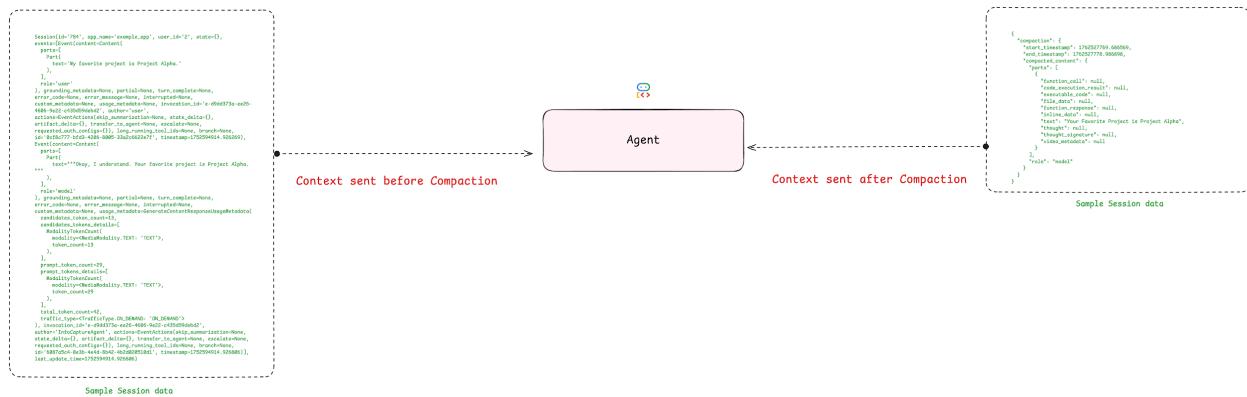
```
['app_name', 'session_id', 'author', 'content']  
('default', 'test-db-session-01', 'user', '{"parts": [{"text": "Hi, I am  
Sam! What is the capital of the United States?"}], "role": "user"}')  
('default', 'test-db-session-01', 'text_chat_bot', '{"parts": [{"text":  
"Hi Sam! The capital of the United States is Washington, D.C."}], "role":  
"model"}')  
('default', 'test-db-session-01', 'user', '{"parts": [{"text": "Hello!  
What is my name?"}], "role": "user"}')  
('default', 'test-db-session-01', 'text_chat_bot', '{"parts": [{"text":  
"Your name is Sam."}], "role": "model"}')
```

```
('default', 'test-db-session-01', 'user', '{"parts": [{"text": "What is the capital of India?"}], "role": "user"})\n('default', 'test-db-session-01', 'text_chat_bot', '{"parts": [{"text": "The capital of India is New Delhi."}], "role": "model"})\n('default', 'test-db-session-01', 'user', '{"parts": [{"text": "Hello! What is my name?"}], "role": "user"})\n('default', 'test-db-session-01', 'text_chat_bot', '{"parts": [{"text": "Your name is Sam."}], "role": "model"})\n('default', 'test-db-session-02', 'user', '{"parts": [{"text": "Hello! What is my name?"}], "role": "user"})\n('default', 'test-db-session-02', 'text_chat_bot', '{"parts": [{"text": "I do not have access to your personal information, including your name. I am a large language model, trained by Google."}], "role": "model"})
```

## Section 4: Context Compaction

As you can see, all the events are stored in full in the session Database, and this quickly adds up. For a long, complex task, this list of events can become very large, leading to slower performance and higher costs.

But what if we could automatically summarize the past? Let's use ADK's **Context Compaction** feature to see **how to automatically reduce the context that's stored in the Session**.



## 4.1 Create an App for the agent

To enable this feature, let's use the same `chatbot_agent` we created in Section 3.2.

The first step is to create an object called `App`. We'll give it a name and pass in our `chatbot_agent`.

We'll also create a new config to do the Context Compaction. This `EventsCompactionConfig` defines two key variables:

- `compaction_interval`: Asks the Runner to compact the history after every n conversations
- `overlap_size`: Defines the number of previous conversations to retain for overlap

We'll then provide this app to the Runner.

```
# Re-define our app with Events Compaction enabled
research_app_compacting = App(
    name="research_app_compacting",
    root_agent=chatbot_agent,
    # This is the new part!
    events_compaction_config=EventsCompactionConfig(
        compaction_interval=3, # Trigger compaction every 3 invocations
        overlap_size=1, # Keep 1 previous turn for context
    ),
)

db_url = "sqlite:///my_agent_data.db" # Local SQLite file
session_service = DatabaseSessionService(db_url=db_url)

# Create a new runner for our upgraded app
research_runner_compacting = Runner(
    app=research_app_compacting, session_service=session_service
)

print("✅ Research App upgraded with Events Compaction!")
```

✅ Research App upgraded with Events Compaction!

```
/tmp/ipykernel_48/3773147741.py:6: UserWarning: [EXPERIMENTAL]  
EventsCompactionConfig: This feature is experimental and may change or be  
removed in future versions without notice. It may introduce breaking  
changes at any time.  
    events_compaction_config=EventsCompactionConfig(
```

## 4.2 Running the Demo

Now, let's have a conversation that is long enough to trigger the compaction. When you run the cell below, the output will look like a normal conversation. However, because we configured our App, a compaction process will run silently in the background after the 3rd invocation.

In the next step, we'll prove that it happened.

```
# Turn 1  
await run_session(  
    research_runner_compacting,  
    "What is the latest news about AI in healthcare?",  
    "compaction_demo",  
)  
  
# Turn 2  
await run_session(  
    research_runner_compacting,  
    "Are there any new developments in drug discovery?",  
    "compaction_demo",  
)  
  
# Turn 3 - Compaction should trigger after this turn!  
await run_session(  
    research_runner_compacting,  
    "Tell me more about the second development you found.",  
    "compaction_demo",  
)  
  
# Turn 4
```

```
await run_session(  
    research_runner_compacting,  
    "Who are the main companies involved in that?",  
    "compaction_demo",  
)
```

```
### Session: compaction_demo
```

User > What is the latest news about AI in healthcare?

gemini-2.5-flash-lite > Here's a summary of the latest news and trends in AI in healthcare, covering a range of exciting developments:

\*\*Key Areas of Advancement & News:\*\*

\* \*\*Drug Discovery and Development:\*\*

- \* \*\*Accelerated Research:\*\* AI is revolutionizing drug discovery by rapidly analyzing vast datasets of biological and chemical information, identifying potential drug candidates, and predicting their efficacy and safety. Companies are seeing shorter timelines for preclinical research.

- \* \*\*Personalized Medicine:\*\* AI is enabling the development of targeted therapies based on an individual's genetic makeup and disease profile. This is leading to more effective treatments with fewer side effects.

- \* \*\*Example:\*\* Companies like Recursion Pharmaceuticals and Insilico Medicine are making headlines with their AI-driven drug discovery platforms.

\* \*\*Diagnostics and Imaging:\*\*

- \* \*\*Enhanced Image Analysis:\*\* AI algorithms are becoming increasingly adept at analyzing medical images (X-rays, CT scans, MRIs, pathology slides) to detect subtle abnormalities that might be missed by the human eye. This is crucial for early disease detection, especially in areas like cancer, diabetic retinopathy, and cardiovascular disease.

- \* \*\*Improved Accuracy and Speed:\*\* AI can process images much faster than humans, reducing radiologist workload and potentially speeding up diagnosis.

- \* \*\*Example:\*\* Google's AI for detecting diabetic retinopathy and various AI tools for identifying cancerous nodules in lung scans are prominent examples.

- \*    \*\*Personalized Treatment Plans and Predictive Analytics:\*\*
  - \*    \*\*Individualized Care:\*\* AI can analyze patient data (medical history, genomics, lifestyle) to create highly personalized treatment plans, optimizing dosages and therapies for better outcomes.
    - \*    \*\*Predicting Disease Risk:\*\* AI models are being developed to predict a patient's risk of developing certain diseases (e.g., sepsis, heart failure, readmission) allowing for proactive interventions.
    - \*    \*\*Example:\*\* AI is being used in intensive care units to predict the likelihood of patient deterioration, enabling timely medical response.
  
- \*    \*\*Robotics and Surgery:\*\*
  - \*    \*\*Assisted Surgery:\*\* AI is enhancing robotic-assisted surgery by providing surgeons with real-time guidance, improving precision, and enabling minimally invasive procedures.
    - \*    \*\*Automated Tasks:\*\* While fully autonomous surgery is still some way off, AI is automating certain repetitive surgical tasks, freeing up surgeons for more complex decision-making.
    - \*    \*\*Example:\*\* Robotic surgical systems like the da Vinci Surgical System are increasingly incorporating AI features for enhanced performance.
  
- \*    \*\*Administrative and Operational Efficiency:\*\*
  - \*    \*\*Streamlining Workflows:\*\* AI is being used to automate administrative tasks such as scheduling appointments, processing insurance claims, and managing electronic health records (EHRs), reducing burnout for healthcare professionals.
    - \*    \*\*Optimizing Resource Allocation:\*\* AI can help hospitals optimize staffing, bed allocation, and supply chain management, leading to more efficient operations.
    - \*    \*\*Example:\*\* AI-powered chatbots are handling patient inquiries and appointment scheduling, freeing up administrative staff.
  
- \*    \*\*Mental Health:\*\*
  - \*    \*\*Early Detection and Support:\*\* AI is being explored for early detection of mental health conditions through analysis of speech patterns, text messages, and social media activity.
  - \*    \*\*AI-Powered Therapy:\*\* Chatbots and virtual assistants are offering accessible mental health support, providing coping strategies and guidance.

- \* \*\*Example:\*\* Companies are developing AI tools to identify signs of depression or anxiety from digital footprints.

#### \*\*Emerging Trends and Challenges:\*\*

- \* \*\*Explainable AI (XAI):\*\* A major focus is on developing AI models that can explain their reasoning, fostering trust among clinicians and regulatory bodies.
- \* \*\*Data Privacy and Security:\*\* Ensuring the secure handling and privacy of sensitive patient data is paramount and a continuous area of development.
- \* \*\*Regulatory Hurdles:\*\* Navigating the complex regulatory landscape for AI-powered medical devices and treatments is an ongoing challenge.
- \* \*\*Integration into Clinical Practice:\*\* The successful adoption of AI requires seamless integration into existing healthcare workflows and systems.
- \* \*\*Ethical Considerations:\*\* Addressing biases in AI algorithms, ensuring equitable access to AI-driven healthcare, and defining accountability are critical ethical discussions.

#### \*\*Where to Find Latest News:\*\*

To stay truly up-to-date, I'd recommend following:

- \* \*\*Reputable Medical Journals:\*\* JAMA, The Lancet, NEJM, Nature Medicine.
- \* \*\*Technology News Outlets with Health Sections:\*\* TechCrunch, STAT News, FierceBiotech, MobiHealthNews.
- \* \*\*AI and Healthcare Conferences:\*\* HIMSS, RSNA, NeurIPS (medical tracks).
- \* \*\*Leading AI/Healthcare Company Announcements:\*\* Keep an eye on news from companies like Google Health, Microsoft Healthcare, IBM Watson Health, NVIDIA, and numerous innovative startups.

The field of AI in healthcare is evolving at an incredible pace, with continuous breakthroughs promising to transform patient care, research, and operational efficiency.

### Session: compaction\_demo

User > Are there any new developments in drug discovery?

gemini-2.5-flash-lite > Yes, there are \*\*significant and rapid new developments in drug discovery driven by AI\*\*. It's one of the most active and impactful areas of AI application in healthcare. Here's a breakdown of some of the latest trends and breakthroughs:

#### \*\*1. Generative AI for Novel Molecule Design:\*\*

- \* \*\*Creating Entirely New Compounds:\*\* Beyond just analyzing existing data, generative AI models (like those related to large language models but adapted for molecular structures) can now \*\*design novel molecules from scratch\*\*. These AI systems learn the underlying principles of molecular behavior and chemistry to propose compounds with desired properties that may not have been conceived by humans.
- \* \*\*"In Silico" Design:\*\* This allows for rapid exploration of vast chemical spaces, leading to promising drug candidates much faster than traditional methods.
- \* \*\*Example:\*\* Companies are using generative AI to design novel antibiotics, anti-cancer agents, and treatments for rare diseases.

#### \*\*2. Accelerated Target Identification and Validation:\*\*

- \* \*\*Unraveling Disease Mechanisms:\*\* AI is exceptionally good at sifting through massive datasets (genomics, proteomics, clinical data) to identify new biological targets implicated in diseases. This goes beyond known pathways to uncover novel connections.
- \* \*\*Predicting Target "Druggability":\*\* AI models can assess whether a newly identified target is likely to be successfully modulated by a drug, saving resources on targets that are unlikely to yield a therapeutic.

#### \*\*3. Predicting Drug Efficacy and Toxicity with Higher Accuracy:\*\*

- \* \*\*"Digital Twins" and Simulation:\*\* Advanced AI models can create more sophisticated in-vitro and in-vivo simulations of how a drug will behave in the human body, predicting its effectiveness and potential side effects with greater precision.
- \* \*\*Reducing Preclinical Failure Rates:\*\* By better predicting failures early on, AI helps reduce the costly attrition rates in preclinical drug development.

#### **\*\*4. Repurposing Existing Drugs:\*\***

- \*   **Finding New Uses:** AI can analyze the known properties of approved drugs and identify potential new therapeutic applications for diseases they weren't originally intended for. This is a faster and often cheaper route to new treatments.
- \*   **Example:** During the COVID-19 pandemic, AI played a role in identifying existing drugs that showed promise against the virus.

#### **\*\*5. Optimizing Clinical Trial Design and Patient Selection:\*\***

- \*   **Stratifying Patients:** AI can analyze patient data to identify subgroups that are most likely to respond to a particular drug. This leads to more efficient and successful clinical trials.
- \*   **Predicting Trial Outcomes:** AI models can help predict the likelihood of success for a clinical trial based on various factors, allowing for better resource allocation.

#### **\*\*6. AI for Antibody and Protein Design:\*\***

- \*   **Tailored Biologics:** AI is increasingly used to design novel antibodies and therapeutic proteins with specific binding properties and enhanced stability. This is crucial for developing treatments for complex diseases like autoimmune disorders and cancer.

#### **\*\*Key Companies and Trends to Watch:\*\***

- \*   **AI-Native Biotech Companies:** A surge of startups are being built entirely around AI-driven drug discovery platforms. Examples include **Insilico Medicine, Recursion Pharmaceuticals, Exscientia, Atomwise, and Schrödinger.**
- \*   **Big Pharma Partnerships:** Established pharmaceutical companies are heavily investing in AI, either by building their own capabilities or forging strategic partnerships with AI-focused biotech firms.
- \*   **Focus on Specific Diseases:** While AI can be applied broadly, there's a growing trend of companies focusing AI expertise on specific disease areas like oncology, neurodegenerative diseases, and infectious diseases.

#### **\*\*Challenges Remain:\*\***

Despite the rapid progress, challenges persist:

- \* \*\*Data Quality and Access:\*\* The effectiveness of AI heavily relies on high-quality, diverse, and well-annotated data, which can be a bottleneck.
- \* \*\*Interpretability (Explainable AI):\*\* Understanding \*why\* an AI model proposes a specific molecule or target is crucial for scientific validation and regulatory approval.
- \* \*\*Integration with Wet Lab Experiments:\*\* AI insights need to be rigorously validated through traditional laboratory experiments.
- \* \*\*Regulatory Acceptance:\*\* Regulatory bodies are still developing frameworks for evaluating AI-generated drug candidates.

In summary, AI is not just a tool in drug discovery; it's fundamentally changing the paradigm, enabling faster, more efficient, and more innovative approaches to finding new medicines. The pace of innovation in this area is breathtaking.

### Session: compaction\_demo

User > Tell me more about the second development you found.

gemini-2.5-flash-lite > You're likely referring to the \*\*second development I mentioned in the context of drug discovery, which is "Accelerated Target Identification and Validation."\*\*

This is a truly transformative area where AI is making a profound impact. Let me elaborate:

#### \*\*The Traditional Challenge of Target Identification:\*\*

For decades, identifying and validating a biological target (a specific molecule in the body, like a protein or gene, that a drug can interact with to produce a therapeutic effect) has been a slow, laborious, and often serendipitous process. Researchers would:

- \* \*\*Hypothesize based on existing knowledge:\*\* Start with what was already understood about a disease.
- \* \*\*Conduct extensive experiments:\*\* Perform countless lab tests, genetic studies, and biochemical assays.

\* \*\*Face high failure rates:\*\* Many identified targets turned out to be not as crucial as initially thought, or not "druggable" (meaning a molecule couldn't effectively bind to it).

#### \*\*How AI is Revolutionizing Target Identification and Validation:\*\*

AI, particularly machine learning and deep learning, excels at processing and finding patterns in massive, complex datasets that are far beyond human capacity. Here's how it's changing the game:

##### 1. \*\*Unprecedented Data Integration and Analysis:\*\*

\* \*\*Multi-omics Data:\*\* AI can simultaneously analyze data from genomics (DNA), transcriptomics (RNA), proteomics (proteins), metabolomics (metabolites), and more. This holistic view reveals intricate biological pathways and disease mechanisms that are impossible to discern from a single data type.

\* \*\*Real-World Evidence (RWE):\*\* AI can sift through electronic health records (EHRs), patient registries, and claims data to identify correlations between biological factors and disease outcomes.

\* \*\*Literature Mining:\*\* Natural Language Processing (NLP) AI can scan millions of scientific papers, patents, and clinical trial reports to extract relevant information, identify emerging trends, and connect seemingly unrelated research findings.

##### 2. \*\*Identifying Novel Disease Drivers:\*\*

\* \*\*Uncovering Hidden Relationships:\*\* AI algorithms can detect subtle, non-obvious correlations between genes, proteins, and disease states that human researchers might miss. This can lead to the identification of entirely new therapeutic targets previously unknown.

\* \*\*Network Biology:\*\* AI models can map complex biological networks, highlighting key nodes (potential targets) whose manipulation could have a significant impact on disease progression.

##### 3. \*\*Predicting "Druggability":\*\*

\* \*\*Assessing Target Suitability:\*\* Not all biological targets are good candidates for drug development. AI models can be trained to predict:

\* \*\*Binding Affinity:\*\* How well a potential drug molecule can bind to the target.

- \* \*\*Modulation Potential:\*\* Whether binding to the target will lead to the desired therapeutic effect (e.g., inhibiting an enzyme, activating a receptor).
- \* \*\*Off-Target Effects:\*\* The likelihood of the drug interacting with other unintended targets, which can lead to side effects.
- \* \*\*Prioritizing Promising Targets:\*\* This predictive capability allows researchers to focus their efforts and resources on targets that have a higher probability of success, significantly reducing wasted time and money.

#### 4. \*\*Accelerated Hypothesis Generation and Testing:\*\*

- \* \*\*Faster Insights:\*\* AI can generate hypotheses about potential targets much faster than traditional methods.
- \* \*\*Guiding Experimental Design:\*\* The insights from AI can inform and optimize the design of subsequent laboratory experiments, making the validation process more efficient.

#### \*\*Examples and Impact:\*\*

- \* \*\*Identifying new cancer targets:\*\* AI is helping to pinpoint specific genetic mutations or protein expressions that drive cancer growth, leading to the development of targeted therapies.
- \* \*\*Discovering targets for neurodegenerative diseases:\*\* By analyzing complex brain imaging and genetic data, AI is helping to uncover targets for Alzheimer's, Parkinson's, and other debilitating conditions.
- \* \*\*Finding novel targets for rare diseases:\*\* For diseases with limited understanding, AI can analyze sparse data to uncover potential targets that would be nearly impossible to find otherwise.

\*\*In essence, AI acts as an incredibly powerful microscope and detective for biology.\*\* It allows us to see and connect pieces of the biological puzzle that were previously invisible or too complex to assemble, thereby accelerating the identification and validation of the most promising targets for new drugs.

### Session: compaction\_demo

User > Who are the main companies involved in that?  
gemini-2.5-flash-lite > You're asking about the companies at the forefront of \*\*AI-driven Target Identification and Validation\*\* in drug

discovery. This is a rapidly evolving space with a mix of established players and nimble startups.

Here's a breakdown of the main categories of companies involved and some prominent examples:

**\*\*1. AI-Native Biotech Companies (Building Platforms from the Ground Up):\*\***

These companies are founded with AI at their core, specifically designed to accelerate drug discovery, with target identification being a critical first step.

- \* \*\*Recursion Pharmaceuticals:\*\* Known for its massive dataset of cellular images and its AI platform that analyzes these images to understand disease biology and identify drug targets. They focus on a wide range of therapeutic areas.
- \* \*\*Insilico Medicine:\*\* Employs generative AI and deep learning for target discovery, preclinical candidate generation, and has been notable for advancing candidates from AI discovery to clinical trials.
- \* \*\*Exscientia:\*\* Focuses on using AI to design novel molecules, but their platform also excels at identifying and validating novel targets by analyzing vast biological and chemical data.
- \* \*\*Atomwise:\*\* Specializes in using deep learning for structure-based drug discovery, which often starts with identifying and characterizing novel protein targets.
- \* \*\*BenevolentAI:\*\* Has a broad AI platform that integrates scientific literature, clinical data, and other sources to identify novel drug targets and therapeutic hypotheses.
- \* \*\*Cerebras Systems (in partnership):\*\* While primarily a hardware company, Cerebras provides powerful AI computing infrastructure that enables large-scale drug discovery efforts, including target identification, for their partners.

**\*\*2. Big Pharmaceutical Companies (Integrating AI into Existing R&D):\*\***

Major pharmaceutical companies are investing heavily in AI, either by building internal teams, acquiring AI startups, or forming strategic partnerships. They leverage AI to augment their existing target identification and validation processes.

- \* \*\*Pfizer:\*\* Has been actively integrating AI into its R&D, including target discovery and validation, often through collaborations.
- \* \*\*Novartis:\*\* Is a significant player in adopting AI for drug discovery, including target identification, leveraging both internal efforts and external partnerships.
- \* \*\*Roche/Genentech:\*\* Has been at the forefront of applying computational biology and AI to understand disease mechanisms and identify novel targets, especially in oncology.
- \* \*\*Sanofi:\*\* Investing in AI to accelerate various stages of drug discovery, including target identification.
- \* \*\*Amgen:\*\* Uses AI and machine learning for target identification and validation, particularly in areas like genomics and proteomics.
- \* \*\*Merck (MSD):\*\* Actively exploring AI applications to enhance their discovery pipelines, including target selection.
- \* \*\*Johnson & Johnson (Janssen):\*\* Employing AI to mine complex biological data for new target insights.

#### \*\*3. Cloud Computing & AI Infrastructure Providers:\*\*

While not directly doing drug discovery, these companies provide the essential AI tools, platforms, and computing power that enable the above companies to operate.

- \* \*\*NVIDIA:\*\* Their GPUs and specialized AI hardware/software (like Clara Discovery) are foundational for many AI drug discovery efforts, including processing massive biological datasets for target identification.
- \* \*\*Google Cloud / Amazon Web Services (AWS) / Microsoft Azure:\*\* These cloud providers offer scalable computing power and AI/ML services that are crucial for training complex models for target discovery.

#### \*\*4. Data & Analytics Companies:\*\*

Some companies focus on curating and providing the high-quality datasets that AI models need for target identification.

- \* \*\*Unnatural Products:\*\* Uses AI to explore natural product chemistry for drug discovery, which inherently involves understanding biological targets.

## \*\*Key Trends in Who is Involved:\*\*

- \* \*\*Partnerships are Key:\*\* The most effective strategies often involve AI-native companies partnering with large pharma, bringing together cutting-edge AI technology with deep biological expertise and clinical trial infrastructure.
- \* \*\*Acquisitions:\*\* Big pharma is acquiring successful AI biotech companies to bring their talent and technology in-house.
- \* \*\*Democratization of AI:\*\* Cloud providers and specialized AI software platforms are making powerful AI tools more accessible to a wider range of researchers and companies.

It's a dynamic landscape, and new players are constantly emerging, often with specialized approaches to target identification using unique data sources or AI methodologies.

## 4.3 Verifying Compaction in the Session History

The conversation above looks normal, but the history has been changed behind the scenes. How can we prove it?

We can inspect the events list from our session. The compaction process **doesn't delete old events; it replaces them with a single, new Event that contains the summary**. Let's find it.

```
# Get the final session state
final_session = await session_service.get_session(
    app_name=research_runner_compacting.app_name,
    user_id=USER_ID,
    session_id="compaction_demo",
)

print("--- Searching for Compaction Summary Event ---")
found_summary = False
for event in final_session.events:
    # Compaction events have a 'compaction' attribute
    if event.actions and event.actions.compaction:
```

```

        print("\n✓ SUCCESS! Found the Compaction Event:")
        print(f" Author: {event.author}")
        print(f"\n Compacted information: {event}")
        found_summary = True
        break

if not found_summary:
    print(
        "\n✗ No compaction event found. Try increasing the number of
turns in the demo."
)

```

--- Searching for Compaction Summary Event ---

```

✓ SUCCESS! Found the Compaction Event:
Author: user

Compacted information: model_version=None content=None
grounding_metadata=None partial=None turn_complete=None finish_reason=None
error_code=None error_message=None interrupted=None custom_metadata=None
usage_metadata=None live_session_resumption_update=None
input_transcription=None output_transcription=None avg_logprobs=None
logprobs_result=None cache_metadata=None citation_metadata=None
invocation_id='54c00559-2a46-4b97-b842-5b35bfc151a2' author='user'
actions=EventActions(skip_summarization=None, state_delta={}, artifact_delta={}, transfer_to_agent=None, escalate=None, requested_auth_configs={}, requested_tool_confirmations={}, compaction={'start_timestamp': 1762841081.755067, 'end_timestamp': 1762841147.291387, 'compacted_content': {'parts': [{}{'function_call': None, 'code_execution_result': None, 'executable_code': None, 'file_data': None, 'function_response': None, 'inline_data': None, 'text': 'This conversation history covers a user\'s inquiries about AI in healthcare, specifically focusing on recent developments in drug discovery.\n\n**Key Information & Decisions:**\n\n* **Initial Request:** The user asked for the "latest news about AI in healthcare."*\n* **AI\'s Response:** The AI provided a comprehensive overview of AI advancements in healthcare, categorizing them into:\n    * Drug Discovery and Development\n    * Diagnostics and Imaging\n    * Personalized Treatment Plans and Predictive Analytics\n    * Robotics and Surgery\n    * Administrative and Operational'}}}}
```

Efficiency\n \* Mental Health\n\* \*\*Emerging Trends & Challenges:\*\*  
The AI also highlighted key trends like Explainable AI (XAI), data privacy, regulatory hurdles, integration, and ethical considerations.\n\* \*\*User's Follow-up:\*\* The user then specifically requested more information on "new developments in drug discovery."\n\* \*\*AI's Detailed Response on Drug Discovery:\*\* The AI elaborated on advancements in drug discovery, including:\n \* Generative AI for novel molecule design\n \* Accelerated target identification and validation\n \* Predicting drug efficacy and toxicity\n \* Repurposing existing drugs\n \* Optimizing clinical trial design\n \* AI for antibody and protein design\n \* Mentioned key companies and trends in this area.\n\* \*\*User's Specific Inquiry:\*\* The user then asked for more details about the "second development" mentioned in the drug discovery summary.\n\* \*\*AI's Deep Dive into Target Identification:\*\* The AI explained "Accelerated Target Identification and Validation" in detail, contrasting traditional methods with how AI revolutionizes the process through:\n \* Unprecedented data integration (multi-omics, RWE, literature mining)\n \* Identifying novel disease drivers and hidden relationships\n \* Predicting "druggability" (binding affinity, modulation potential, off-target effects)\n \* Accelerating hypothesis generation and guiding experiments.\n\*\*Unresolved Questions or Tasks:\*\*\n\* None.  
The conversation ended with the AI providing a detailed explanation of a specific AI development in drug discovery, fulfilling the user's last request. The user did not pose any further questions or assign any new tasks.', 'thought': None, 'thought\_signature': None, 'video\_metadata': None}], 'role': 'model'}}}, end\_of\_agent=None, agent\_state=None, rewind\_before\_invocation\_id=None) long\_running\_tool\_ids=set() branch=None id='bd8588fa-0e01-4de1-aef3-61d9263f8101' timestamp=1762841154.120688

#### 4.4 What you've accomplished: Automatic Context Management

You just found the proof! The presence of that special summary Event in your session's history is the tangible result of the compaction process.

**Let's recap what you just witnessed:**

1. **Silent Operation:** You ran a standard conversation, and from the outside, nothing seemed different.
2. **Background Compaction:** Because you configured the App with `EventsCompactionConfig`, the ADK Runner automatically monitored the conversation length. Once the threshold was met, it triggered the summarization process in the background.
3. **Verified Result:** By inspecting the session's events, you found the summary that the LLM generated. This summary now replaces the older, more verbose turns in the agent's active context.

**For all future turns in this conversation, the agent will be given this concise summary instead of the full history.** This saves costs, improves performance, and helps the agent stay focused on what's most important.

## 4.5 More Context Engineering options in ADK

### 👉 Custom Compaction

In this example, we used ADK's default summarizer. For more advanced use cases, you can provide your own by defining a custom `SlidingWindowCompactor` and passing it to the config. This allows you to control the summarization prompt or even use a different, specialized LLM for the task. You can read more about it in the [official documentation](#).

### 👉 Context Caching

ADK also provides **Context Caching** to help reduce the token size of the static instructions that are fed to the LLM by caching the request data. Read more about it [here](#).

## The Problem

While we can do Context Compaction and use a database to resume a session, we face new challenges now. In some cases, **we have key information or preferences that we want to share across other sessions.**

In these scenarios, instead of sharing the entire session history, transferring information from a few key variables can improve the session experience. Let's see how to do it!

---

## 🤝 Section 5: Working with Session State

### 5.1 Creating custom tools for Session state management

Let's explore how to manually manage session state through custom tools. In this example, we'll identify a **transferable characteristic**, like a user's name and their country, and create tools to capture and save it.

#### Why This Example?

The username is a perfect example of information that:

- Is introduced once but referenced multiple times
- Should persist throughout a conversation
- Represents a user-specific characteristic that enhances personalization

Here, for demo purposes, we'll create two tools that can store and retrieve user name and country from the Session State. **Note that all tools have access to the ToolContext object.** You don't have to create separate tools for each piece of information you want to share.

```
# Define scope levels for state keys (following best practices)
USER_NAME_SCOPE_LEVELS = ("temp", "user", "app")
```

```
# This demonstrates how tools can write to session state using
tool_context.

# The 'user:' prefix indicates this is user-specific data.
def save_userinfo(
    tool_context: ToolContext, user_name: str, country: str
) -> Dict[str, Any]:
    """
    Tool to record and save user name and country in session state.

```

```

Args:
    user_name: The username to store in session state
    country: The name of the user's country
"""

# Write to session state using the 'user:' prefix for user data
tool_context.state["user:name"] = user_name
tool_context.state["user:country"] = country

return {"status": "success"}


# This demonstrates how tools can read from session state.
def retrieve userinfo(tool_context: ToolContext) -> Dict[str, Any]:
    """

Tool to retrieve user name and country from session state.
"""

    # Read from session state
    user_name = tool_context.state.get("user:name", "Username not found")
    country = tool_context.state.get("user:country", "Country not found")

    return {"status": "success", "user_name": user_name, "country": country}

print("✓ Tools created.")

```

 Tools created.

### Key Concepts:

- Tools can access `tool_context.state` to read/write session state
- Use descriptive key prefixes (`user:`, `app:`, `temp:`) for organization
- State persists across conversation turns within the same session

## 5.2 Creating an Agent with Session State Tools

Now let's create a new agent that has access to our session state management tools:

```
# Configuration
APP_NAME = "default"
USER_ID = "default"
MODEL_NAME = "gemini-2.5-flash-lite"

# Create an agent with session state tools
root_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="text_chat_bot",
    description="""A text chatbot.
Tools for managing user context:
* To record username and country when provided use `saveuserinfo` tool.
* To fetch username and country when required use `retrieveuserinfo` tool.
""",
    tools=[saveuserinfo, retrieveuserinfo], # Provide the tools to the agent
)

# Set up session service and runner
session_service = InMemorySessionService()
runner = Runner(agent=root_agent, session_service=session_service,
app_name="default")

print("✅ Agent with session state tools initialized!")
```

✅ Agent with session state tools initialized!

## 5.3 Testing Session State in Action

Let's test how the agent uses session state to remember information across conversation turns:

```
# Test conversation demonstrating session state
await run_session(
    runner,
    [
        "Hi there, how are you doing today? What is my name?", # Agent
        shouldn't know the name yet
        "My name is Sam. I'm from Poland.", # Provide name - agent should
        save it
        "What is my name? Which country am I from?", # Agent should
        recall from session state
    ],
    "state-demo-session",
)
```

```
### Session: state-demo-session
```

```
User > Hi there, how are you doing today? What is my name?
gemini-2.5-flash-lite > Hello! I'm doing great. I can't tell you your
name just yet, as I haven't been told what it is. Can you please tell me
your name and your country? That way, I can remember it for you.
```

```
User > My name is Sam. I'm from Poland.
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
```

```
gemini-2.5-flash-lite > It's nice to meet you, Sam! I've saved that
you're from Poland. What else can I help you with today?
```

```
User > What is my name? Which country am I from?
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
```

```
gemini-2.5-flash-lite > Your name is Sam and you are from Poland.
```

## 5.4 Inspecting Session State

Let's directly inspect the session state to see what's stored:

```
# Retrieve the session and inspect its state
session = await session_service.get_session(
    app_name=APP_NAME, user_id=USER_ID, session_id="state-demo-session"
)

print("Session State Contents:")
print(session.state)
print("\n🔍 Notice the 'user:name' and 'user:country' keys storing our
data!")
```

```
Session State Contents:
{'user:name': 'Sam', 'user:country': 'Poland'}
```

🔍 Notice the 'user:name' and 'user:country' keys storing our data!

## 5.5 Session State Isolation

As we've already seen, an important characteristic of session state is that it's isolated per session.

Let's demonstrate this by starting a new session:

```
# Start a completely new session - the agent won't know our name
await run_session()
```

```
runner,  
    ["Hi there, how are you doing today? What is my name?"],  
    "new-isolated-session",  
)  
  
# Expected: The agent won't know the name because this is a different  
session
```

```
### Session: new-isolated-session
```

```
User > Hi there, how are you doing today? What is my name?  
gemini-2.5-flash-lite > Hello! I'm doing great, thank you for asking. I  
don't have access to your name yet. If you'd like to tell me, I can  
remember it for you!
```

## 5.6 Cross-Session State Sharing

While sessions are isolated by default, you might notice something interesting. Let's check the state of our new session (new-isolated-session):

```
# Check the state of the new session  
session = await session_service.get_session(  
    app_name=APP_NAME, user_id=USER_ID, session_id="new-isolated-session"  
)  
  
print("New Session State:")  
print(session.state)  
  
# Note: Depending on implementation, you might see shared state here.  
# This is where the distinction between session-specific and user-specific  
state becomes important.
```

```
New Session State:
```

```
{'user:name': 'Sam', 'user:country': 'Poland'}
```

---



## Cleanup

```
# Clean up any existing database to start fresh (if Notebook is restarted)
import os

if os.path.exists("my_agent_data.db"):
    os.remove("my_agent_data.db")
print("✅ Cleaned up old database files")
```



✅ Cleaned up old database files

---



## Summary



Congratulations! You've learned the fundamentals of building stateful AI agents:

- **✅ Context Engineering** - You understand how to assemble context for LLMs using Context Compaction
  - **✅ Sessions & Events** - You can maintain conversation history across multiple turns
  - **✅ Persistent Storage** - You know how to make conversations survive restarts
  - **✅ Session State** - You can track structured data during conversations
  - **✅ Manual State Management** - You've experienced both the power and limitations of manual approaches
  - **✅ Production Considerations** - You're ready to handle real-world challenges
-

 Congratulations! You did it 

 Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

## Learn More

Refer to the following documentation to learn more:

- [ADK Documentation](#)
- [ADK Sessions](#)
- [ADK Session-State](#)
- [ADK Session Compaction](#)

## Next Steps - Long Term Memory Systems (Part 2)

Why do we need memory?

In this notebook, we manually identified a couple characteristic (username and country) and built tools to manage it. But real conversations involve hundreds of such characteristics:

- User preferences and habits
- Past interactions and their outcomes
- Domain knowledge and expertise levels
- Communication styles and patterns
- Contextual relationships between topics

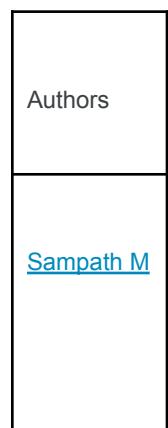
The **Memory System in ADK automates this entire process**, making it a valuable asset for building truly Context-Aware Agents that can accommodate any user's current and future needs.

In the next notebook (Part 2: Memory Management), you'll learn how to:

- Enable automatic memory extraction from conversations
- Build agents that learn and adapt over time
- Create truly personalized experiences at scale
- Manage long-term knowledge across sessions

Ready to transform your manual state management into an intelligent, automated Memory system?

Let's continue to Part 2!



*Copyright 2025 Google LLC.*

```
# @title 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
#
# https://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.
```

## Memory Management - Part 2 - Memory

Welcome to Day 3 of the Kaggle 5-day Agents course!

In the previous notebook, you learned how **Sessions** manage conversation threads. Now you'll add **Memory** - a searchable, long-term knowledge store that persists across multiple conversations.

## What is Memory ?

Memory is a service that provides long-term knowledge storage for your agents. The key distinction:

**Session = Short-term memory** (single conversation)

**Memory = Long-term knowledge** (across multiple conversations)

Think of it in software engineering terms: **Session** is like application state (temporary), while **Memory** is like a database (persistent).

## 🤔 Why Memory?

Memory provides capabilities that Sessions alone cannot:

Capability	What It Means	Example
<b>Cross-Conversation Recall</b>	Access information from any past conversation	"What preferences has this user mentioned across all chats?"
<b>Intelligent Extraction</b>	LLM-powered consolidation extracts key facts	Stores "allergic to peanuts" instead of 50 raw messages
<b>Semantic Search</b>	Meaning-based retrieval, not just keyword matching	Query "preferred hue" matches "favorite color is blue"

Persistent Storage	Survives application restarts	Build knowledge that grows over time
--------------------	-------------------------------	--------------------------------------

**Example:** Imagine talking to a personal assistant:

-  **Session:** They remember what you said 10 minutes ago in THIS conversation
-  **Memory:** They remember your preferences from conversations LAST WEEK

### What you'll learn:

-  Initialize MemoryService and integrate with your agent
-  Transfer session data to memory storage
-  Search and retrieve memories
-  Automate memory storage and retrieval
-  Understand memory consolidation (conceptual overview)

### Implementation Note

This notebook uses InMemoryMemoryService for learning - it performs keyword matching and doesn't persist data.

For production applications, use **Vertex AI Memory Bank** (covered in Day 5), which provides LLM-powered consolidation and semantic search with persistent cloud storage.

## !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**"Waiting for the next available notebook"**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

---

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

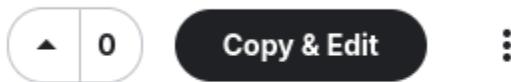
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the **Copy** and **Edit** button in the top-right corner.

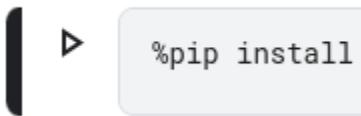


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶ Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

#### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

---

## ⚙️ Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires authentication.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

## 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

## 3. Authenticate in the notebook

Run the cell below to complete authentication.

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Gemini API key setup complete.")
except Exception as e:
    print(
        f"⚠️ Authentication Error: Please make sure you have added
'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

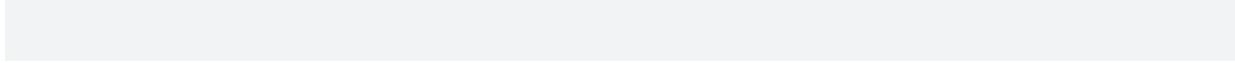
 Gemini API key setup complete.

### 1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.memory import InMemoryMemoryService
from google.adk.tools import load_memory, preload_memory
from google.genai import types

print("✅ ADK components imported successfully.")
```



✅ ADK components imported successfully.

## 1.4: Helper functions

This helper function manages a complete conversation session, handling session creation/retrieval, query processing, and response streaming.

```
async def run_session(
    runner_instance: Runner, user_queries: list[str] | str, session_id: str = "default"
):
    """Helper function to run queries in a session and display responses."""
    print(f"\n### Session: {session_id}")

    # Create or retrieve session
    try:
        session = await session_service.create_session(
            app_name=APP_NAME, user_id=USER_ID, session_id=session_id
        )
    except:
```

```

        session = await session_service.get_session(
            app_name=APP_NAME, user_id=USER_ID, session_id=session_id
        )

    # Convert single query to list
    if isinstance(user_queries, str):
        user_queries = [user_queries]

    # Process each query
    for query in user_queries:
        print(f"\nUser > {query}")
        query_content = types.Content(role="user",
parts=[types.Part(text=query)])

        # Stream agent response
        async for event in runner_instance.run_async(
            user_id=USER_ID, session_id=session.id,
new_message=query_content
        ):
            if event.is_final_response() and event.content and
event.content.parts:
                text = event.content.parts[0].text
                if text and text != "None":
                    print(f"Model: > {text}")

print("✓ Helper functions defined.")

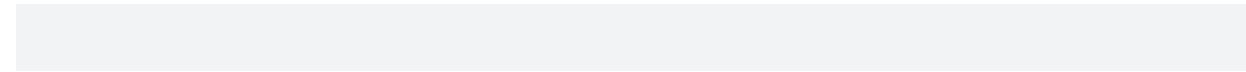
```

 Helper functions defined.

## 1.5: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config = types.HttpRetryOptions(  
    attempts=5, # Maximum retry attempts  
    exp_base=7, # Delay multiplier  
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```



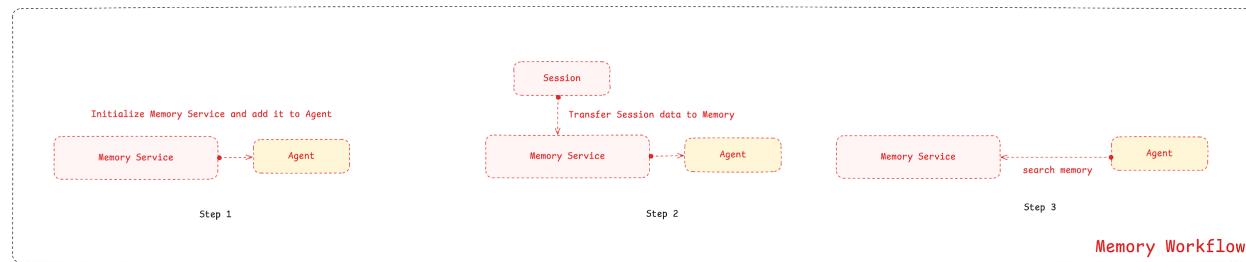
## 🧠 Section 2: Memory Workflow

From the Introduction section, you now know why we need Memory. In order to integrate Memory into your Agents, there are **three high-level steps**.

### Three-step integration process:

1. **Initialize** → Create a `MemoryService` and provide it to your agent via the `Runner`
2. **Ingest** → Transfer session data to memory using `add_session_to_memory()`
3. **Retrieve** → Search stored memories using `search_memory()`

Let's explore each step in the following sections.



## 🧠 Section 3: Initialize MemoryService

### 3.1 Initialize Memory

ADK provides multiple `MemoryService` implementations through the `BaseMemoryService` interface:

- **InMemoryMemoryService** - Built-in service for prototyping and testing (keyword matching, no persistence)
- **VertexAiMemoryBankService** - Managed cloud service with LLM-powered consolidation and semantic search
- **Custom implementations** - You can build your own using databases, though managed services are recommended

For this notebook, we'll use `InMemoryMemoryService` to learn the core mechanics. The same methods work identically with production-ready services like Vertex AI Memory Bank.

```
memory_service = (
    InMemoryMemoryService()
) # ADK's built-in Memory Service for development and testing
```

### 3.2 Add Memory to Agent

Next, create a simple agent to answer user queries.

```
# Define constants used throughout the notebook
APP_NAME = "MemoryDemoApp"
USER_ID = "demo_user"

# Create agent
user_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    name="MemoryDemoAgent",
    instruction="Answer user questions in simple words.",
)

print("✅ Agent created")
```

 Agent created

## Create Runner

Now provide both Session and Memory services to the Runner.

### Key configuration:

The Runner requires both services to enable memory functionality:

- **session\_service** → Manages conversation threads and events
- **memory\_service** → Provides long-term knowledge storage

Both services work together: Sessions capture conversations, Memory stores knowledge for retrieval across sessions.

```
# Create Session Service
session_service = InMemorySessionService() # Handles conversations

# Create runner with BOTH services
runner = Runner(
    agent=user_agent,
    app_name="MemoryDemoApp",
    session_service=session_service,
    memory_service=memory_service, # Memory service is now available!
)

print("✅ Agent and Runner created with memory support!")
```

 Agent and Runner created with memory support!

## !! Important

 **Configuration vs. Usage:** Adding `memory_service` to the Runner makes memory *available* to your agent, but doesn't automatically use it. You must explicitly:

1. **Ingest data** using `add_session_to_memory()`
2. **Enable retrieval** by giving your agent memory tools (`load_memory` or `preload_memory`)

Let's learn these steps next!

### 3.3 MemoryService Implementation Options

#### This notebook: InMemoryMemoryService

- Stores raw conversation events without consolidation
- Keyword-based search (simple word matching)
- In-memory storage (resets on restart)
- Ideal for learning and local development

#### Production: VertexAiMemoryBankService (You'll learn this on Day 5)

- LLM-powered extraction of key facts
- Semantic search (meaning-based retrieval)
- Persistent cloud storage
- Integrates external knowledge sources

 **API Consistency:** Both implementations use identical methods (`add_session_to_memory()`, `search_memory()`). The workflow you learn here applies to all memory services!

---

## Section 4: Ingest Session Data into Memory

### Why should you transfer Session data to Memory?

Now that memory is initialized, you need to populate it with knowledge. When you initialize a MemoryService, it starts completely empty. All your conversations are stored in Sessions, which contain raw events including every message, tool call, and metadata. To make this information available for long-term recall, you explicitly transfer it to memory using `add_session_to_memory()`.

Here's where managed memory services like Vertex AI Memory Bank shine. **During transfer, they perform intelligent consolidation - extracting key facts while discarding conversational noise.** The InMemoryMemoryService we're using stores everything without consolidation, which is sufficient for learning the mechanics.

Before we can transfer anything, we need data. Let's have a conversation with our agent to populate the session. This conversation will be stored in the SessionService just like you learned in the previous notebook.

```
# User tells agent about their favorite color
await run_session(
    runner,
    "My favorite color is blue-green. Can you write a Haiku about it?",
    "conversation-01", # Session ID
)
```

```
### Session: conversation-01
```

```
User > My favorite color is blue-green. Can you write a Haiku about it?
Model: > A tranquil blend,
Ocean's calm and nature's green,
Peace in every hue.
```

Let's verify the conversation was captured in the session. You should see the session events containing both the user's prompt and the model's response.

```
session = await session_service.get_session()
```

```
    app_name=APP_NAME, user_id=USER_ID, session_id="conversation-01"
)

# Let's see what's in the session
print("📝 Session contains:")
for event in session.events:
    text = (
        event.content.parts[0].text[:60]
        if event.content and event.content.parts
        else "(empty)"
    )
    print(f"  {event.content.role}: {text}...")
```

```
📝 Session contains:
user: My favorite color is blue-green. Can you write a Haiku about...
model: A tranquil blend,
Ocean's calm and nature's green,
Peace in ...
```

Perfect! The session contains our conversation. Now we're ready to transfer it to memory. Call `add_session_to_memory()` and pass the session object. This ingests the conversation into the memory store, making it available for future searches.

```
# This is the key method!
await memory_service.add_session_to_memory(session)

print("✅ Session added to memory!")
```

```
✅ Session added to memory!
```

## Section 5: Enable Memory Retrieval in Your Agent

You've successfully transferred session data to memory, but there's one crucial step remaining.

**Agents can't directly access the MemoryService - they need tools to search it.**

This is by design: it gives you control over when and how memory is retrieved.

### 5.1 Memory Retrieval in ADK

ADK provides two built-in tools for memory retrieval:

#### **load\_memory (Reactive)**

- Agent decides when to search memory
- Only retrieves when the agent thinks it's needed
- More efficient (saves tokens)
- Risk: Agent might forget to search

#### **preload\_memory (Proactive)**

- Automatically searches before every turn
- Memory always available to the agent
- Guaranteed context, but less efficient
- Searches even when not needed

Think of it like studying for an exam: `load_memory` is looking things up only when you need them, while `preload_memory` is reading all your notes before answering each question.

### 5.2 Add Load Memory Tool to Agent

Let's start by implementing the reactive pattern. We'll recreate the agent from Section 3, this time adding the `load_memory` tool to its toolkit. Since this is a built-in ADK tool, you simply include it in the `tools` array without any custom implementation.

```
# Create agent
user_agent = LlmAgent(
```

```
model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="MemoryDemoAgent",
    instruction="Answer user questions in simple words. Use load_memory
tool if you need to recall past conversations.",
    tools=[
        load_memory
    ], # Agent now has access to Memory and can search it whenever it
decides to!
)

print("✅ Agent with load_memory tool created.")
```

✅ Agent with load\_memory tool created.

### 5.3 Update the Runner and Test

Let's now update the Runner to use our new user\_agent that has the load\_memory tool. And we'll ask the Agent about the favorite color which we had stored previously in another session.

👉 Since sessions don't share conversation history, the only way the agent can answer correctly is by using the `load_memory` tool to retrieve the information from long-term memory that we manually stored.

```
# Create a new runner with the updated agent
runner = Runner(
    agent=user_agent,
    app_name=APP_NAME,
    session_service=session_service,
    memory_service=memory_service,
)

await run_session(runner, "What is my favorite color?", "color-test")
```

```
### Session: color-test

User > What is my favorite color?
```

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

## 5.4 Complete Manual Workflow Test

Let's see the complete workflow in action. We'll have a conversation about a birthday, manually save it to memory, then test retrieval in a new session. This demonstrates the full cycle: **ingest** → **store** → **retrieve**.

```
await run_session(runner, "My birthday is on March 15th.",
"birthday-session-01")
```

```
### Session: birthday-session-01
```

User > My birthday is on March 15th.  
Model: > Okay, I will remember that your birthday is on March 15th.

Now manually save this session to memory. This is the crucial step that transfers the conversation from short-term session storage to long-term memory storage.

```
# Manually save the session to memory
birthday_session = await session_service.get_session(
    app_name=APP_NAME, user_id=USER_ID, session_id="birthday-session-01"
)

await memory_service.add_session_to_memory(birthday_session)
```

```
print("✓ Birthday session saved to memory!")
```

```
✓ Birthday session saved to memory!
```

Here's the crucial test: we'll start a completely new session with a different session ID and ask the agent to recall the birthday.

```
# Test retrieval in a NEW session
await run_session(
    runner, "When is my birthday?", "birthday-session-02" # Different
session ID
)
```

```
### Session: birthday-session-02
```

User > When is my birthday?

WARNING:google\_genai.types:Warning: there are non-text parts in the response: ['function\_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.

Model: > Your birthday is on March 15th.

### What happens:

1. Agent receives: "When is my birthday?"
2. Agent recognizes: This requires past conversation context

3. Agent calls: `load_memory("birthday")`
4. Memory returns: Previous conversation containing "March 15th"
5. Agent responds: "Your birthday is on March 15th"

The memory retrieval worked even though this is a completely different session!

### Your Turn: Experiment with Both Patterns

Try swapping `load_memory` with `preload_memory` by changing the tools array to `tools=[preload_memory]`.

#### What changes:

- `load_memory` (reactive): Agent decides when to search
- `preload_memory` (proactive): Automatically loads memory before every turn

#### Test it:

1. Ask "What is my favorite color?" in a new session
2. Ask "Tell me a joke" - notice that `preload_memory` still searches memory even though it's unnecessary
3. Which pattern is better for different use cases?

## 5.5 Manual Memory Search

Beyond agent tools, you can also search memories directly in your code. This is useful for:

- Debugging memory contents
- Building analytics dashboards
- Creating custom memory management UIs

The `search_memory()` method takes a text query and returns a `SearchMemoryResponse` with matching memories.

```
# Search for color preferences
search_response = await memory_service.search_memory(
    app_name=APP_NAME, user_id=USER_ID, query="What is the user's favorite
color?"
```

```

)
print("🔍 Search Results:")
print(f"  Found {len(search_response.memories)} relevant memories")
print()

for memory in search_response.memories:
    if memory.content and memory.content.parts:
        text = memory.content.parts[0].text[:80]
        print(f"  [{memory.author}]: {text}...")

```

🔍 Search Results:  
Found 4 relevant memories

[user]: My favorite color is blue-green. Can you write a Haiku about it?...  
 [MemoryDemoAgent]: A tranquil blend,  
 Ocean's calm and nature's green,  
 Peace in every hue....  
 [user]: My birthday is on March 15th....  
 [MemoryDemoAgent]: Okay, I will remember that your birthday is on March 15th....

### 🚀 Your Turn: Test Different Queries

Try these searches to understand how keyword matching works with InMemoryMemoryService:

1. "what color does the user like"
2. "haiku"
3. "age"
4. "preferred hue"

Notice which queries return results and which don't. What pattern do you observe?

 **Key Insight:** Memory search is grounded in reality - agents can't hallucinate memories that don't exist.

## 5.6 How Search Works

**InMemoryMemoryService (this notebook):**

- **Method:** Keyword matching
- **Example:** "favorite color" matches because those exact words exist
- **Limitation:** "preferred hue" won't match

**VertexAiMemoryBankService (Day 5):**

- **Method:** Semantic search via embeddings
- **Example:** "preferred hue" WILL match "favorite color"
- **Advantage:** Understands meaning, not just keywords

You'll explore semantic search in Day 5!

---

## Section 6: Automating Memory Storage

So far, we've **manually** called `add_session_to_memory()` to transfer data to long-term storage. Production systems need this to happen **automatically**.

### 6.1 Callbacks

ADK's callback system lets you hook into key execution moments. Callbacks are **Python functions** you define and attach to agents - ADK automatically calls them at specific stages, acting like checkpoints during the agent's execution flow.

**Think of callbacks as event listeners in your agent's lifecycle.** When an agent processes a request, it goes through multiple stages: receiving the input, calling the LLM, invoking tools, and generating the response. Callbacks let you insert custom logic at each of these stages without modifying the core agent code.

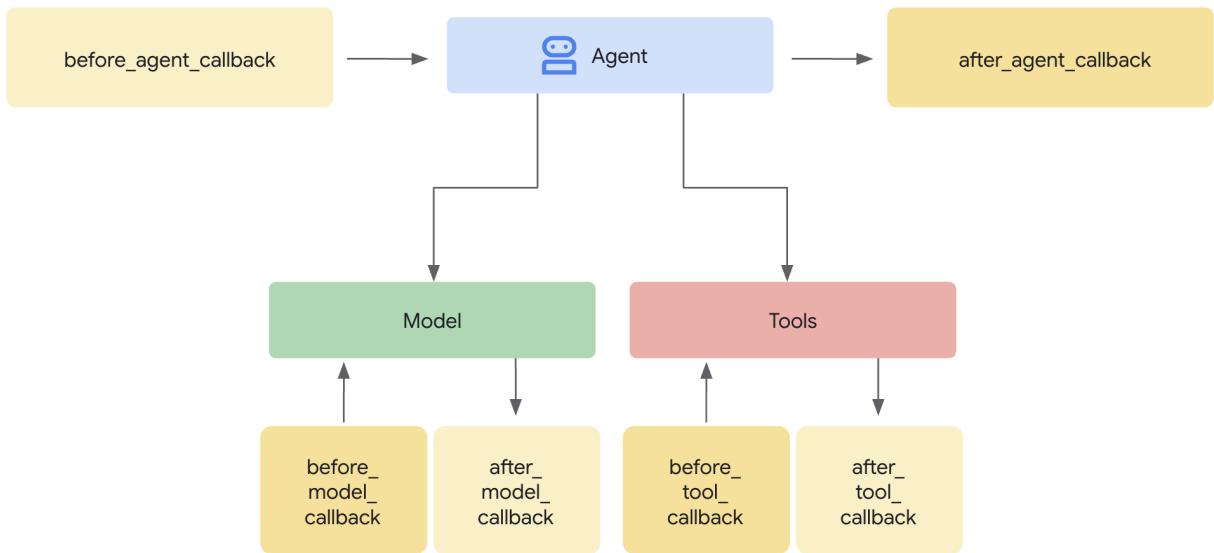
**Available callback types:**

- `before_agent_callback` → Runs before agent starts processing a request
- `after_agent_callback` → Runs after agent completes its turn
- `before_tool_callback / after_tool_callback` → Around tool invocations
- `before_model_callback / after_model_callback` → Around LLM calls
- `on_model_error_callback` → When errors occur

#### Common use cases:

- Logging and observability (track what the agent does)
- Automatic data persistence (like saving to memory)
- Custom validation or filtering
- Performance monitoring

 [Learn More: ADK Callbacks Documentation](#)



## 6.2 Automatic Memory Storage with Callbacks

For automatic memory storage, we'll use `after_agent_callback`. This function triggers every time the agent finishes a turn, then calls `add_session_to_memory()` to persist the conversation automatically.

But here's the challenge: how does our callback function actually access the memory service and current session? That's where `callback_context` comes in.

When you define a callback function, ADK automatically passes a special parameter called `callback_context` to it. The `callback_context` provides access to the Memory Service and other runtime components.

**How we'll use it:** In our callback, we'll access the memory service and current session to automatically save conversation data after each turn.

 **Important:** You don't create this context - ADK creates it and passes it to your callback automatically when the callback runs.

```
async def auto_save_to_memory(callback_context):
    """Automatically save session to memory after each agent turn."""
    await
    callback_context._invocation_context.memory_service.add_session_to_memory(
        callback_context._invocation_context.session
    )

    print("✅ Callback created.")
```

 Callback created.

## 6.3 Create an Agent: Callback and PreLoad Memory Tool

Now create an agent that combines:

- **Automatic storage:** `after_agent_callback` saves conversations
- **Automatic retrieval:** `preload_memory` loads memories

This creates a fully automated memory system with zero manual intervention.

```
# Agent with automatic memory saving
auto_memory_agent = LlmAgent(
```

```
model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
name="AutoMemoryAgent",
instruction="Answer user questions.",
tools=[preload_memory],
after_agent_callback=auto_save_to_memory, # Saves after each turn!
)

print("✅ Agent created with automatic memory saving!")
```

✅ Agent created with automatic memory saving!

### What happens automatically:

- After every agent response → callback triggers
- Session data → transferred to memory
- No manual add\_session\_to\_memory() calls needed

The framework handles everything!

## 6.4 Create a Runner and Test The Agent

Time to test! Create a Runner with the auto-memory agent, connecting the session and memory services.

```
# Create a runner for the auto-save agent
# This connects our automated agent to the session and memory services
auto_runner = Runner(
    agent=auto_memory_agent, # Use the agent with callback +
    preload_memory
    app_name=APP_NAME,
    session_service=session_service, # Same services from Section 3
    memory_service=memory_service,
)

print("✅ Runner created.")
```

 Runner created.

```
# Test 1: Tell the agent about a gift (first conversation)
# The callback will automatically save this to memory when the turn
# completes
await run_session(
    auto_runner,
    "I gifted a new toy to my nephew on his 1st birthday!",
    "auto-save-test",
)

# Test 2: Ask about the gift in a NEW session (second conversation)
# The agent should retrieve the memory using preload_memory and answer
# correctly
await run_session(
    auto_runner,
    "What did I gift my nephew?",
    "auto-save-test-2", # Different session ID - proves memory works
    across sessions!
)
```

### Session: auto-save-test

User > I gifted a new toy to my nephew on his 1st birthday!  
Model: > That's wonderful! A 1st birthday is such a special milestone. I  
hope your nephew enjoys his new toy!

### Session: auto-save-test-2

User > What did I gift my nephew?  
Model: > You gifted your nephew a new toy on his 1st birthday.

**What just happened:**

1. **First conversation:** Mentioned gift to nephew
  - Callback automatically saved to memory ✓
2. **Second conversation (new session):** Asked about the gift
  - preload\_memory automatically retrieved the memory ✓
  - Agent answered correctly ✓

**Zero manual memory calls!** This is automated memory management in action.

## 6.5 How often should you save Sessions to Memory?

**Options:**

Timing	Implementation	Best For
After every turn	after_agent_callback	Real-time memory updates
End of conversation	Manual call when session ends	Batch processing, reduce API calls
Periodic intervals	Timer-based background job	Long-running conversations

---

## Section 7: Memory Consolidation

### 7.1 The Limitation of Raw Storage

**What we've stored so far:**

- Every user message
- Every agent response
- Every tool call

#### The problem:

Session: 50 messages = 10,000 tokens

Memory: All 50 messages stored

Search: Returns all 50 messages → Agent must process 10,000 tokens

This doesn't scale. We need **consolidation**.

#### 7.2 What is Memory Consolidation?

**Memory Consolidation** = Extracting **only important facts** while discarding conversational noise.

#### Before (Raw Storage):

User: "My favorite color is BlueGreen. I also like purple.

Actually, I prefer BlueGreen most of the time."

Agent: "Great! I'll remember that."

User: "Thanks!"

Agent: "You're welcome!"

→ Stores ALL 4 messages (redundant, verbose)

#### After (Consolidation):

Extracted Memory: "User's favorite color: BlueGreen"

→ Stores 1 concise fact

**Benefits:** Less storage, faster retrieval, more accurate answers.

```

Session(id='784', app_name='example_app', user_id='2', state={},
events=[Event(content=Content(
    parts=[
        Part(
            text='My favorite project is Project Alpha.'
        ),
        Part(
            role='user'
        )
    ],
    grounding_metadata=None, partial=None, turn_complete=None,
    error_code=None, error_message=None, interrupted=None,
    custom_metadata=None, usage_metadata=None, invocation_id='e-99dd373a-ea26-
4d8c-a45d-0ed5d0d03f01', author='InfoCaptureAgent'
), actions=EventActions(skip_summarization=None, state_delays={}, artifact_delta=0, transfer_to_agent=None, escalate=None,
requested_auth_desires={}, long_running_tool_ids=None, branch=None,
id='8cf771463-426b-0000-332cd622e7f1', timestamp='1726994914.926269'],
Event(content=Content(
    parts=[
        Part(
            text='Okay, I understand. Your favorite project is Project Alpha.
        ),
        Part(
            role='model'
        )
    ],
    grounding_metadata=None, partial=None, turn_complete=None,
    error_code=None, error_message=None, interrupted=None,
    custom_metadata=None, usage_metadata=GenerateContentResponseUsageMetadata(
        candidates_token_count=13,
        candidates_tokens_details=[{
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        }
    ],
    prompt_token_count=29,
    prompt_token_details=[{
        'ModalityTokenCount': {
            'Modality': 'Text',
            'Text': 'Okay, I understand. Your favorite project is Project Alpha.
        },
        'token_count': 29
    },
    total_token_count=42
    traffic_type=TrafficType.ON_DEMAND, 'ON_DEMAND' >
), actions=EventActions(skip_summarization=None,
state_delays={}, artifact_delta=0, transfer_to_agent=None, escalate=None,
requested_auth_desires={}, long_running_tool_ids=None, branch=None,
id='5807dec1-9ab-403b-9209091bd1', timestamp='1726994914.926806),
lost_updates_time='1726994914.926806'],
memories=[MemoryEntry(content=Content(
    parts=[
        Part(
            text='My favorite project is Project Alpha.'
        ),
        Part(
            role='user'
        )
    ],
    author='user', timestamp='2025-07-15T15:55:14.926269'), MemoryEntry(content=Content(
    parts=[
        Part(
            text='Okay, I understand. Your favorite project is Project Alpha.
        ),
        Part(
            role='model'
        )
    ],
    author='InfoCaptureAgent', timestamp='2025-07-15T15:55:14.926806')]
)

```

Sample Session data      Sample Memory

## 7.3 How Consolidation Works (Conceptual)

**The pipeline:**

1. Raw Session Events
- ↓
2. LLM analyzes conversation
- ↓
3. Extracts key facts
- ↓
4. Stores concise memories
- ↓
5. Merges with existing memories (deduplication)

**Example transformation:**

**Input:** "I'm allergic to peanuts. I can't eat anything with nuts."

**Output:** Memory {  
 allergy: "peanuts, tree nuts"  
 severity: "avoid completely"  
}

Natural language → Structured, actionable data.

## 7.4 Next Steps for Memory Consolidation

 **Key Point:** Managed Memory Services handle consolidation **automatically**.

**You use the same API:**

- `add_session_to_memory()` ← Same method
- `search_memory()` ← Same method

**The difference:** What happens behind the scenes.

- **InMemoryMemoryService:** Stores raw events
- **VertexAiMemoryBankService:** Intelligently consolidates before storing

 **Learn More:**

- [Vertex AI Memory Bank: Memory Consolidation Guide](#) -> You'll explore this in Day 5!
- 

## Summary

You've learned the **core mechanics** of Memory in ADK:

1.  **Adding Memory**
  - Initialize MemoryService alongside SessionService
  - Both services are provided to the Runner
2.  **Storing Information**
  - `await memory_service.add_session_to_memory(session)`
  - Transfers session data to long-term storage
  - Can be automated with callbacks
3.  **Searching Memory**
  - `await memory_service.search_memory(app_name, user_id, query)`
  - Returns relevant memories from past conversations
4.  **Retrieving in Agents**

- **Reactive:** load\_memory tool (agent decides when to use memory)
  - **Proactive:** preload\_memory tool (always loads memory into LLM's system instructions)
5.  **Memory Consolidation**
- Extracts key information from Session data
  - Provided by managed memory services such as Vertex AI Memory Bank



**Congratulations!** You've learned Memory Management in ADK!



**Learn More:**

- [ADK Memory Documentation](#)
- [Vertex AI Memory Bank](#)
- [Memory Consolidation Guide](#)



**Next Steps:**

Ready for Day 4? Learn how to **implement Observability and Evaluate your agents** to ensure they're working as intended in production!

---

## Authors

Authors
<a href="#">Sampath M</a>

Copyright 2025 Google LLC.

# @title 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  
#  
# https://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.
```



## Agent Observability - Logs, Traces & Metrics

Welcome to Day 4 of the Kaggle 5-day Agents course!

In Day 3, you learned the **"What, Why & How"** of Session and Memory management, focusing on long-term, short-term, and shared memory (state).

Today, you'll learn:

- How to add observability to the agent you've built and
- How to evaluate if the agents are working as expected

In this notebook, we'll focus on the first part - **Agent Observability!**

### What is Agent Observability?



**The challenge:** Unlike traditional software that fails predictably, AI agents can fail mysteriously.

Example:

```
User: "Find quantum computing papers"  
Agent: "I cannot help with that request."
```

```
You: 🤔 WHY?? Is it the prompt? Missing tools? API error?
```

 **The Solution:** Agent observability gives you complete visibility into your agent's decision-making process. You'll see exactly what prompts are sent to the LLM, which tools are available, how the model responds, and where failures occur.

```
DEBUG Log: LLM Request shows "Functions: []" (no tools!)
```

```
You: 🎯 Aha! Missing google_search tool - easy fix!
```

## Foundational pillars of Agent Observability

1. **Logs:** A log is a record of a single event, telling you **what** happened at a specific moment.
2. **Traces:** A trace connects the logs into a single story, showing you **why** a final result occurred by revealing the entire sequence of steps.
3. **Metrics:** Metrics are the summary numbers (like averages and error rates) that tell you **how** well the agent is performing overall.

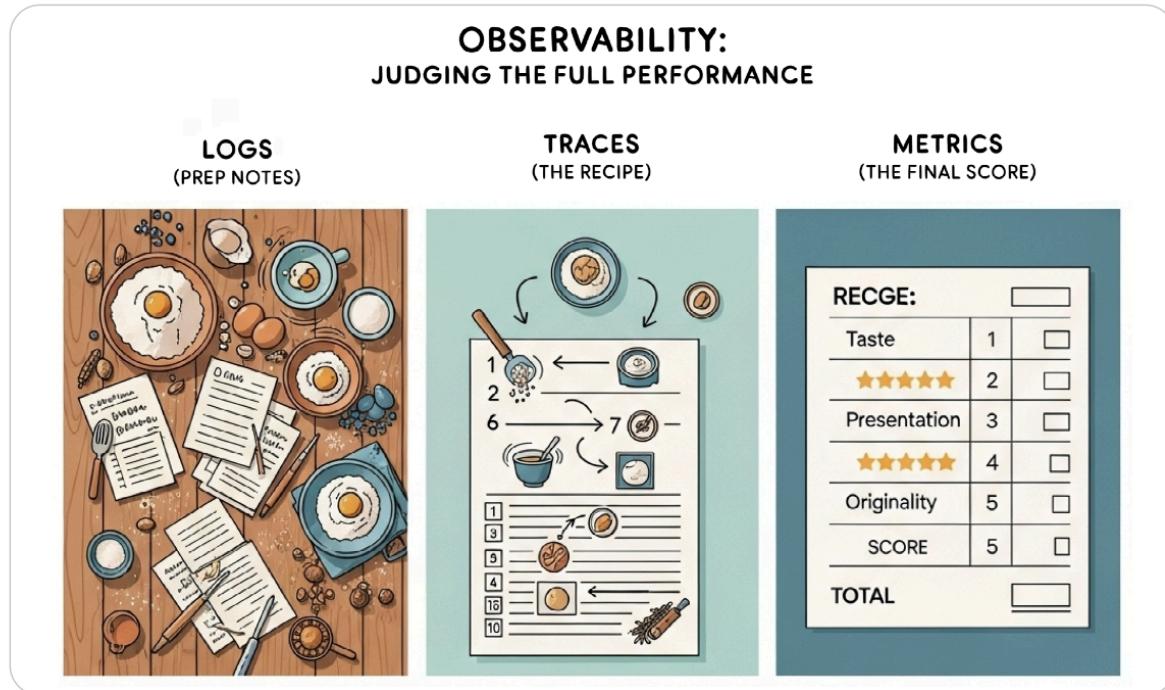


Figure 4: Three foundational pillars for Agent Observability

In this notebook, you'll:

- Set up logging configuration
- Create a broken agent. Use adk web UI & logs to identify exactly why the agent fails
- Understand how to implement logging in production
- Learn when to use built-in logging vs custom solutions

## !! Please Read

  **Note:** No submission required! This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. [See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Section 1: Setup

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires an API key.

## 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

## 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

## 3. Authenticate in the notebook

Run the cell below to access the GOOGLE\_API\_KEY you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Setup and authentication complete.")
except Exception as e:
    print(
        f"⚠️ Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

✅ Setup and authentication complete.

### 1.3: Set up logging and cleanup old files

Let's configure logging for our debugging session. The following cell makes sure we also capture other log levels, like DEBUG.

```
import logging
import os

# Clean up any previous logs
for log_file in ["logger.log", "web.log", "tunnel.log"]:
    if os.path.exists(log_file):
        os.remove(log_file)
        print(f"🧹 Cleaned up {log_file}")

# Configure logging with DEBUG log level.
logging.basicConfig(
    filename="logger.log",
    level=logging.DEBUG,
    format="%(filename)s:%(lineno)s %(levelname)s:%(message)s",
)

print("✅ Logging configured")
```

```
✅ Logging configured
```

### 1.4: Set up proxy and tunneling

We'll use a proxy to access the ADK web UI from within the Kaggle Notebooks environment. If you are running this outside the Kaggle environment, you don't need to do this.

```
from IPython.core.display import display, HTML
from jupyter_server.serverapp import list_running_servers
```

```
# Gets the proxied URL in the Kaggle Notebooks environment
def get_adk_proxy_url():
```

```

PROXY_HOST = "https://kkb-production.jupyter-proxy.kaggle.net"
ADK_PORT = "8000"

servers = list(list_running_servers())
if not servers:
    raise Exception("No running Jupyter servers found.")

baseURL = servers[0]["base_url"]

try:
    path_parts = baseURL.split("/")
    kernel = path_parts[2]
    token = path_parts[3]
except IndexError:
    raise Exception(f"Could not parse kernel/token from base URL: {baseURL}")

url_prefix = f"/k/{kernel}/{token}/proxy/proxy/{ADK_PORT}"
url = f"{PROXY_HOST}{url_prefix}"

styled_html = """
<div style="padding: 15px; border: 2px solid #f0ad4e; border-radius: 8px; background-color: #fef9f0; margin: 20px 0;">
    <div style="font-family: sans-serif; margin-bottom: 12px; color: #333; font-size: 1.1em;">
        <strong>⚠️ IMPORTANT: Action Required</strong>
    </div>
    <div style="font-family: sans-serif; margin-bottom: 15px; color: #333; line-height: 1.5;">
        The ADK web UI is <strong>not running yet</strong>. You must start it in the next cell.
        <ol style="margin-top: 10px; padding-left: 20px;">
            <li style="margin-bottom: 5px;"><strong>Run the next cell</strong> (the one with <code>!adk web ...</code>) to start the ADK web UI.</li>
            <li style="margin-bottom: 5px;">Wait for that cell to show it is "Running" (it will not "complete").</li>
                <li>Once it's running, <strong>return to this button</strong> and click it to open the UI.</li>
        </ol>
    </div>
</div>
"""

```

```

        <em style="font-size: 0.9em; color: #555;">(If you click the
button before running the next cell, you will get a 500 error.)</em>
    </div>
    <a href='{url}' target='_blank' style="
        display: inline-block; background-color: #1a73e8; color:
white; padding: 10px 20px;
        text-decoration: none; border-radius: 25px; font-family:
sans-serif; font-weight: 500;
        box-shadow: 0 2px 5px rgba(0,0,0,0.2); transition: all 0.2s
ease;">
        Open ADK Web UI (after running cell below) ^
    </a>
</div>
"""

display(HTML(styled_html))

return url_prefix

print("✅ Helper functions defined.")

```

✅ Helper functions defined.

---



## Section 2: Hands-On Debugging with ADK Web UI

### 2.1: Create a "Research Paper Finder" Agent

**Our goal:** Build a research paper finder agent that helps users find academic papers on any topic.

But first, let's intentionally create an incorrect version of the agent to practice debugging! We'll start by creating a new agent folder using the adk create CLI command.

```
!adk create research-agent --model gemini-2.5-flash-lite --api_key  
$GOOGLE_API_KEY
```

```
Agent created in /kaggle/working/research-agent:
```

```
- .env  
- __init__.py  
- agent.py
```

## Agent definition

Next, let's create our root agent.

- We'll configure it as an LlmAgent, give it a name, model and instruction.
- The root\_agent gets the user prompt and delegates the search to the google\_search\_agent.
- Then, the agent uses the count\_papers tool to count the number of papers returned.

 **Pay attention to the root agent's instructions and the count\_papers tool parameter!**

```
%%writefile research-agent/agent.py

from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.tools.agent_tool import AgentTool
from google.adk.tools.google_search_tool import google_search

from google.genai import types
from typing import List

retry_config = types.HttpRetryOptions(
    attempts=5, # Maximum retry attempts
    exp_base=7, # Delay multiplier
    initial_delay=1,
```

```

        http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors
    )

# ---- Intentionally pass incorrect datatype - `str` instead of `List[str]`
-----
def count_papers(papers: str):
    """
    This function counts the number of papers in a list of strings.

    Args:
        papers: A list of strings, where each string is a research paper.

    Returns:
        The number of papers in the list.
    """
    return len(papers)

# Google Search agent
google_search_agent = LlmAgent(
    name="google_search_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    description="Searches for information using Google search",
    instruction="""Use the google_search tool to find information on the
given topic. Return the raw search results.

If the user asks for a list of papers, then give them the list of
research papers you found and not the summary.""",
    tools=[google_search]
)

# Root agent
root_agent = LlmAgent(
    name="research_paper_finder_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    instruction="""Your task is to find research papers and count them.

You MUST ALWAYS follow these steps:
1) Find research papers on the user provided topic using the
'google_search_agent'.
"""
)

```

```
    2) Then, pass the papers to 'count_papers' tool to count the number of  
    papers returned.  
    3) Return both the list of research papers and the total number of  
    papers.  
    """,  
    tools=[AgentTool(agent=google_search_agent), count_papers]  
)
```

Overwriting research-agent/agent.py

## 2.2: Run the agent

Let's now run our agent with the `adk web --log_level DEBUG` CLI command.

💡 The key here is `--log_level DEBUG` - this shows us:

- **Full LLM Prompts:** The complete request sent to the language model, including system instructions, history, and tools.
- Detailed API responses from services.
- Internal state transitions and variable values.

Other log levels include: INFO, ERROR and WARNING.

Get the proxied URL to access the ADK web UI in the Kaggle Notebooks environment:

```
url_prefix = get_adk_proxy_url()
```

### ⚠️ IMPORTANT: Action Required

The ADK web UI is **not running yet**. You must start it in the next cell.

1. **Run the next cell** (the one with `!adk web . . .`) to start the ADK web UI.
2. Wait for that cell to show it is "Running" (it will not "complete").
3. Once it's running, **return to this button** and click it to open the UI.

(If you click the button before running the next cell, you will get a 500 error.)

[Open ADK Web UI \(after running cell below\) ↗](#)

Now you can start the ADK web UI with the `--log_level` parameter.

👉 **Note:** The following cell will not "complete", but will remain running and serving the ADK web UI until you manually stop the cell.

```
!adk web --log_level DEBUG --url_prefix {url_prefix}
```

```
/usr/local/lib/python3.11/dist-packages/google/adk/cli/fast_api.py:130:  
UserWarning: [EXPERIMENTAL] InMemoryCredentialService: This feature is  
experimental and may change or be removed in future versions without  
notice. It may introduce breaking changes at any time.  
    credential_service = InMemoryCredentialService()  
/usr/local/lib/python3.11/dist-packages/google/adk/auth/credential_service  
/in_memory_credential_service.py:33: UserWarning: [EXPERIMENTAL]  
BaseCredentialService: This feature is experimental and may change or be  
removed in future versions without notice. It may introduce breaking  
changes at any time.  
    super().__init__()  
INFO:     Started server process [71]  
INFO:     Waiting for application startup.
```

```
+-----+  
| ADK Web Server started  
|  
|  
|
```

```
| For local testing, access at http://127.0.0.1:8000.  
|  
+-----+  
----+  
  
INFO: Application startup complete.  
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)  
^C  
INFO: Shutting down  
INFO: Waiting for application shutdown.  
  
+-----+  
----+  
| ADK Web Server shutting down...  
|  
+-----+  
----+  
  
INFO: Application shutdown complete.  
INFO: Finished server process [71]  
  
Aborted!
```

Once the ADK web UI starts, open the proxy link using the button in the previous cell.

As you start chatting with the agent, you should see the DEBUG logs appear in the output cell below!

**!! IMPORTANT: DO NOT SHARE THE PROXY LINK** with anyone - treat it as sensitive data as it contains your authentication token in the URL.

### 2.3: Test the agent in ADK web UI

#### Do: In the ADK web UI

1. Select "research-agent" from the dropdown in the top-left.

2. In the chat interface, type: Find latest quantum computing papers
3. Send the message and observe the response. The agent should return a list of research papers and their count.

It looks like our agent works and we got a response! 🤔 **But wait, isn't the count of papers unusually large? Let's look at the logs and trace.**

#### 👉 Do: Events tab - Traces in detail

1. In the web UI, click the "**Events**" tab on the left sidebar
2. You'll see a chronological list of all agent actions
3. Click on any event to expand its details in the bottom panel
4. Try clicking the "**Trace**" button to see timing information for each step.
5. **Click the `execute_tool count_papers` span. You'll see that the function call to `count_papers` returns the large number as the response.**
6. Let's look at what was passed as input to this function.
7. **Find the `call_llm` span corresponding to the `count_papers` function call.**

#### 👉 Do: Inspect the Function call in Events:

- Click on the specific span to open the Events tab.
- Examine the `function_call`, focusing on the `papers` argument.
- Notice that `root_agent` passes the list of papers as a `str` instead of a `List[str]` - there's our bug!



#### 2.4: Your Turn - fix it! 💭

Update the datatype of the `papers` argument in the `count_papers` tool to a `List[str]` and rerun the `adk web` command!

---

**!! Stop the ADK web UI** ⚡

In order to run cells in the remainder of this notebook, please stop the running cell where you started adk web in Section 3.1.

Otherwise that running cell will block / prevent other cells from running as long as the ADK web UI is running.

---

## 2.5: Debug through local Logs

Optionally, you can also examine the local DEBUG logs to find the root cause. Run the following cell to print the contents of the log file. Look for detailed logs like:

```
DEBUG - google_adk.models.google_llm - LLM Request: ...
```

```
DEBUG - google_adk.models.google_llm - LLM Response: ...
```

```
# Check the DEBUG logs from the broken agent
print("🔍 Examining web server logs for debugging clues...\n")
!cat logger.log
```

```
🔍 Examining web server logs for debugging clues...
```

**Other Observability questions you can now answer from logs and adk web:**

- **Efficiency:** Is the agent making optimal tool choices?
- **Reasoning Quality:** Are the prompts well-structured and context-appropriate?
- **Performance:** Look at the traces to identify which steps take the longest?
- **Failure Diagnosis:** When something goes wrong, where exactly did it fail?

**Key Learning:** Core debugging pattern: symptom → logs → root cause → fix.

**Debugging Victory:** You just went from "Agent mysteriously failed" to "I know exactly why and how to fix it!" This is the power of observability!



## Section 3: Logging in production

⌚ Great! You can now debug agent failures using ADK web UI and DEBUG logs.

But what happens when you move beyond development? Real-world scenarios where you need to move beyond the web UI:

### ✗ Problem 1: Production Deployment

You: "Let me open the ADK web UI to check why the agent failed"  
DevOps: "Um... this is a production server. No web UI access."

You: 😱 "How do I debug production issues?"

### ✗ Problem 2: Automated Systems

You: "The agent runs 1000 times per day in our pipeline"  
Boss: "Which runs are slow? What's our success rate?"

You: 😰 "I'd have to manually check the web UI 1000 times..."



### The Solution:

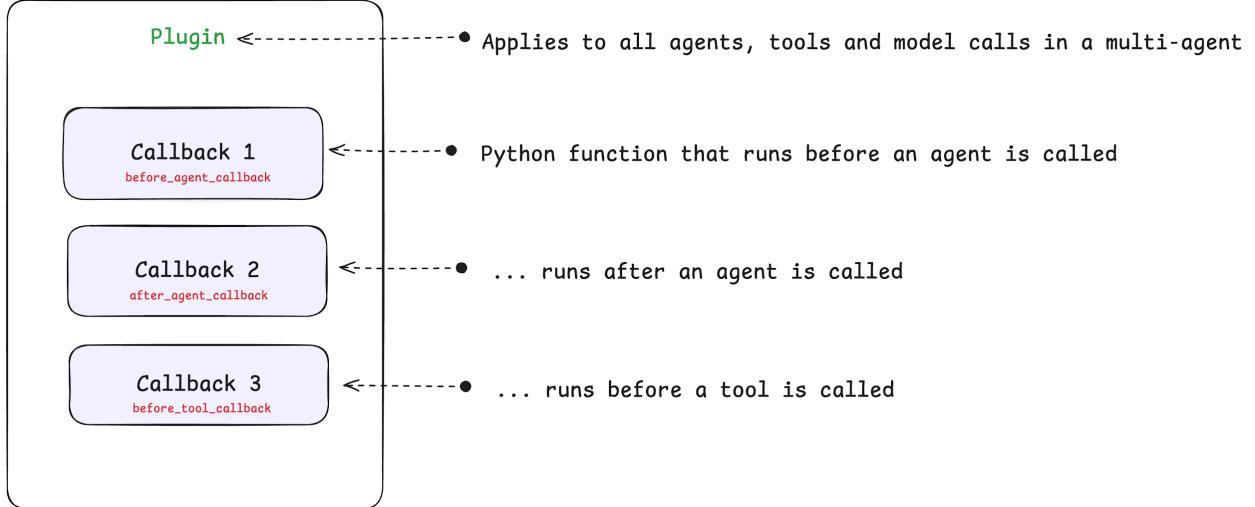
We need a way to capture observability data or in other words, **add logs to our code**.

👉 In traditional software development, this is done by adding log statements in Python functions - **and agents are no different!** We need to add log statements to our agent and a common approach is to add log statements to **Plugins**.

#### 3.1: How to add logs for production observability?

A Plugin is a custom code module that runs automatically at various stages of your agent's lifecycle. Plugins are composed of "**Callbacks**" which provide the hooks to interrupt an agent's flow. Think of it like this:

- **Your agent workflow:** User message → Agent thinks → Calls tools → Returns response
- **Plugin hooks into this:** Before agent starts → After tool runs → When LLM responds → etc.
- **Plugin contains your custom code:** Logging, monitoring, security checks, caching, etc.

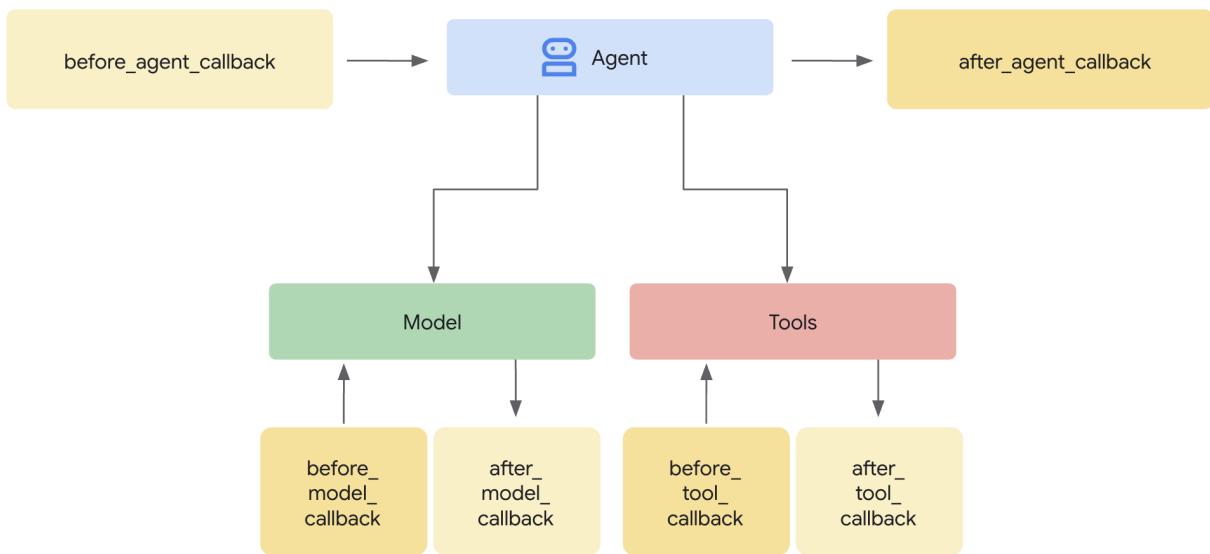


## Callbacks

Callbacks are the **atomic components inside a Plugin** - these are just Python functions that run at specific points in an agent's lifecycle! **Callbacks are grouped together to create a Plugin**.

There are different kinds of callbacks such as:

- **before/after\_agent\_callbacks** - runs before/after an agent is invoked
- **before/after\_tool\_callbacks** - runs before/after a tool is called
- **before/after\_model\_callbacks** - similarly, runs before/after the LLM model is called
- **on\_model\_error\_callback** - which runs when a model error is encountered



### 3.2: To make things more concrete, what does a Plugin look like?

```

print("----- EXAMPLE PLUGIN - DOES NOTHING ----- ")

import logging
from google.adk.agents.base_agent import BaseAgent
from google.adk.agents.callback_context import CallbackContext
from google.adk.models.llm_request import LlmRequest
from google.adk.plugins.base_plugin import BasePlugin


# Applies to all agent and model calls
class CountInvocationPlugin(BasePlugin):
    """A custom plugin that counts agent and tool invocations."""

    def __init__(self) -> None:
        """Initialize the plugin with counters."""
        super().__init__(name="count_invocation")
        self.agent_count: int = 0
        self.tool_count: int = 0
        self.llm_request_count: int = 0

    # Callback 1: Runs before an agent is called. You can add any custom logic here.
    @async def before_agent_callback(

```

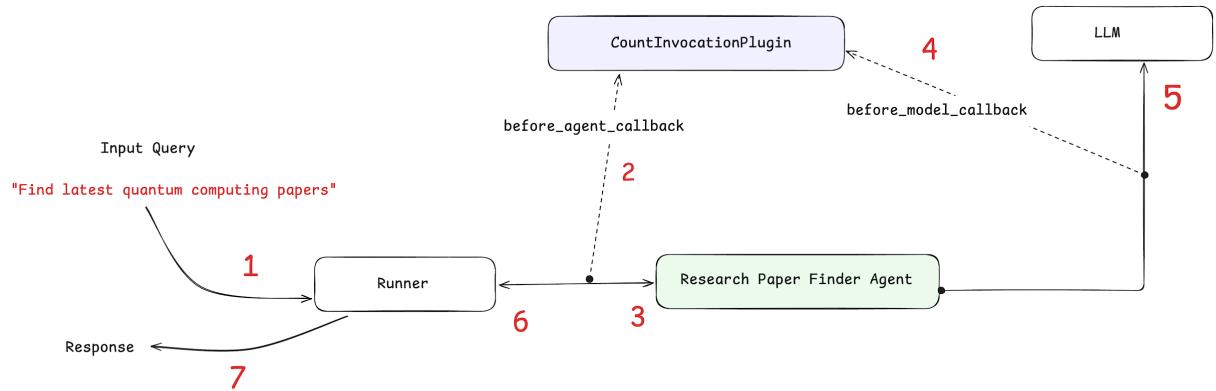
```
        self, *, agent: BaseAgent, callback_context: CallbackContext
    ) -> None:
        """Count agent runs."""
        self.agent_count += 1
        logging.info(f"[Plugin] Agent run count: {self.agent_count}")

# Callback 2: Runs before a model is called. You can add any custom
logic here.
async def before_model_callback(
    self, *, callback_context: CallbackContext, llm_request:
LlmRequest
) -> None:
    """Count LLM requests."""
    self.llm_request_count += 1
    logging.info(f"[Plugin] LLM request count:
{self.llm_request_count}")
```

----- EXAMPLE PLUGIN - DOES NOTHING -----

**Key insight:** You register a plugin **once** on your runner, and it automatically applies to **every agent, tool call, and LLM request** in your system as per your definition. Read more about Plugin hooks [here](#).

You can follow along with the numbers in the diagram below to understand the flow.



The Runner calls the `CountInvocationPlugin` (2) before calling the agent since the plugin has a `before\_agent\_callback`. Similarly, the plugin is also executed before an LLM call is made by the agent (4).

### 3.3: ADK's built-in LoggingPlugin

But you don't have to define all the callbacks and plugins to capture *standard* Observability data in ADK. Instead, ADK provides a built-in **LoggingPlugin** that automatically captures all agent activity:

- 🚀 User messages and agent responses
- ⏳ Timing data for performance analysis
- 🧠 LLM requests and responses for debugging
- 🔐 Tool calls and results
- ✅ Complete execution traces

#### Agent definition

Let's use the same agent from the previous demo - the Research paper finder!

```

from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini
from google.adk.tools.agent_tool import AgentTool
from google.adk.tools.google_search_tool import google_search

from google.genai import types
from typing import List

retry_config = types.HttpRetryOptions(
    attempts=5, # Maximum retry attempts
    exp_base=7, # Delay multiplier
    initial_delay=1,
)

```

```
        http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors
    )

def count_papers(papers: List[str]):
    """
    This function counts the number of papers in a list of strings.

    Args:
        papers: A list of strings, where each string is a research paper.

    Returns:
        The number of papers in the list.

    """
    return len(papers)

# Google search agent
google_search_agent = LlmAgent(
    name="google_search_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    description="Searches for information using Google search",
    instruction="Use the google_search tool to find information on the
given topic. Return the raw search results.",
    tools=[google_search],
)

# Root agent
research_agent_with_plugin = LlmAgent(
    name="research_paper_finder_agent",
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    instruction="""Your task is to find research papers and count them.

    You must follow these steps:
    1) Find research papers on the user provided topic using the
    'google_search_agent'.
    2) Then, pass the papers to 'count_papers' tool to count the number of
    papers returned.
    3) Return both the list of research papers and the total number of
    papers.
    """
)
```

```
    """ ,  
    tools=[AgentTool(agent=google_search_agent), count_papers],  
)  
  
print("✅ Agent created")
```

```
✅ Agent created
```

### 3.4: Add LoggingPlugin to Runner

The following code creates the InMemoryRunner. This is used to programmatically invoke the agent.

**To use LoggingPlugin in the above research agent,** 1) Import the plugin 2) Add it when initializing the InMemoryRunner.

```
from google.adk.runners import InMemoryRunner  
from google.adk.plugins.logging_plugin import (  
    LoggingPlugin,  
) # <---- 1. Import the Plugin  
from google.genai import types  
import asyncio  
  
runner = InMemoryRunner(  
    agent=research_agent_with_plugin,  
    plugins=[  
        LoggingPlugin()  
    ], # <---- 2. Add the plugin. Handles standard Observability logging  
across ALL agents  
)  
  
print("✅ Runner configured")
```

```
✅ Runner configured
```

 Runner configured

Let's now run the agent using `run_debug` function.

```
print("🚀 Running agent with LoggingPlugin...")  
print("📊 Watch the comprehensive logging output below:\n")  
  
response = await runner.run_debug("Find recent papers on quantum  
computing")
```

```
🚀 Running agent with LoggingPlugin...  
📊 Watch the comprehensive logging output below:
```

```
### Created new session: debug_session_id  
  
User > Find recent papers on quantum computing  
[logging_plugin] 🚀 USER MESSAGE RECEIVED  
[logging_plugin] Invocation ID: e-c8943591-6d63-4a49-8e91-3a56ca8764ea  
[logging_plugin] Session ID: debug_session_id  
[logging_plugin] User ID: debug_user_id  
[logging_plugin] App Name: InMemoryRunner  
[logging_plugin] Root Agent: research_paper_finder_agent  
[logging_plugin] User Content: text: 'Find recent papers on quantum  
computing'  
[logging_plugin] 🚶 INVOCATION STARTING  
[logging_plugin] Invocation ID: e-c8943591-6d63-4a49-8e91-3a56ca8764ea  
[logging_plugin] Starting Agent: research_paper_finder_agent  
[logging_plugin] 🤖 AGENT STARTING  
[logging_plugin] Agent Name: research_paper_finder_agent  
[logging_plugin] Invocation ID: e-c8943591-6d63-4a49-8e91-3a56ca8764ea  
[logging_plugin] 🧠 LLM REQUEST  
[logging_plugin] Model: gemini-2.5-flash-lite  
[logging_plugin] Agent: research_paper_finder_agent  
[logging_plugin] System Instruction: 'Your task is to find research  
papers and count them.'
```

You must follow these steps:

1) Find research papers on the user provided topic using the 'google\_search\_agent'.

2) Then, pass the p...'

```
[logging_plugin] Available Tools: ['google_search_agent',  
'count_papers']  
[logging_plugin] 🧠 LLM RESPONSE  
[logging_plugin] Agent: research_paper_finder_agent  
[logging_plugin] Content: function_call: google_search_agent  
[logging_plugin] Token Usage - Input: 242, Output: 21  
[logging_plugin] 💡 EVENT YIELDED  
[logging_plugin] Event ID: 2e6acd5a-f3d0-4a69-83e8-7903633e750c  
[logging_plugin] Author: research_paper_finder_agent  
[logging_plugin] Content: function_call: google_search_agent  
[logging_plugin] Final Response: False  
[logging_plugin] Function Calls: ['google_search_agent']  
[logging_plugin] 🔐 TOOL STARTING  
[logging_plugin] Tool Name: google_search_agent  
[logging_plugin] Agent: research_paper_finder_agent  
[logging_plugin] Function Call ID:  
adk-c7e83818-7d0a-4eb3-824a-a2edceb661eb  
[logging_plugin] Arguments: {'request': 'recent papers on quantum  
computing'}  
[logging_plugin] 🚀 USER MESSAGE RECEIVED  
[logging_plugin] Invocation ID: e-40537826-a7cc-4baa-a15a-b3bdeca10e4d  
[logging_plugin] Session ID: 66f986fc-455b-4681-8568-10d70da7d2c0  
[logging_plugin] User ID: debug_user_id  
[logging_plugin] App Name: InMemoryRunner  
[logging_plugin] Root Agent: google_search_agent  
[logging_plugin] User Content: text: 'recent papers on quantum  
computing'  
[logging_plugin] 🚶 INVOCATION STARTING  
[logging_plugin] Invocation ID: e-40537826-a7cc-4baa-a15a-b3bdeca10e4d  
[logging_plugin] Starting Agent: google_search_agent  
[logging_plugin] 🤖 AGENT STARTING  
[logging_plugin] Agent Name: google_search_agent  
[logging_plugin] Invocation ID: e-40537826-a7cc-4baa-a15a-b3bdeca10e4d  
[logging_plugin] 🧠 LLM REQUEST  
[logging_plugin] Model: gemini-2.5-flash-lite
```

```
[logging_plugin]    Agent: google_search_agent
[logging_plugin]    System Instruction: 'Use the google_search tool to
find information on the given topic. Return the raw search results.'
```

You are an agent. Your internal name is "google\_search\_agent". The description about you is "Searches..."

```
[logging_plugin] 🧠 LLM RESPONSE
[logging_plugin]    Agent: google_search_agent
[logging_plugin]    Content: text: 'The field of quantum computing is
experiencing rapid advancements, with a particular focus on stabilizing
qubits, developing specialized hardware and software, and improving error
correction technique...'
[logging_plugin]    Token Usage - Input: 58, Output: 608
[logging_plugin] 🔍 EVENT YIELDED
[logging_plugin]    Event ID: 64c47153-db33-4b18-b56b-afce2a957c51
[logging_plugin]    Author: google_search_agent
[logging_plugin]    Content: text: 'The field of quantum computing is
experiencing rapid advancements, with a particular focus on stabilizing
qubits, developing specialized hardware and software, and improving error
correction technique...'
[logging_plugin]    Final Response: True
[logging_plugin] 🤖 AGENT COMPLETED
[logging_plugin]    Agent Name: google_search_agent
[logging_plugin]    Invocation ID: e-40537826-a7cc-4baa-a15a-b3bdeca10e4d
[logging_plugin] ✅ INVOCATION COMPLETED
[logging_plugin]    Invocation ID: e-40537826-a7cc-4baa-a15a-b3bdeca10e4d
[logging_plugin]    Final Agent: google_search_agent
[logging_plugin] 🔐 TOOL COMPLETED
[logging_plugin]    Tool Name: google_search_agent
[logging_plugin]    Agent: research_paper_finder_agent
[logging_plugin]    Function Call ID:
adk-c7e83818-7d0a-4eb3-824a-a2edceb661eb
[logging_plugin]    Result: The field of quantum computing is experiencing
rapid advancements, with a particular focus on stabilizing qubits,
developing specialized hardware and software, and improving error
correction techniques. Researchers are also exploring hybrid
quantum-classical systems and the integration of quantum co...}
[logging_plugin] 🔍 EVENT YIELDED
[logging_plugin]    Event ID: add009be-b674-49ca-abd1-70867cb0c511
[logging_plugin]    Author: research_paper_finder_agent
```

```
[logging_plugin] Content: function_response: google_search_agent
[logging_plugin] Final Response: False
[logging_plugin] Function Responses: ['google_search_agent']
[logging_plugin] 🧠 LLM REQUEST
[logging_plugin] Model: gemini-2.5-flash-lite
[logging_plugin] Agent: research_paper_finder_agent
[logging_plugin] System Instruction: 'Your task is to find research
papers and count them.'
```

You must follow these steps:

- 1) Find research papers on the user provided topic using the 'google\_search\_agent'.
- 2) Then, pass the p...'

```
[logging_plugin] Available Tools: ['google_search_agent',
'count_papers']
[logging_plugin] 🧠 LLM RESPONSE
[logging_plugin] Agent: research_paper_finder_agent
[logging_plugin] Content: function_call: count_papers
[logging_plugin] Token Usage - Input: 856, Output: 591
[logging_plugin] 💡 EVENT YIELDED
[logging_plugin] Event ID: c7f632b3-b4ca-4cbb-928f-b013170db40c
[logging_plugin] Author: research_paper_finder_agent
[logging_plugin] Content: function_call: count_papers
[logging_plugin] Final Response: False
[logging_plugin] Function Calls: ['count_papers']
[logging_plugin] 🔐 TOOL STARTING
[logging_plugin] Tool Name: count_papers
[logging_plugin] Agent: research_paper_finder_agent
[logging_plugin] Function Call ID:
adk-58482ea8-878e-443d-83de-927a89aea240
[logging_plugin] Arguments: {'papers': ["The field of quantum computing
is experiencing rapid advancements, with a particular focus on stabilizing
qubits, developing specialized hardware and software, and improving error
correction techniques. Researchers are also exploring hybrid
quantum-classical systems and the integration ..."]}
[logging_plugin] 🔐 TOOL COMPLETED
[logging_plugin] Tool Name: count_papers
[logging_plugin] Agent: research_paper_finder_agent
[logging_plugin] Function Call ID:
adk-58482ea8-878e-443d-83de-927a89aea240
```

```
[logging_plugin]    Result: 1
[logging_plugin] 🗑 EVENT YIELDED
[logging_plugin]    Event ID: 155fdc19-6b70-4359-96fc-7be92a48274a
[logging_plugin]    Author: research_paper_finder_agent
[logging_plugin]    Content: function_response: count_papers
[logging_plugin]    Final Response: False
[logging_plugin]    Function Responses: ['count_papers']
[logging_plugin] 🧠 LLM REQUEST
[logging_plugin]    Model: gemini-2.5-flash-lite
[logging_plugin]    Agent: research_paper_finder_agent
[logging_plugin]    System Instruction: 'Your task is to find research papers and count them.'
```

You must follow these steps:

- 1) Find research papers on the user provided topic using the 'google\_search\_agent'.
- 2) Then, pass the p...'

```
[logging_plugin]    Available Tools: ['google_search_agent',
'count_papers']
[logging_plugin] 🧠 LLM RESPONSE
[logging_plugin]    Agent: research_paper_finder_agent
[logging_plugin]    Content: text: 'I found 1 research paper on quantum computing. The paper discusses recent advancements and trends in the field, including logical qubits and error correction, specialized hardware and software, hybrid...'
[logging_plugin]    Token Usage - Input: 1462, Output: 61
[logging_plugin] 🗑 EVENT YIELDED
[logging_plugin]    Event ID: 18fb7b67-d4a1-4022-a8c0-28f265402789
[logging_plugin]    Author: research_paper_finder_agent
[logging_plugin]    Content: text: 'I found 1 research paper on quantum computing. The paper discusses recent advancements and trends in the field, including logical qubits and error correction, specialized hardware and software, hybrid...'
[logging_plugin]    Final Response: True
research_paper_finder_agent > I found 1 research paper on quantum computing. The paper discusses recent advancements and trends in the field, including logical qubits and error correction, specialized hardware and software, hybrid quantum-classical systems, and industry applications. It also highlights notable research developments from institutions like Google, IBM, Princeton, and MIT.
```

```
[logging_plugin] 🤖 AGENT COMPLETED
[logging_plugin]     Agent Name: research_paper_finder_agent
[logging_plugin]     Invocation ID: e-c8943591-6d63-4a49-8e91-3a56ca8764ea
[logging_plugin] ✓ INVOCATION COMPLETED
[logging_plugin]     Invocation ID: e-c8943591-6d63-4a49-8e91-3a56ca8764ea
[logging_plugin]     Final Agent: research_paper_finder_agent
```

---

## Summary

### ❓ When to use which type of Logging?

1. **Development debugging?** → Use `adk web --log_level DEBUG`
2. **Common production observability?** → Use `LoggingPlugin()`
3. **Custom requirements?** → Build Custom Callbacks and Plugins

Try it out!

👉 Extend the agent's observability by implementing a **custom ADK plugin** that tracks and reports the total number of tool calls made during a session.

## 🎉 Congratulations!

You now know how to:

- ✓ Debug agent failures through DEBUG logs and the ADK web UI
- ✓ Use the core debugging pattern: symptom → logs → root cause → fix
- ✓ Scale observability with `LoggingPlugin` for production systems
- ✓ Understand when to use the different logging types

 Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

## Resources

Refer to the ADK documentation to learn more about observability:

- [ADK Observability Documentation](#) - Complete guide to logging in ADK
- [Custom Plugin](#) - Build your own Plugins
- [External Integrations](#) - Explore external third-party observability integrations with ADK

## Next Steps

Ready for the next challenge? Continue to the next notebook to learn how to **Evaluate an Agent** and ensure it's working as expected in production.

---

Authors
<a href="#">Sita Lakshmi Sangameswaran</a>

Copyright 2025 Google LLC.

```
# @title 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
#
# https://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.
```



## Agent Evaluation

Welcome to Day 4 of the Kaggle 5-day Agents course!

In the previous notebook, we explored how to implement Observability in AI agents. This approach is primarily **reactive**; it comes into play after an issue has surfaced, providing the necessary data to debug and understand the root cause.

In this notebook, we'll complement those observability practices with a **proactive** approach using **Agent Evaluation**. By continuously evaluating our agent's performance, we can catch any quality degradations much earlier!

Observability + Agent Evaluation

(reactive)      (proactive)

## What is Agent Evaluation?

It is the systematic process of testing and measuring how well an AI agent performs across different scenarios and quality dimensions.



### The story

You've built a home automation agent. It works perfectly in your tests, so you launch it confidently...

- **Week 1:** 🚨 "Agent turned on the fireplace when I asked for lights!"
- **Week 2:** 🚨 "Agent won't respond to commands in the guest room!"
- **Week 3:** 🚨 "Agent gives rude responses when devices are unavailable!"

**The Problem:** Standard testing ≠ Evaluation

Agents are different from traditional software:

- They are non-deterministic
- Users give unpredictable, ambiguous commands
- Small prompt changes cause dramatic behavior shifts and different tool calls

To accommodate all these differences, agents need systematic evaluation, not just "happy path" testing. **Which means assessing the agent's entire decision-making process - including the final response and the path it took to get the response (trajectory)!**

By the end of this notebook, you will be able to:

- Understand what agent evaluation is and how to use it
- Run evaluations and analyze results directly in the ADK web UI
- Detect regression in the agent's performance over a period of time
- Understand and create the necessary evaluation files (\*.test.json, \*.evalset.json, test\_config.json).

## !! Please Read

  **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

---

## Section 1: Setup

Before we begin our evaluation journey, let's set up our environment.

### 1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

### 1.2: Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires an API key.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

#### 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

#### 3. Authenticate in the notebook

Run the cell below to access the GOOGLE\_API\_KEY you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✓ Setup and authentication complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

```
✓ Setup and authentication complete.
```

### 1.3: Set up proxy and tunneling

We'll use a proxy to access the ADK web UI from within the Kaggle Notebooks environment. If you are running this outside the Kaggle environment, you don't need to do this.

```
from IPython.core.display import display, HTML
from jupyter_server.serverapp import list_running_servers

# Gets the proxied URL in the Kaggle Notebooks environment
def get_adk_proxy_url():
    PROXY_HOST = "https://kkb-production.jupyter-proxy.kaggle.net"
    ADK_PORT = "8000"

    servers = list(list_running_servers())
    if not servers:
        raise Exception("No running Jupyter servers found.")
```

```

baseURL = servers[0]["base_url"]

try:
    path_parts = baseURL.split("/")
    kernel = path_parts[2]
    token = path_parts[3]
except IndexError:
    raise Exception(f"Could not parse kernel/token from base URL: {baseURL}")

url_prefix = f"/k/{kernel}/{token}/proxy/proxy/{ADK_PORT}"
url = f"{PROXY_HOST}{url_prefix}"

styled_html = """


<div style="font-family: sans-serif; margin-bottom: 12px; color: #333; font-size: 1.1em;">
        <strong>⚠️ IMPORTANT: Action Required</strong>
    </div>
    <div style="font-family: sans-serif; margin-bottom: 15px; color: #333; line-height: 1.5;">
        The ADK web UI is <strong>not running yet</strong>. You must start it in the next cell.
        <ol style="margin-top: 10px; padding-left: 20px;">
            <li style="margin-bottom: 5px;"><strong>Run the next cell</strong> (the one with <code>!adk web ...</code>) to start the ADK web UI.</li>
            <li style="margin-bottom: 5px;">Wait for that cell to show it is "Running" (it will not "complete").</li>
            <li>Once it's running, <strong>return to this button</strong> and click it to open the UI.</li>
        </ol>
        <em style="font-size: 0.9em; color: #555;">(If you click the button before running the next cell, you will get a 500 error.)</em>
    </div>
    <a href='{url}' target='_blank' style="display: inline-block; background-color: #1a73e8; color: white; padding: 10px 20px;">


```

```
        text-decoration: none; border-radius: 25px; font-family: sans-serif; font-weight: 500; box-shadow: 0 2px 5px rgba(0,0,0,0.2); transition: all 0.2s ease;">
            Open ADK Web UI (after running cell below) ^
        </a>
    </div>
    """
display(HTML(styled_html))

return url_prefix

print("✅ Helper functions defined.")
```

```
✅ Helper functions defined.
```

---

## Section 2: Create a Home Automation Agent

Let's create the agent that will be the center of our evaluation story. This home automation agent seems perfect in basic tests but has hidden flaws we'll discover through comprehensive evaluation. Run the `adk create` CLI command to set up the project scaffolding.

```
!adk create home_automation_agent --model gemini-2.5-flash-lite --api_key
$GOOGLE_API_KEY
```

```
Agent created in /kaggle/working/home_automation_agent:
- .env
```

- `__init__.py`
- `agent.py`

Run the below cell to create the home automation agent.

This agent uses a single `set_device_status` tool to control smart home devices. A device's status can only be ON or OFF. **The agent's instruction is deliberately overconfident** - it claims to control "ALL smart devices" and "any device the user mentions" - setting up the evaluation problems we'll discover.

```
%%writefile home_automation_agent/agent.py

from google.adk.agents import LlmAgent
from google.adk.models.google_llm import Gemini

from google.genai import types

# Configure Model Retry on errors
retry_config = types.HttpRetryOptions(
    attempts=5, # Maximum retry attempts
    exp_base=7, # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors
)

def set_device_status(location: str, device_id: str, status: str) -> dict:
    """Sets the status of a smart home device.

    Args:
        location: The room where the device is located.
        device_id: The unique identifier for the device.
        status: The desired status, either 'ON' or 'OFF'.

    Returns:
        A dictionary confirming the action.
    """

```

```

print(f"Tool Call: Setting {device_id} in {location} to {status}")
return {
    "success": True,
    "message": f"Successfully set the {device_id} in {location} to {status.lower()}."
}

# This agent has DELIBERATE FLAWS that we'll discover through evaluation!
root_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
    retry_options=retry_config),
    name="home_automation_agent",
    description="An agent to control smart devices in a home.",
    instruction="""You are a home automation assistant. You control ALL
smart devices in the house.

    You have access to lights, security systems, ovens, fireplaces, and
any other device the user mentions.

    Always try to be helpful and control whatever device the user asks
for.

    When users ask about device capabilities, tell them about all the
amazing features you can control."""
    tools=[set_device_status],
)

```

Overwriting home\_automation\_agent/agent.py

---

## ✓ Section 3: Interactive Evaluation with ADK Web UI

### 3.1: Launch ADK Web UI

Get the proxied URL to access the ADK web UI in the Kaggle Notebooks environment:

```
url_prefix = get_adk_proxy_url()
```

```
[REDACTED]
```

### **IMPORTANT: Action Required**

The ADK web UI is **not running yet**. You must start it in the next cell.

1. **Run the next cell** (the one with `!adk web . . .`) to start the ADK web UI.
2. Wait for that cell to show it is "Running" (it will not "complete").
3. Once it's running, **return to this button** and click it to open the UI.

*(If you click the button before running the next cell, you will get a 500 error.)*

[Open ADK Web UI \(after running cell below\) ↗](#)

Now you can start the ADK web UI using the following command.

 **Note:** The following cell will not "complete", but will remain running and serving the ADK web UI until you manually stop the cell.

```
!adk web --url_prefix {url_prefix}
```

```
[REDACTED]  
/usr/local/lib/python3.11/dist-packages/google/adk/cli/fast_api.py:130:  
UserWarning: [EXPERIMENTAL] InMemoryCredentialService: This feature is  
experimental and may change or be removed in future versions without  
notice. It may introduce breaking changes at any time.  
    credential_service = InMemoryCredentialService()  
/usr/local/lib/python3.11/dist-packages/google/adk/auth/credential_service  
/in_memory_credential_service.py:33: UserWarning: [EXPERIMENTAL]
```

BaseCredentialService: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.

```
super().__init__()
```

```
INFO: Started server process [87]
```

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

```
+-----+
----+
| ADK Web Server started
|
|
|
| For local testing, access at http://127.0.0.1:8000.
|
+-----+
----+
```

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

```
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

```
INFO: 35.191.85.0:0 - "GET / HTTP/1.1" 307 Temporary Redirect
```

```
INFO: 35.191.85.3:0 - "GET /dev-ui/ HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.2:0 - "GET /dev-ui/main-OS20H2S3.js HTTP/1.1" 200 OK
```

```
INFO: 35.191.114.48:0 - "GET /dev-ui/polyfills-B6TNHZQ6.js HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.3:0 - "GET /dev-ui/chunk-2WH2EVR6.js HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.0:0 - "GET /dev-ui/styles-EVMPSV3U.css HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.1:0 - "GET /dev-ui/assets/config/runtime-config.json HTTP/1.1" 200 OK
```

```
INFO: 35.191.114.48:0 - "GET /dev-ui/adk_favicon.svg HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.3:0 - "GET /list-apps?relative_path=../ HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.1:0 - "GET /dev-ui/assets/ADK-512-color.svg HTTP/1.1" 200 OK
```

```
INFO: 35.191.85.3:0 - "GET
```

```
/builder/app/home_automation_agent?ts=1762973405499 HTTP/1.1" 200 OK
```

```
INFO:google_adk.google.adk.cli.adk_web_server:New session created:
```

```
11f272fb-f872-46bc-b642-22a5c17ecfd8
```

```
INFO:      35.191.85.0:0 - "POST
/apps/home_automation_agent/users/user/sessions HTTP/1.1" 200 OK
INFO:      35.191.85.0:0 - "GET
/apps/home_automation_agent/users/user/sessions HTTP/1.1" 200 OK
INFO:      35.191.85.3:0 - "GET
/apps/home_automation_agent/users/user/sessions/11f272fb-f872-46bc-b642-22
a5c17ecfd8 HTTP/1.1" 200 OK
INFO:      35.191.85.2:0 - "GET
/debug/trace/session/11f272fb-f872-46bc-b642-22a5c17ecfd8 HTTP/1.1" 200 OK
INFO:      35.191.85.2:0 - "GET /apps/home_automation_agent/eval_sets
HTTP/1.1" 200 OK
INFO:      35.191.85.1:0 - "GET /apps/home_automation_agent/eval_results
HTTP/1.1" 200 OK
INFO:      35.191.85.1:0 - "GET
/apps/home_automation_agent/users/user/sessions HTTP/1.1" 200 OK
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with
value 'appName' was provided to the `Field()` function, which has no
effect in the context it was used. 'alias' is field-specific metadata, and
can only be attached to a model field using `Annotated` metadata or by
assignment. This may have happened because an `Annotated` type alias using
the `type` statement was used, or if the `Field()` function was attached
to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'
attribute with value 'appName' was provided to the `Field()` function,
which has no effect in the context it was used. 'validation_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'
attribute with value 'appName' was provided to the `Field()` function,
which has no effect in the context it was used. 'serialization_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
```

```
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with
value 'userId' was provided to the `Field()` function, which has no effect
in the context it was used. 'alias' is field-specific metadata, and can
only be attached to a model field using `Annotated` metadata or by
assignment. This may have happened because an `Annotated` type alias using
the `type` statement was used, or if the `Field()` function was attached
to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'
attribute with value 'userId' was provided to the `Field()` function,
which has no effect in the context it was used. 'validation_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'
attribute with value 'userId' was provided to the `Field()` function,
which has no effect in the context it was used. 'serialization_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with
value 'sessionId' was provided to the `Field()` function, which has no
effect in the context it was used. 'alias' is field-specific metadata, and
can only be attached to a model field using `Annotated` metadata or by
assignment. This may have happened because an `Annotated` type alias using
the `type` statement was used, or if the `Field()` function was attached
to a single member of a union type.

    warnings.warn(
```

```
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'
attribute with value 'sessionId' was provided to the `Field()` function,
which has no effect in the context it was used. 'validation_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'
attribute with value 'sessionId' was provided to the `Field()` function,
which has no effect in the context it was used. 'serialization_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with
value 'newMessage' was provided to the `Field()` function, which has no
effect in the context it was used. 'alias' is field-specific metadata, and
can only be attached to a model field using `Annotated` metadata or by
assignment. This may have happened because an `Annotated` type alias using
the `type` statement was used, or if the `Field()` function was attached
to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'
attribute with value 'newMessage' was provided to the `Field()` function,
which has no effect in the context it was used. 'validation_alias' is
field-specific metadata, and can only be attached to a model field using
`Annotated` metadata or by assignment. This may have happened because an
`Annotated` type alias using the `type` statement was used, or if the
`Field()` function was attached to a single member of a union type.

    warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'
attribute with value 'newMessage' was provided to the `Field()` function,
which has no effect in the context it was used. 'serialization_alias' is
```

field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by assignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Field()` function was attached to a single member of a union type.

```
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with  
value 'streaming' was provided to the `Field()` function, which has no  
effect in the context it was used. 'alias' is field-specific metadata, and  
can only be attached to a model field using `Annotated` metadata or by  
assignment. This may have happened because an `Annotated` type alias using  
the `type` statement was used, or if the `Field()` function was attached  
to a single member of a union type.
```

```
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'  
attribute with value 'streaming' was provided to the `Field()` function,  
which has no effect in the context it was used. 'validation_alias' is  
field-specific metadata, and can only be attached to a model field using  
`Annotated` metadata or by assignment. This may have happened because an  
`Annotated` type alias using the `type` statement was used, or if the  
`Field()` function was attached to a single member of a union type.
```

```
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'  
attribute with value 'streaming' was provided to the `Field()` function,  
which has no effect in the context it was used. 'serialization_alias' is  
field-specific metadata, and can only be attached to a model field using  
`Annotated` metadata or by assignment. This may have happened because an  
`Annotated` type alias using the `type` statement was used, or if the  
`Field()` function was attached to a single member of a union type.
```

```
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with  
value 'stateDelta' was provided to the `Field()` function, which has no  
effect in the context it was used. 'alias' is field-specific metadata, and  
can only be attached to a model field using `Annotated` metadata or by  
assignment. This may have happened because an `Annotated` type alias using  
the `type` statement was used, or if the `Field()` function was attached  
to a single member of a union type.
```

```
warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'  
attribute with value 'stateDelta' was provided to the `Field()`` function,  
which has no effect in the context it was used. 'validation_alias' is  
field-specific metadata, and can only be attached to a model field using  
'Annotated` metadata or by assignment. This may have happened because an  
'Annotated` type alias using the `type` statement was used, or if the  
'Field()`` function was attached to a single member of a union type.  
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'  
attribute with value 'stateDelta' was provided to the `Field()`` function,  
which has no effect in the context it was used. 'serialization_alias' is  
field-specific metadata, and can only be attached to a model field using  
'Annotated` metadata or by assignment. This may have happened because an  
'Annotated` type alias using the `type` statement was used, or if the  
'Field()`` function was attached to a single member of a union type.  
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'alias' attribute with  
value 'invocationId' was provided to the `Field()`` function, which has no  
effect in the context it was used. 'alias' is field-specific metadata, and  
can only be attached to a model field using 'Annotated` metadata or by  
assignment. This may have happened because an 'Annotated` type alias using  
the `type` statement was used, or if the `Field()`` function was attached  
to a single member of a union type.  
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'validation_alias'  
attribute with value 'invocationId' was provided to the `Field()``  
function, which has no effect in the context it was used.  
'validation_alias' is field-specific metadata, and can only be attached to  
a model field using 'Annotated` metadata or by assignment. This may have  
happened because an 'Annotated` type alias using the `type` statement was  
used, or if the `Field()`` function was attached to a single member of a  
union type.  
    warnings.warn(  
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schem  
a.py:2249: UnsupportedFieldAttributeWarning: The 'serialization_alias'
```

attribute with value 'invocationId' was provided to the `Field()` function, which has no effect in the context it was used.  
'serialization\_alias' is field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by assignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Field()` function was attached to a single member of a union type.

```
warnings.warn(  
INFO: 35.191.114.48:0 - "POST /run_sse HTTP/1.1" 200 OK  
INFO:google_adk.google.adk.cli.utils.agent_loader:Found root_agent in home_automation_agent.agent  
INFO:google_adk.google.adk.models.google_llm:Sending out request, model: gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False  
INFO:google_adk.google.adk.models.google_llm:Response received from the model.  
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.  
Tool Call: Setting desk lamp in office to ON  
INFO:google_adk.google.adk.models.google_llm:Sending out request, model: gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False  
INFO:google_adk.google.adk.models.google_llm:Response received from the model.  
INFO: 35.191.114.48:0 - "GET  
/debug/trace/session/11f272fb-f872-46bc-b642-22a5c17ecfd8 HTTP/1.1" 200 OK  
INFO: 35.191.85.0:0 - "GET  
/apps/home_automation_agent/users/user/sessions/11f272fb-f872-46bc-b642-22a5c17ecfd8 HTTP/1.1" 200 OK  
INFO: 35.191.114.48:0 - "GET  
/debug/trace/session/11f272fb-f872-46bc-b642-22a5c17ecfd8 HTTP/1.1" 200 OK  
^C  
INFO: Shutting down  
INFO: Waiting for application shutdown.  
  
+-----+  
| ADK Web Server shutting down...  
|
```

```
+-----+
|-----+
```

INFO: Application shutdown complete.  
INFO: Finished server process [87]

Aborted!

Once the ADK web UI starts, open the proxy link using the button in the previous cell.

**!! IMPORTANT: DO NOT SHARE THE PROXY LINK** with anyone - treat it as sensitive data as it contains your authentication token in the URL.

### 3.2: Create Your First "Perfect" Test Case

👉 Do: In the ADK web UI:

1. Click the public URL above to open the ADK web UI
2. Select "home\_automation\_agent" from the dropdown
3. **Have a normal conversation:** Type Turn on the desk lamp in the office
4. **Agent responds correctly** - controls device and confirms action

👉 Do: Save this as your first evaluation case:

1. Navigate to the **Eval** tab on the right-hand panel
2. Click **Create Evaluation set** and name it home\_automation\_tests
3. In the home\_automation\_tests set, click the ">" arrow and click **Add current session**
4. Give it the case name basic\_device\_control

✓ Success! You've just saved your first interaction as an evaluation case.



### 3.3: Run the Evaluation

## 👉 Do: Run your first evaluation

Now, let's run the test case to see if the agent can replicate its previous success.

1. In the Eval tab, make sure your new test case is checked.
2. Click the Run Evaluation button.
3. The EVALUATION METRIC dialog will appear. For now, leave the default values and click Start.
4. The evaluation will run, and you should see a green Pass result in the Evaluation History. This confirms the agent's behavior matched the saved session.

## !! Understanding the Evaluation Metrics

When you run evaluation, you'll see two key scores:

- **Response Match Score:** Measures how similar the agent's actual response is to the expected response. Uses text similarity algorithms to compare content. A score of 1.0 = perfect match, 0.0 = completely different.
- **Tool Trajectory Score:** Measures whether the agent used the correct tools with correct parameters. Checks the sequence of tool calls against expected behavior. A score of 1.0 = perfect tool usage, 0.0 = wrong tools or parameters.

## 👉 Do: Analyze a Failure

Let's intentionally break the test to see what a failure looks like.

1. In the list of eval cases, click the Edit (pencil) icon next to your test case.
2. In the "Final Response" text box, change the expected text to something incorrect, like: The desk lamp is off.
3. Save the changes and re-run the evaluation.
4. This time, the result will be a red Fail. Hover your mouse over the "Fail" label. A tooltip will appear showing a side-by-side comparison of the Actual vs. Expected Output, highlighting exactly why the test failed (the final response didn't match). This immediate, detailed feedback is invaluable for debugging.



## 3.4: (Optional) Create challenging test cases

Now create more test cases to expose hidden problems:

**Create these scenarios in separate conversations:**

1. **Ambiguous Commands:** "Turn on the lights in the bedroom"
  - Save as a new test case: ambiguous\_device\_reference
  - Run evaluation - it likely passes but the agent might be confused
2. **Invalid Locations:** "Please turn off the TV in the garage"
  - Save as a new test case: invalid\_location\_test
  - Run evaluation - the agent might try to control non-existent devices
3. **Complex Commands:** "Turn off all lights and turn on security system"
  - Save as a new test case: complex\_multi\_device\_command
  - Run evaluation - the agent might attempt operations beyond its capabilities

**The Problem You'll Discover:** Even when tests "pass," you can see the agent:

- Makes assumptions about devices that don't exist
- Gives responses that sound helpful but aren't accurate
- Tries to control devices it shouldn't have access to

 **What am I missing?**

 **Web UI Limitation:** So far, we've seen how to create and evaluate test cases in the ADK web UI. The web UI is great for interactive test creation, but testing one conversation at a time doesn't scale.

 **The Question:** How do I proactively detect regressions in my agent's performance?

Let's answer that question in the next section!

---

**!! Stop the ADK web UI** 

In order to run cells in the remainder of this notebook, please stop the running cell where you started adk\_web in Section 3.1.

Otherwise that running cell will block / prevent other cells from running as long as the ADK web UI is running.

---



## Section 4: Systematic Evaluation

Regression testing is the practice of re-running existing tests to ensure that new changes haven't broken previously working functionality.

ADK provides two methods to do automatic regression and batch testing: using [pytest](#) and the [adk eval](#) CLI command. In this section, we'll use the CLI command. For more information on the pytest approach, refer to the links in the resource section at the end of this notebook.

The following image shows the overall process of evaluation. **At a high-level, there are four steps to evaluate:**

- 1) **Create an evaluation configuration** - define metrics or what you want to measure
- 2) **Create test cases** - sample test cases to compare against
- 3) **Run the agent with test query**
- 4) **Compare the results**



### 4.1: Create evaluation configuration

This optional file lets us define the pass/fail thresholds. Create `test_config.json` in the root directory.

```
import json

# Create evaluation configuration with basic criteria
eval_config = {
    "criteria": {
```

```

        "tool_trajectory_avg_score": 1.0, # Perfect tool usage required
        "response_match_score": 0.8, # 80% text similarity threshold
    }
}

with open("home_automation_agent/test_config.json", "w") as f:
    json.dump(eval_config, f, indent=2)

print("✓ Evaluation configuration created!")
print("\n📊 Evaluation Criteria:")
print("• tool_trajectory_avg_score: 1.0 - Requires exact tool usage match")
print("• response_match_score: 0.8 - Requires 80% text similarity")
print("\n⌚ What this evaluation will catch:")
print("✓ Incorrect tool usage (wrong device, location, or status)")
print("✓ Poor response quality and communication")
print("✓ Deviations from expected behavior patterns")

```

Evaluation configuration created!

Evaluation Criteria:

- tool\_trajectory\_avg\_score: 1.0 - Requires exact tool usage match
- response\_match\_score: 0.8 - Requires 80% text similarity

What this evaluation will catch:

- Incorrect tool usage (wrong device, location, or status)
- Poor response quality and communication
- Deviations from expected behavior patterns

## 4.2: Create test cases

This file (`integration.evalset.json`) will contain multiple test cases (sessions).

This evaluation set can be created synthetically or from the conversation sessions in the ADK web UI.

**Tip:** To persist the conversations from the ADK web UI, simply create an evalset in the UI and add the current session to it. All the conversations in that session will be auto-converted to an evalset and downloaded locally.

```
# Create evaluation test cases that reveal tool usage and response quality
# problems
test_cases = {
    "eval_set_id": "home_automation_integration_suite",
    "eval_cases": [
        {
            "eval_id": "living_room_light_on",
            "conversation": [
                {
                    "user_content": {
                        "parts": [
                            {"text": "Please turn on the floor lamp in the
living room"}
                        ]
                    },
                    "final_response": {
                        "parts": [
                            {
                                "text": "Successfully set the floor lamp
in the living room to on."
                            }
                        ]
                    },
                    "intermediate_data": {
                        "tool_uses": [
                            {
                                "name": "set_device_status",
                                "args": {
                                    "location": "living room",
                                    "device_id": "floor lamp",
                                    "status": "ON",
                                },
                            }
                        ]
                    },
                }
            ]
        }
    ]
}
```

```

        ],
    },
{
    "eval_id": "kitchen_on_off_sequence",
    "conversation": [
        {
            "user_content": {
                "parts": [{"text": "Switch on the main light in
the kitchen."}]
            },
            "final_response": {
                "parts": [
                    {
                        "text": "Successfully set the main light
in the kitchen to on."
                    }
                ]
            },
            "intermediate_data": {
                "tool_uses": [
                    {
                        "name": "set_device_status",
                        "args": {
                            "location": "kitchen",
                            "device_id": "main light",
                            "status": "ON",
                        },
                    }
                ]
            },
        },
    ],
},
]
}

```

Let's write the test cases to the `integration.evalset.json` in our agent's root directory.

```

import json

with open("home_automation_agent/integration.evalset.json", "w") as f:
    json.dump(test_cases, f, indent=2)

print("✓ Evaluation test cases created")
print("\n📝 Test scenarios:")
for case in test_cases["eval_cases"]:
    user_msg = case["conversation"][0]["user_content"]["parts"][0]["text"]
    print(f"• {case['eval_id']}: {user_msg}")

print("\n📊 Expected results:")
print("• basic_device_control: Should pass both criteria")
print(
    "• wrong_tool_usage_test: May fail tool_trajectory if agent uses wrong
parameters"
)
print(
    "• poor_response_quality_test: May fail response_match if response
differs too much"
)

```

✓ Evaluation test cases created

Test scenarios:

- living\_room\_light\_on: Please turn on the floor lamp in the living room
- kitchen\_on\_off\_sequence: Switch on the main light in the kitchen.

Expected results:

- basic\_device\_control: Should pass both criteria
- wrong\_tool\_usage\_test: May fail tool\_trajectory if agent uses wrong parameters
- poor\_response\_quality\_test: May fail response\_match if response differs too much

#### 4.3: Run CLI Evaluation

Execute the adk eval command, pointing it to your agent directory, the evalset, and the config file.

```
print("🚀 Run this command to execute evaluation:")
!adk eval home_automation_agent
home_automation_agent/integration.evalset.json
--config_file_path=home_automation_agent/test_config.json
--print_detailed_results
```

```
🚀 Run this command to execute evaluation:
/usr/local/lib/python3.11/dist-packages/google/adk/evaluation/metric_evaluator_registry.py:90: UserWarning: [EXPERIMENTAL] MetricEvaluatorRegistry: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
    metric_evaluator_registry = MetricEvaluatorRegistry()
/usr/local/lib/python3.11/dist-packages/google/adk/evaluation/local_eval_service.py:79: UserWarning: [EXPERIMENTAL] UserSimulatorProvider: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
    user_simulator_provider: UserSimulatorProvider =
UserSimulatorProvider(),
Using evaluation criteria: criteria={'tool_trajectory_avg_score': 1.0,
'response_match_score': 0.8} user_simulator_config=None
/usr/local/lib/python3.11/dist-packages/google/adk/cli/cli_tools_click.py:650: UserWarning: [EXPERIMENTAL] UserSimulatorProvider: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
    user_simulator_provider = UserSimulatorProvider(
/usr/local/lib/python3.11/dist-packages/google/adk/cli/cli_tools_click.py:655: UserWarning: [EXPERIMENTAL] LocalEvalService: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
    eval_service = LocalEvalService(
/usr/local/lib/python3.11/dist-packages/google/adk/evaluation/user_simulator_provider.py:77: UserWarning: [EXPERIMENTAL] StaticUserSimulator: This feature is experimental and may change or be removed in future versions without notice. It may introduce breaking changes at any time.
    return StaticUserSimulator(static_conversation=eval_case.conversation)
```

```
/usr/local/lib/python3.11/dist-packages/google/adk/evaluation/static_user_
simulator.py:39: UserWarning: [EXPERIMENTAL] UserSimulator: This feature
is experimental and may change or be removed in future versions without
notice. It may introduce breaking changes at any time.
    super().__init__()
INFO:google_adk.google.adk.plugins.plugin_manager:Plugin
'request_interceptor_plugin' registered.
INFO:google_adk.google.adk.models.google_llm:Sending out request, model:
gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False
INFO:google_adk.google.adk.plugins.plugin_manager:Plugin
'request_interceptor_plugin' registered.
INFO:google_adk.google.adk.models.google_llm:Sending out request, model:
gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False
INFO:google_adk.google.adk.models.google_llm:Response received from the
model.
INFO:google_adk.google.adk.models.google_llm:Response received from the
model.
WARNING:google_genai.types:Warning: there are non-text parts in the
response: ['function_call'], returning concatenated text result from text
parts. Check the full candidates.content.parts accessor to get the full
model response.
Tool Call: Setting main light in kitchen to ON
INFO:google_adk.google.adk.models.google_llm:Sending out request, model:
gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False
INFO:google_adk.google.adk.models.google_llm:Response received from the
model.
INFO:google_adk.google.adk.evaluation.local_eval_set_results_manager:Writing eval result to file:
/kaggle/working/home_automation_agent/.adk/eval_history/home_automation_ag
ent_home_automation_integration_suite_1762973450.1228065.evalset_result.js
on
INFO:google_adk.google.adk.evaluation.local_eval_set_results_manager:Writing eval result to file:
/kaggle/working/home_automation_agent/.adk/eval_history/home_automation_ag
ent_home_automation_integration_suite_1762973450.1237617.evalset_result.js
on
*****
Eval Run Summary
home_automation_integration_suite:
    Tests passed: 0
```

```
Tests failed: 2
*****
Eval Set Id: home_automation_integration_suite
Eval Id: living_room_light_on
Overall Eval Status: FAILED
-----
Metric: tool_trajectory_avg_score, Status: FAILED, Score: 0.0, Threshold: 1.0
-----
Metric: response_match_score, Status: FAILED, Score: 0.32, Threshold: 0.8
-----
Invocation Details:
+-----+-----+-----+
| prompt | expected_response | 
actual_response | expected_tool_calls | actual_tool_calls
| tool_trajectory_avg_score | response_match_score | 
+=====+=====+=====+
=====+=====+=====+=====+
| 0 | Please turn on the floor | Successfully set the | I can help
with that. | id=None args={'location': | | Status:
FAILED, Score: | Status: FAILED, Score: |
| lamp in the living room | floor lamp in the living | What is the
device ID of | 'living room', | | 0.0
| 0.32 | 
| | | room to on. | the floor
| lamp? | 'device_id': 'floor' | | 
| | | | 
| | | | 
| lamp', 'status': 'ON'} | | 
| | | | 
| | | | 
| name='set_device_status' | | 
| | | 
+-----+-----+-----+
-----+-----+-----+
-----+-----+
```

```
*****
Eval Set Id: home_automation_integration_suite
Eval Id: kitchen_on_off_sequence
Overall Eval Status: FAILED
-----
Metric: tool_trajectory_avg_score, Status: PASSED, Score: 1.0, Threshold: 1.0
-----
Metric: response_match_score, Status: FAILED, Score: 0.7, Threshold: 0.8
-----
Invocation Details:
+---+-----+-----+-----+
|   | prompt           | expected_response      |
actual_response       | expected_tool_calls    | actual_tool_calls
| tool_trajectory_avg_score | response_match_score  |
+=====+=====+=====+=====+
=====+=====+=====+=====+
=====+=====+=====+=====+
| 0 | Switch on the main light | Successfully set the main | The main
light in the          | id=None args={'location': | id='adk- aebc88a8-4e81-43 |
Status: PASSED, Score:      | Status: FAILED, Score: |
|   | in the kitchen.        | light in the kitchen to | kitchen has
been switched | 'kitchen', 'device_id': | 46-bef0-69ed0bb7d25a' |
1.0                  | 0.7                  |
|   |                         | on.                      | on.
| 'main light', 'status': | args={'device_id': 'main | |
|                           | |
|   |                         |                         |
| 'ON'}                   | light', 'location':    |
|                           | |
|   |                         |                         |
| name='set_device_status'| 'kitchen', 'status':    |
|                           | |
|   |                         |                         |
| 'ON'}                   | 'ON'}                  |
|                           |
```

```
| | | name='set_device_status' | |  
| | +-----+-----+-----+  
-----+-----+-----+  
-----+-----+
```

#### 4.4: Analyzing sample evaluation results

The command will run all test cases and print a summary. The --print\_detailed\_results flag provides a turn-by-turn breakdown of each test, showing scores and a diff for any failures.

```
# Analyzing evaluation results - the data science approach  
print("📊 Understanding Evaluation Results:")  
print()  
print("🔍 EXAMPLE ANALYSIS:")  
print()  
print("Test Case: living_room_light_on")  
print("  ✗ response_match_score: 0.45/0.80")  
print("  ✓ tool_trajectory_avg_score: 1.0/1.0")  
print()  
print("⚠ What this tells us:")  
print("• TOOL USAGE: Perfect - Agent used correct tool with correct parameters")  
print("• RESPONSE QUALITY: Poor - Response text too different from expected")  
print("• ROOT CAUSE: Agent's communication style, not functionality")  
print()  
print("⌚ ACTIONABLE INSIGHTS:")  
print("1. Technical capability works (tool usage perfect)")  
print("2. Communication needs improvement (response quality failed)")  
print("3. Fix: Update agent instructions for clearer language or constrained response.")  
print()
```

## Understanding Evaluation Results:

### EXAMPLE ANALYSIS:

Test Case: living\_room\_light\_on

-  response\_match\_score: 0.45/0.80
-  tool\_trajectory\_avg\_score: 1.0/1.0

### What this tells us:

- TOOL USAGE: Perfect - Agent used correct tool with correct parameters
- RESPONSE QUALITY: Poor - Response text too different from expected
- ROOT CAUSE: Agent's communication style, not functionality

### ACTIONABLE INSIGHTS:

1. Technical capability works (tool usage perfect)
2. Communication needs improvement (response quality failed)
3. Fix: Update agent instructions for clearer language or constrained response.

---

## Section 5: User Simulation (Optional)

While **traditional evaluation methods rely on fixed test cases**, real-world conversations are dynamic and unpredictable. This is where User Simulation comes in.

User Simulation is a powerful feature in ADK that addresses the limitations of static evaluation. Instead of using pre-defined, fixed user prompts, User Simulation employs a generative AI model (like Gemini) to **dynamically generate user prompts during the evaluation process**.

### How it works

- You define a ConversationScenario that outlines the user's overall conversational goals and a conversation\_plan to guide the dialogue.
- A large language model (LLM) then acts as a simulated user, using this plan and the ongoing conversation history to generate realistic and varied prompts.
- This allows for more comprehensive testing of your agent's ability to handle unexpected turns, maintain context, and achieve complex goals in a more natural, unpredictable conversational flow.

User Simulation helps you uncover edge cases and improve your agent's robustness in ways that static test cases often miss.

## 👉 Exercise

Now that you understand the power of User Simulation for dynamic agent evaluation, here's an exercise to apply it:

Apply the **User Simulation** feature to your agent. Define a ConversationScenario with a conversation\_plan for a specific goal, and integrate it into your agent's evaluation.

★ Refer to this [documentation](#) to learn how to do it.

## 🏆 Congratulations!

You've learned

- Interactive test creation and analysis in the ADK web UI
- Tool trajectory and response metrics
- Automated regression testing using adk eval CLI command
- How to analyze evaluation results and fix agents based on it

**i Note: No submission required!**

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

## 📚 Resources

- [ADK Evaluation overview](#)
- Different [evaluation criteria](#)
- [Pytest based Evaluation](#)

## Advanced Evaluation

For production deployments, ADK supports [advanced criteria](#) like safety\_v1 and hallucinations\_v1 (requires Google Cloud credentials).

## ⌚ Next Steps

Ready for the next challenge? Stay tuned for the final Day 5 notebooks where we'll bring it all home!



We'll learn how to **Deploy an Agent to Production** and extend them with **Agent2Agent Protocol**.

---

Authors
<a href="#">Sita Lakshmi Sangameswaran</a>
<a href="#">Ivan Nardini</a>

Copyright 2025 Google LLC.

```
# @title 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
#
```

```
# https://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.
```



## Agent2Agent (A2A) Communication with ADK

Welcome to the Kaggle 5-day Agents course!

This notebook teaches you how to build **multi-agent systems** where different agents can communicate and collaborate using the **Agent2Agent (A2A) Protocol**. You'll learn how to integrate with external agent services and consume remote agents as if they were local.

In this notebook, you'll:

- Understand the A2A protocol and when to use it vs sub-agents
- Learn common A2A architecture patterns (cross-framework, cross-language, cross-organization)
- Expose an ADK agent via A2A using `to_a2a()`
- Consume remote agents using `RemoteA2aAgent`
- Build a product catalog integration system

## !! Please Read



**Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.



**Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.



## Overview of Agent2Agent (A2A)



### The Problem

As you build more complex AI systems, you'll find that:

- **A single agent can't do everything** - Specialized agents for different domains work better
- **You need agents to collaborate** - Customer support needs product data, order systems need inventory info
- **Different teams build different agents** - You want to integrate agents from external vendors
- **Agents may use different languages/frameworks** - You need a standard communication protocol



### The Solution: A2A Protocol

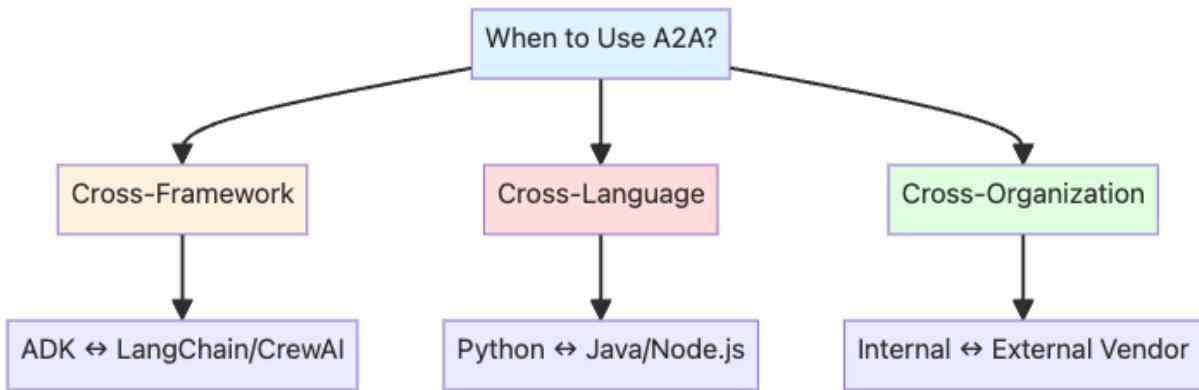
The [Agent2Agent \(A2A\) Protocol](#) is a **standard** that allows agents to:

- **✨ Communicate over networks** - Agents can be on different machines
- **✨ Use each other's capabilities** - One agent can call another agent like a tool
- **✨ Work across frameworks** - Language/framework agnostic
- **✨ Maintain formal contracts** - Agent cards describe capabilities



### Common A2A Architecture Patterns

The A2A protocol is particularly useful in three scenarios:

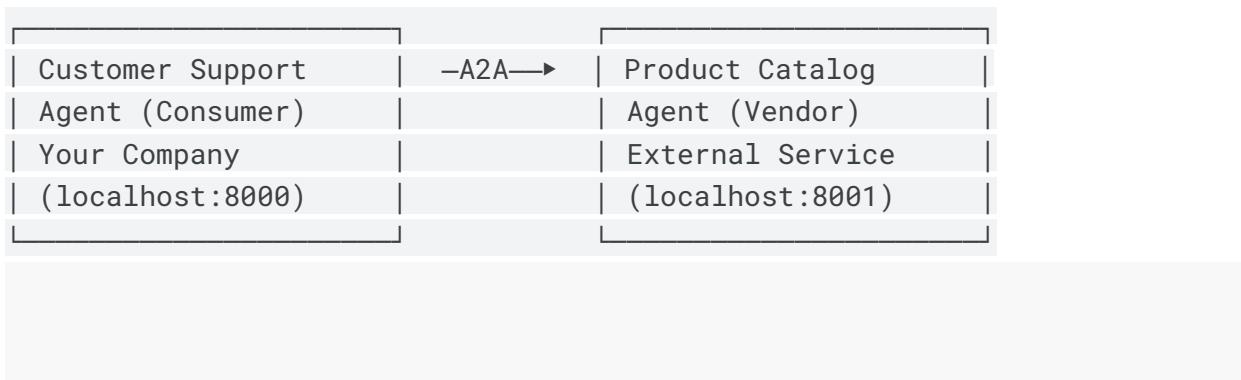


1. **Cross-Framework Integration:** ADK agent communicating with other agent frameworks
2. **Cross-Language Communication:** Python agent calling Java or Node.js agent
3. **Cross-Organization Boundaries:** Your internal agent integrating with external vendor services

### What This Notebook Demonstrates

We'll build a practical e-commerce integration:

1. **Product Catalog Agent** (exposed via A2A) - External vendor service that provides product information
2. **Customer Support Agent** (consumer) - Your internal agent that helps customers by querying product data



**Why this justifies A2A:**

- Product Catalog is maintained by an external vendor (you can't modify their code)

- Different organizations with separate systems
- Formal contract needed between services
- Product Catalog could be in a different language/framework

 A2A vs Local Sub-Agents: Decision Table

Factor	Use A2A	Use Local Sub-Agents
<b>Agent Location</b>	External service, different codebase	Same codebase, internal
<b>Ownership</b>	Different team/organization	Your team
<b>Network</b>	Agents on different machines	Same process/machine
<b>Performance</b>	Network latency acceptable	Need low latency
<b>Language/Framework</b>	Cross-language/framework needed	Same language
<b>Contract</b>	Formal API contract required	Internal interface
<b>Example</b>	External vendor product catalog	Internal order processing steps



In this tutorial, we'll simulate both agents locally (both running on localhost) for learning purposes.

In production:

- Product Catalog Agent would run on vendor's infrastructure (e.g., <https://vendor.com>)
- Customer Support Agent would run on your infrastructure
- They'd communicate over the internet using A2A protocol

This local simulation lets you learn A2A without needing to deploy services!

---

## What We'll Build

Here's a high-level view of the system architecture you'll create in this tutorial:



## How it works:

1. **Customer** asks a product question to your Customer Support Agent
  2. **Support Agent** realizes it needs product information
  3. **Support Agent** calls the **Product Catalog Agent** via A2A protocol
  4. **Product Catalog Agent** (external vendor) returns product data
  5. **Support Agent** formulates an answer and responds to the customer
- 

## Tutorial Steps

In this tutorial, you'll complete **6 steps**:

1. **Create the Product Catalog Agent** - Build the vendor's agent with product lookup
2. **Expose via A2A** - Make it accessible using `to_a2a()`
3. **Start the Server** - Run the agent as a background service
4. **Create the Customer Support Agent** - Build the consumer agent
5. **Test Communication** - See A2A in action with real queries
6. **Understand the Flow** - Learn what happened behind the scenes

Let's get started! 

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

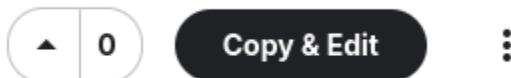
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy` and `Edit` button in the top-right corner.

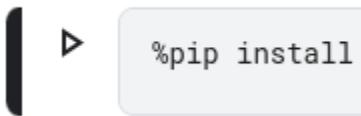


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶ Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

#### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

## ⚙️ Setup

Before we go into today's concepts, follow the steps below to set up the environment.

### Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the [google-adk](#) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK, including A2A and its dependencies, in your own Python development environment outside of this course, you can do so by running:

```
pip install -q google-adk[a2a]
```

### Configure your Gemini API Key

This notebook uses the [Gemini API](#), which requires an API key.

#### 1. Get your API key

If you don't have one already, create an [API key in Google AI Studio](#).

## 2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select Add-ons then Secrets.
2. Create a new secret with the label GOOGLE\_API\_KEY.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to GOOGLE\_API\_KEY is selected so that the secret is attached to the notebook.

## 3. Authenticate in the notebook

Run the cell below to access the GOOGLE\_API\_KEY you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Setup and authentication complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added
'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

 Setup and authentication complete.

## Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
import json
import requests
import subprocess
import time
import uuid

from google.adk.agents import LlmAgent
from google.adk.agents.remote_a2a_agent import (
    RemoteA2aAgent,
    AGENT_CARD_WELL_KNOWN_PATH,
)

from google.adk.a2a.utils.agent_to_a2a import to_a2a
from google.adk.models.google_llm import Gemini
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types

# Hide additional warnings in the notebook
import warnings

warnings.filterwarnings("ignore")

print("✅ ADK components imported successfully.")
```

✅ ADK components imported successfully.

## Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config = types.HttpRetryOptions(
    attempts=5, # Maximum retry attempts
    exp_base=7, # Delay multiplier
```

```
    initial_delay=1,  
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors  
)
```

## 📦 Section 1: Create the Product Catalog Agent (To Be Exposed)

We'll create a **Product Catalog Agent** that provides product information from an external vendor's catalog. This agent will be **exposed via A2A** so other agents (like customer support) can use it.

Why expose this agent?

- In a real system, this would be maintained by an **external vendor** or third-party provider
- Your internal agents (customer support, sales, inventory) need product data
- The vendor **controls their own codebase** - you can't modify their implementation
- By exposing it via A2A, any authorized agent can consume it using the standard protocol

```
# Define a product catalog lookup tool  
# In a real system, this would query the vendor's product database  
def get_product_info(product_name: str) -> str:  
    """Get product information for a given product.  
  
    Args:  
        product_name: Name of the product (e.g., "iPhone 15 Pro", "MacBook  
        Pro")  
  
    Returns:  
        Product information as a string  
    """  
  
    # Mock product catalog - in production, this would query a real  
    # database  
    product_catalog = {  
        "iphone 15 pro": "iPhone 15 Pro, $999, Low Stock (8 units), 128GB,  
        Titanium finish",  
        "samsung galaxy s24": "Samsung Galaxy S24, $799, In Stock (31  
        units), 256GB, Phantom Black",
```

```

        "dell xps 15": 'Dell XPS 15, $1,299, In Stock (45 units), 15.6" display, 16GB RAM, 512GB SSD',
        "macbook pro 14": 'MacBook Pro 14", $1,999, In Stock (22 units), M3 Pro chip, 18GB RAM, 512GB SSD',
        "sony wh-1000xm5": "Sony WH-1000XM5 Headphones, $399, In Stock (67 units), Noise-canceling, 30hr battery",
        "ipad air": 'iPad Air, $599, In Stock (28 units), 10.9" display, 64GB',
        "lg ultrawide 34": 'LG UltraWide 34" Monitor, $499, Out of Stock, Expected: Next week',
    }

product_lower = product_name.lower().strip()

if product_lower in product_catalog:
    return f"Product: {product_catalog[product_lower]}"
else:
    available = ", ".join([p.title() for p in product_catalog.keys()])
    return f"Sorry, I don't have information for {product_name}.\nAvailable products: {available}"

# Create the Product Catalog Agent
# This agent specializes in providing product information from the vendor's catalog
product_catalog_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="product_catalog_agent",
    description="External vendor's product catalog agent that provides product information and availability.",
    instruction="""
        You are a product catalog specialist from an external vendor.
        When asked about products, use the get_product_info tool to fetch data from the catalog.
        Provide clear, accurate product information including price, availability, and specs.
        If asked about multiple products, look up each one.
        Be professional and helpful.
    """
,
```

```

    tools=[get_product_info], # Register the product lookup tool
)

print("✓ Product Catalog Agent created successfully!")
print("  Model: gemini-2.5-flash-lite")
print("  Tool: get_product_info()")
print("  Ready to be exposed via A2A...")

```

 Product Catalog Agent created successfully!  
 Model: gemini-2.5-flash-lite  
 Tool: get\_product\_info()  
 Ready to be exposed via A2A...

## Section 2: Expose the Product Catalog Agent via A2A

Now we'll use ADK's `to_a2a()` function to make our Product Catalog Agent accessible to other agents.

What `to_a2a()` does:

-  Wraps your agent in an A2A-compatible server (FastAPI/Starlette)
-  Auto-generates an **agent card** that includes:
  - Agent name, description, and version
  - Skills (your tools/functions become "skills" in A2A)
  - Protocol version and endpoints
  - Input/output modes
-  Serves the agent card at `/ .well-known/agent-card.json` (standard A2A path)
-  Handles all A2A protocol details (request/response formatting, task endpoints)

This is the **easiest way** to expose an ADK agent via A2A!

### Key Concept: Agent Cards

An **agent card** is a JSON document that serves as a "business card" for your agent. It describes:

- What the agent does (name, description, version)
- What capabilities it has (skills, tools, functions)
- How to communicate with it (URL, protocol version, endpoints)

Every A2A agent must publish its agent card at the standard path:

`/well-known/agent-card.json`

Think of it as the "contract" that tells other agents how to work with your agent.

 **Learn more:**

- [Exposing Agents with ADK](#)
- [A2A Protocol Specification](#)

```
# Convert the product catalog agent to an A2A-compatible application
# This creates a FastAPI/Starlette app that:
#   1. Serves the agent at the A2A protocol endpoints
#   2. Provides an auto-generated agent card
#   3. Handles A2A communication protocol
product_catalog_a2a_app = to_a2a(
    product_catalog_agent, port=8001 # Port where this agent will be
served
)

print("✅ Product Catalog Agent is now A2A-compatible!")
print("  Agent will be served at: http://localhost:8001")
print("  Agent card will be at:")
http://localhost:8001/.well-known/agent-card.json")
print("  Ready to start the server...")
```

```
✅ Product Catalog Agent is now A2A-compatible!
Agent will be served at: http://localhost:8001
Agent card will be at:
http://localhost:8001/.well-known/agent-card.json
Ready to start the server...
```



## Section 3: Start the Product Catalog Agent Server

We'll start the Product Catalog Agent server in the **background** using `uvicorn`, so it can serve requests from other agents.

Why run in background?

- The server needs to keep running while we create and test the Customer Support Agent
- This simulates a real-world scenario where different agents run as separate services
- In production, the vendor would host this on their infrastructure

```
# First, let's save the product catalog agent to a file that uvicorn can import
product_catalog_agent_code = '''
import os
from google.adk.agents import LlmAgent
from google.adk.a2a.utils.agent_to_a2a import to_a2a
from google.adk.models.google_llm import Gemini
from google.genai import types

retry_config = types.HttpRetryOptions(
    attempts=5, # Maximum retry attempts
    exp_base=7, # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors
)

def get_product_info(product_name: str) -> str:
    """Get product information for a given product."""
    product_catalog = {
        "iphone 15 pro": "iPhone 15 Pro, $999, Low Stock (8 units), 128GB, Titanium finish",
        "samsung galaxy s24": "Samsung Galaxy S24, $799, In Stock (31 units), 256GB, Phantom Black",
        "dell xps 15": "Dell XPS 15, $1,299, In Stock (45 units), 15.6\" display, 16GB RAM, 512GB SSD",
        "macbook pro 14": "MacBook Pro 14\\\", $1,999, In Stock (22 units), M3 Pro chip, 18GB RAM, 512GB SSD",
    }
    return product_catalog.get(product_name, "Product not found")'''
```

```

        "sony wh-1000xm5": "Sony WH-1000XM5 Headphones, $399, In Stock (67
units), Noise-canceling, 30hr battery",
        "ipad air": "iPad Air, $599, In Stock (28 units), 10.9\" display,
64GB",
        "lg ultrawide 34": "LG UltraWide 34\" Monitor, $499, Out of
Stock, Expected: Next week",
    }

product_lower = product_name.lower().strip()

if product_lower in product_catalog:
    return f"Product: {product_catalog[product_lower]}"
else:
    available = ", ".join([p.title() for p in product_catalog.keys()])
    return f"Sorry, I don't have information for {product_name}.

Available products: {available}"

product_catalog_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
    name="product_catalog_agent",
    description="External vendor's product catalog agent that provides
product information and availability.",
    instruction="""
    You are a product catalog specialist from an external vendor.
    When asked about products, use the get_product_info tool to fetch data
from the catalog.
    Provide clear, accurate product information including price,
availability, and specs.
    If asked about multiple products, look up each one.
    Be professional and helpful.
    """,
    tools=[get_product_info]
)

# Create the A2A app
app = to_a2a(product_catalog_agent, port=8001)
...

# Write the product catalog agent to a temporary file
```

```
with open("/tmp/product_catalog_server.py", "w") as f:
    f.write(product_catalog_agent_code)

print("📝 Product Catalog agent code saved to
/tmp/product_catalog_server.py")

# Start uvicorn server in background
# Note: We redirect output to avoid cluttering the notebook
server_process = subprocess.Popen(
    [
        "uvicorn",
        "product_catalog_server:app", # Module:app format
        "--host",
        "localhost",
        "--port",
        "8001",
    ],
    cwd="/tmp", # Run from /tmp where the file is
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    env={**os.environ}, # Pass environment variables (including
GOOGLE_API_KEY)
)

print("🚀 Starting Product Catalog Agent server...")
print("  Waiting for server to be ready...")

# Wait for server to start (poll until it responds)
max_attempts = 30
for attempt in range(max_attempts):
    try:
        response = requests.get(
            "http://localhost:8001/.well-known/agent-card.json", timeout=1
        )
        if response.status_code == 200:
            print(f"\n✅ Product Catalog Agent server is running!")
            print(f"  Server URL: http://localhost:8001")
            print(f"  Agent card:
http://localhost:8001/.well-known/agent-card.json")
            break
    except requests.exceptions.RequestException as e:
        print(f"Attempt {attempt+1}/{max_attempts} failed: {e}")
```

```
except requests.exceptions.RequestException:
    time.sleep(5)
    print(".", end="", flush=True)
else:
    print("\n⚠️ Server may not be ready yet. Check manually if needed.")

# Store the process so we can stop it later
globals()["product_catalog_server_process"] = server_process
```

```
📝 Product Catalog agent code saved to /tmp/product_catalog_server.py
🚀 Starting Product Catalog Agent server...
Waiting for server to be ready...

.....
✓ Product Catalog Agent server is running!
Server URL: http://localhost:8001
Agent card: http://localhost:8001/.well-known/agent-card.json
```

## 🔍 View the Auto-Generated Agent Card

The `to_a2a()` function automatically created an **agent card** that describes the Product Catalog Agent's capabilities. Let's take a look!

```
# Fetch the agent card from the running server
try:
    response = requests.get(
        "http://localhost:8001/.well-known/agent-card.json", timeout=5
    )

    if response.status_code == 200:
        agent_card = response.json()
        print("📋 Product Catalog Agent Card:")
        print(json.dumps(agent_card, indent=2))

        print("\n✨ Key Information:")
        print(f"  Name: {agent_card.get('name')}")
        print(f"  Description: {agent_card.get('description')}")
```

```

        print(f"    URL: {agent_card.get('url')}")
        print(f"    Skills: {len(agent_card.get('skills', []))}")
capabilities exposed")
else:
    print(f"✗ Failed to fetch agent card: {response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"✗ Error fetching agent card: {e}")
    print("    Make sure the Product Catalog Agent server is running
(previous cell)")

```

 Product Catalog Agent Card:

```
{
    "capabilities": {},
    "defaultInputModes": [
        "text/plain"
    ],
    "defaultOutputModes": [
        "text/plain"
    ],
    "description": "External vendor's product catalog agent that provides
product information and availability.",
    "name": "product_catalog_agent",
    "preferredTransport": "JSONRPC",
    "protocolVersion": "0.3.0",
    "skills": [
        {
            "description": "External vendor's product catalog agent that
provides product information and availability. \n    I am a product
catalog specialist from an external vendor.\n    When asked about
products, use the get_product_info tool to fetch data from the catalog.\n    Provide clear, accurate product information including price, availability,
and specs.\n    If asked about multiple products, look up each one.\n    Be professional and helpful.\n    ",
            "id": "product_catalog_agent",
            "name": "model",
            "tags": [
                "llm"
            ]
        }
    ]
}
```

```

    },
    {
      "description": "Get product information for a given product.",
      "id": "product_catalog_agent-get_product_info",
      "name": "get_product_info",
      "tags": [
        "llm",
        "tools"
      ]
    }
  ],
  "supportsAuthenticatedExtendedCard": false,
  "url": "http://localhost:8001",
  "version": "0.0.1"
}

```

💡 Key Information:

Name: product\_catalog\_agent

Description: External vendor's product catalog agent that provides product information and availability.

URL: <http://localhost:8001>

Skills: 2 capabilities exposed

## 🎧 Section 4: Create the Customer Support Agent (Consumer)

Now we'll create a **Customer Support Agent** that consumes the Product Catalog Agent using A2A.

How it works:

1. We use RemoteA2aAgent to create a **client-side proxy** for the Product Catalog Agent
2. The Customer Support Agent can use the Product Catalog Agent like any other tool
3. ADK handles all the A2A protocol communication behind the scenes

This demonstrates the power of A2A: **agents can collaborate as if they were local!**

How RemoteA2aAgent works:

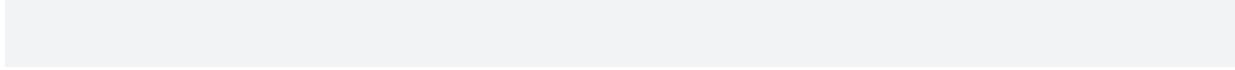
- It's a **client-side proxy** that reads the remote agent's card
- Translates sub-agent calls into A2A protocol requests (HTTP POST to /tasks)
- Handles all the protocol details so you just use it like a regular sub-agent

 **Learn more:**

- [Consuming Remote Agents with ADK](#)
- [What is A2A?](#)

```
# Create a RemoteA2aAgent that connects to our Product Catalog Agent
# This acts as a client-side proxy - the Customer Support Agent can use it
like a local agent
remote_product_catalog_agent = RemoteA2aAgent(
    name="product_catalog_agent",
    description="Remote product catalog agent from external vendor that
provides product information.",
    # Point to the agent card URL - this is where the A2A protocol metadata
lives
    agent_card=f"http://localhost:8001{AGENT_CARD_WELL_KNOWN_PATH}",
)

print("✅ Remote Product Catalog Agent proxy created!")
print(f"  Connected to: http://localhost:8001")
print(f"  Agent card: http://localhost:8001{AGENT_CARD_WELL_KNOWN_PATH}")
print("  The Customer Support Agent can now use this like a local
sub-agent!")
```



✓ Remote Product Catalog Agent proxy created!  
Connected to: http://localhost:8001  
Agent card: http://localhost:8001/.well-known/agent-card.json  
The Customer Support Agent can now use this like a local sub-agent!

```
# Now create the Customer Support Agent that uses the remote Product
Catalog Agent
customer_support_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite",
retry_options=retry_config),
```

```
name="customer_support_agent",
description="A customer support assistant that helps customers with
product inquiries and information.",
instruction="""
You are a friendly and professional customer support agent.

When customers ask about products:
1. Use the product_catalog_agent sub-agent to look up product
information
2. Provide clear answers about pricing, availability, and
specifications
3. If a product is out of stock, mention the expected availability
4. Be helpful and professional!

Always get product information from the product_catalog_agent before
answering customer questions.

""",
sub_agents=[remote_product_catalog_agent], # Add the remote agent as
a sub-agent!
)

print("✓ Customer Support Agent created!")
print("  Model: gemini-2.5-flash-lite")
print("  Sub-agents: 1 (remote Product Catalog Agent via A2A)")
print("  Ready to help customers!")
```

```
✓ Customer Support Agent created!
Model: gemini-2.5-flash-lite
Sub-agents: 1 (remote Product Catalog Agent via A2A)
Ready to help customers!
```

## Section 5: Test A2A Communication

Let's test the agent-to-agent communication! We'll ask the Customer Support Agent about products, and it will communicate with the Product Catalog Agent via A2A.

What happens behind the scenes:

1. Customer asks Support Agent a question about a product
2. Support Agent realizes it needs product info
3. Support Agent calls the `remote_product_catalog_agent` (`RemoteA2aAgent`)
4. ADK sends an A2A protocol request to `http://localhost:8001`
5. Product Catalog Agent processes the request and responds
6. Support Agent receives the response and continues
7. Customer gets the final answer

All of this happens **transparently** - the Support Agent doesn't need to know it's talking to a remote agent!

```
async def test_a2a_communication(user_query: str):  
    """  
    Test the A2A communication between Customer Support Agent and Product  
    Catalog Agent.  
    """
```

*This function:*

1. Creates a new session for this conversation
2. Sends the query to the Customer Support Agent
3. Support Agent communicates with Product Catalog Agent via A2A
4. Displays the response

*Args:*

```
    user_query: The question to ask the Customer Support Agent  
    """  
  
    # Setup session management (required by ADK)  
    session_service = InMemorySessionService()  
  
    # Session identifiers  
    app_name = "support_app"  
    user_id = "demo_user"  
    # Use unique session ID for each test to avoid conflicts  
    session_id = f"demo_session_{uuid.uuid4().hex[:8]}"  
  
    # CRITICAL: Create session BEFORE running agent (synchronous, not  
    #           async!)  
    # This pattern matches the deployment notebook exactly  
    session = await session_service.create_session()
```

```

        app_name=app_name, user_id=user_id, session_id=session_id
    )

# Create runner for the Customer Support Agent
# The runner manages the agent execution and session state
runner = Runner(
    agent=customer_support_agent, app_name=app_name,
session_service=session_service
)

# Create the user message
# This follows the same pattern as the deployment notebook
test_content = types.Content(parts=[types.Part(text=user_query)])

# Display query
print(f"\n👤 Customer: {user_query}")
print(f"\n🎧 Support Agent response:")
print("-" * 60)

# Run the agent asynchronously (handles streaming responses and A2A
communication)
async for event in runner.run_async(
    user_id=user_id, session_id=session_id, new_message=test_content
):
    # Print final response only (skip intermediate events)
    if event.is_final_response() and event.content:
        for part in event.content.parts:
            if hasattr(part, "text"):
                print(part.text)

print("-" * 60)

# Run the test
print("📝 Testing A2A Communication...\n")
await test_a2a_communication("Can you tell me about the iPhone 15 Pro? Is
it in stock?")

```

 Testing A2A Communication...

 Customer: Can you tell me about the iPhone 15 Pro? Is it in stock?

 Support Agent response:

---

```
INFO:google_adk.google.adk.models.google_llm:Sending out request, model: gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False  
INFO:google_adk.google.adk.models.google_llm:Response received from the model.
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
```

```
INFO:google_adk.google.adk.agents.remote_a2a_agent:Successfully resolved remote A2A agent: product_catalog_agent
```

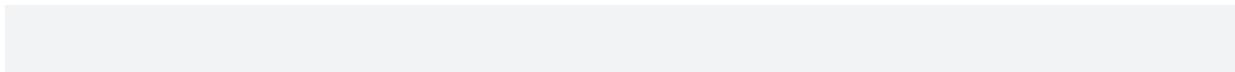
The iPhone 15 Pro is available for \$999. We currently have low stock, with only 8 units remaining. It features a 128GB storage capacity and a titanium finish.

---

## Try More Examples

Let's test a few more scenarios to see A2A communication in action!

```
# Test comparing multiple products
await test_a2a_communication(
    "I'm looking for a laptop. Can you compare the Dell XPS 15 and MacBook Pro 14 for me?"
)
```



```
INFO:google_adk.google.adk.models.google_llm:Sending out request, model: gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False
```

👤 Customer: I'm looking for a laptop. Can you compare the Dell XPS 15 and MacBook Pro 14 for me?

🎧 Support Agent response:

---

```
INFO:google_adk.google.adk.models.google_llm:Response received from the model.
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
```

Here's a comparison of the two laptops:

\*\*Dell XPS 15:\*\*  
\* \*\*Price:\*\* \$1,299  
\* \*\*Availability:\*\* In Stock (45 units)  
\* \*\*Display:\*\* 15.6"  
\* \*\*RAM:\*\* 16GB  
\* \*\*Storage:\*\* 512GB SSD

\*\*MacBook Pro 14":\*\*  
\* \*\*Price:\*\* \$1,999  
\* \*\*Availability:\*\* In Stock (22 units)  
\* \*\*Chip:\*\* M3 Pro  
\* \*\*RAM:\*\* 18GB  
\* \*\*Storage:\*\* 512GB SSD

The MacBook Pro 14" is more expensive but features a more advanced chip (M3 Pro) and slightly more RAM. The Dell XPS 15 offers a larger display at a more affordable price point.

---

```
# Test specific product inquiry
await test_a2a_communication(
    "Do you have the Sony WH-1000XM5 headphones? What's the price?"
)
```

```
INFO:google_adk.google.adk.models.google_llm:Sending out request, model: gemini-2.5-flash-lite, backend: GoogleLLMVariant.GEMINI_API, stream: False
```

👤 Customer: Do you have the Sony WH-1000XM5 headphones? What's the price?

🎧 Support Agent response:

---

```
INFO:google_adk.google.adk.models.google_llm:Response received from the model.
```

```
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
```

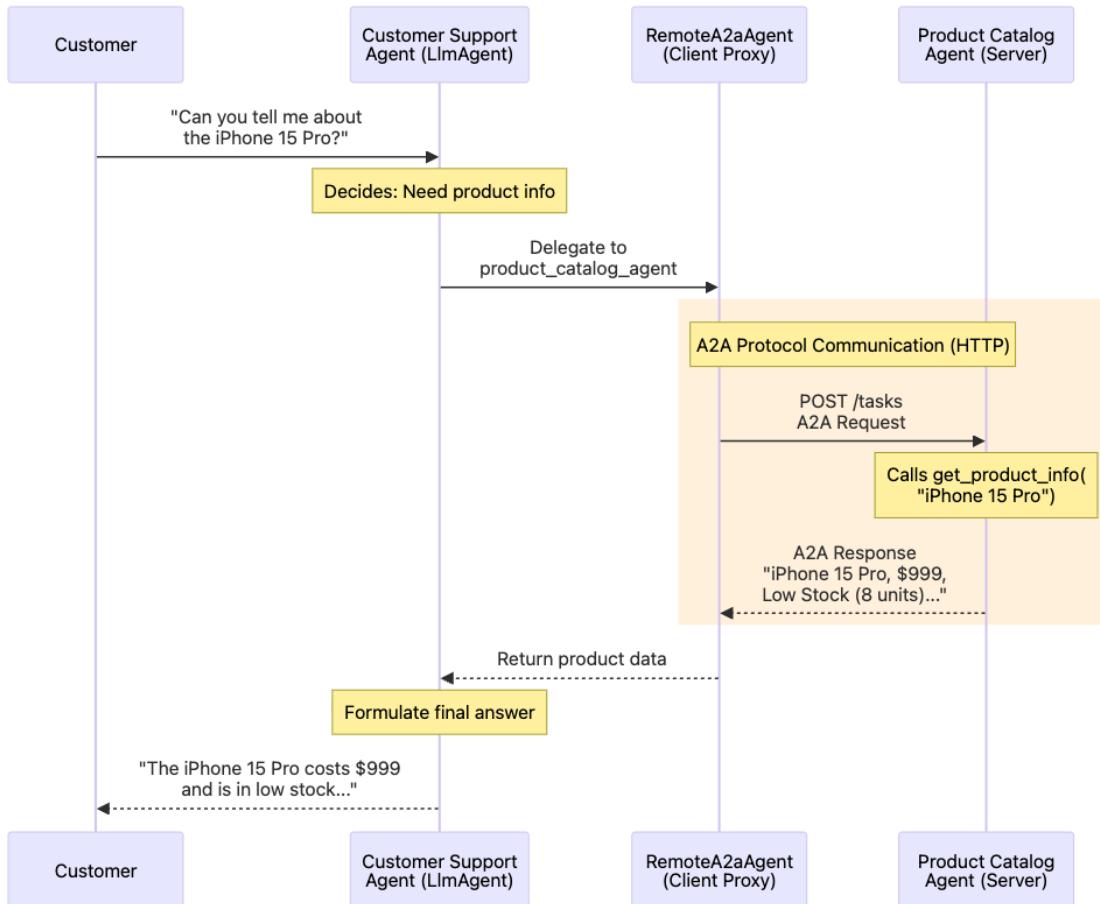
The Sony WH-1000XM5 headphones are currently in stock and available for \$399. They feature noise-canceling technology and a 30-hour battery life.

---

## 🔍 Section 6: Understanding What Just Happened

### A2A Communication Flow

When you ran the tests above, here's the detailed step-by-step flow of how the agents communicated:



### A2A Protocol Communication:

Behind the scenes, here's what happens at the protocol level:

- **RemoteA2aAgent** sends HTTP POST requests to the /tasks endpoint on `http://localhost:8001`
- Request and response data follow the [A2A Protocol Specification](#)
- Data is exchanged in standardized JSON format
- The protocol ensures any A2A-compatible agent (regardless of language/framework) can communicate

This standardization is what makes cross-organization, cross-language agent communication possible!

---

### What happened:

1. **Customer** asked about the iPhone 15 Pro
2. **Customer Support Agent** (LlmAgent) received the question and decided it needs product information
3. **Support Agent** delegated to the product\_catalog\_agent sub-agent
4. **RemoteA2aAgent** (client-side proxy) translated this into an A2A protocol request
5. The A2A request was sent over HTTP to `http://localhost:8001` (highlighted in yellow)
6. **Product Catalog Agent** (server) received the request and called  
`get_product_info("iPhone 15 Pro")`
7. **Product Catalog Agent** returned the product information via A2A response
8. **RemoteA2aAgent** received the response and passed it back to the Support Agent
9. **Support Agent** formulated a final answer with the product details
10. **Customer** received the complete, helpful response

### Key Benefits Demonstrated

1. **Transparency:** Support Agent doesn't "know" Product Catalog Agent is remote
2. **Standard Protocol:** Uses A2A standard - any A2A-compatible agent works
3. **Easy Integration:** Just one line: `sub_agents=[remote_product_catalog_agent]`
4. **Separation of Concerns:** Product data lives in Catalog Agent (vendor), support logic in Support Agent (your company)

### Real-World Applications

This pattern enables:

- **Microservices:** Each agent is an independent service
- **Third-party Integration:** Consume agents from external vendors (e.g., product catalogs, payment processors)
- **Cross-language:** Product Catalog Agent could be Java, Support Agent Python
- **Specialized Teams:** Vendor maintains catalog, your team maintains support agent
- **Cross-Organization:** Vendor hosts catalog on their infrastructure, you integrate via A2A

## Next Steps and Learning Resources

### Enhancement Ideas

Now that you understand A2A basics, try extending this example:

#### 1. Add More Agents:

- Create an **Inventory Agent** that checks stock levels and restocking schedules
- Create a **Shipping Agent** that provides delivery estimates and tracking
- Have Customer Support Agent coordinate all three via A2A

#### 2. Real Data Sources:

- Replace mock product catalog with real database (PostgreSQL, MongoDB)
- Add real inventory tracking system integration
- Connect to real payment gateway APIs

#### 3. Advanced A2A Features:

- Implement authentication between agents (API keys, OAuth)
- Add error handling and retries for network failures
- Use the alternative adk api\_server --a2a approach

#### 4. Deploy to Production:

- Deploy Product Catalog Agent to Agent Engine
- Update agent card URL to point to production server (e.g., <https://vendor-catalog.example.com>)
- Consumer agents can now access it over the internet!

### Documentation

#### A2A Protocol:

- [Official A2A Protocol Website](#)
- [A2A Protocol Specification](#)

#### ADK A2A Guides:

- [Introduction to A2A in ADK](#)
- [Exposing Agents Quickstart](#)
- [Consuming Agents Quickstart](#)

#### Other Deployment Options:

- [Deploy ADK Agents to Cloud Run](#)
  - [Deploy to Agent Engine](#)
  - [Deploy to GKE](#)
- 



## Summary - A2A Communication Patterns

### Key Takeaways

In this notebook, you learned how to build multi-agent systems with A2A:

- **A2A Protocol:** Standardized protocol for agent-to-agent communication across networks and frameworks
  - **Exposing Agents:** Use `to_a2a()` to make your agents accessible to others with auto-generated agent cards
  - **Consuming Agents:** Use `RemoteA2aAgent` to integrate remote agents as if they were local sub-agents
  - **Use Cases:** Best for microservices architectures, cross-team integrations, and third-party agent consumption
- 



## Congratulations! You're an A2A Expert

You've successfully learned how to build multi-agent systems using the A2A protocol!

You now know how to expose agents as services, consume remote agents, and build collaborative multi-agent systems that can scale across teams and organizations.



### Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

## Learn More

Refer to the following documentation to learn more:

- [ADK Documentation](#)
- [A2A Protocol Official Website](#)
- [A2A Tutorials](#)
- [Introduction to A2A in ADK](#)
- [Exposing Agents Quickstart](#)
- [Consuming Agents Quickstart](#)

## Next Steps

Now that you understand A2A communication, you can build complex multi-agent systems where specialized agents collaborate to solve real-world problems. Consider deploying your agents to production using Cloud Run or Agent Engine to make them accessible over the internet!

Ready for more? Explore advanced ADK features like custom agents, streaming, and production deployment patterns!

---

Authors
<a href="#">Lavi Nigam</a>

Copyright 2025 Google LLC.

```
# @title 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
#
# https://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.
```



## Deploy ADK Agent to Vertex AI Agent Engine

Welcome to the final day of the Kaggle 5-day Agents course!

In the previous notebook you learned how to use Agent2Agent Protocol to make your agents interoperable.

Now, let's take the final step: deploying your agents to production using [Vertex AI Agent Engine](#).



## Scaling Your Agent

You've built an amazing AI agent. It works perfectly on your machine. You can chat with it, it responds intelligently, and everything seems ready. But there's a problem.

### Your agent is not publicly available!

It only lives in your notebook and development environment. When you stop your notebook session, it stops working. Your teammates can't access it. Your users can't interact with it. And this is precisely why we need to deploy the agents!

### ⌚ What You'll Learn

In this notebook, you'll:

- Build a production-ready ADK agent
- Deploy your agent to [Vertex AI Agent Engine](#) using the ADK CLI
- Test your deployed agent with Python SDK

- Monitor and manage deployed agents in Google Cloud Console
- Understand how to add Memory to your Agent using Vertex AI Memory Bank
- Understand cost management and cleanup best practices

## !! Please Read

  **Note:** No submission required! This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

 **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads "**Waiting for the next available notebook**". The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

 **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time.

[See FAQ on 429 errors for more information.](#)

For help: Ask questions on the [Kaggle Discord](#) server.

## Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks [in the documentation](#).

Here's how to get started:

### 1. Verify Your Account (Required)

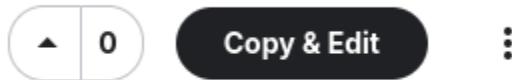
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your [Kaggle settings](#).

### 2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the Copy and Edit button in the top-right corner.

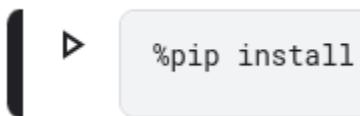


This creates a private copy of the notebook just for you.

### 3. Run Code Cells

Once you have your copy, you can run code.

Click the Run button next to any code cell to execute it.



Run the cells in order from top to bottom.

### 4. If You Get Stuck

To restart: Select Factory reset from the Run menu.

For help: Ask questions on the [Kaggle Discord](#) server.

## Section 1: Setup

### 1.1: Important: Prerequisites

This notebook requires a **Google Cloud account** to deploy agents to Vertex AI Agent Engine.

If you don't have a GCP account yet:

- Step 1. Create a free Google Cloud account - [Sign up here](#)

- New users get **\$300 in free credits** valid for 90 days on Google Cloud
- No charges during the free trial period
  - ✓ Step 2. **Enable billing on your account** - Required even for free trial
- A credit card is needed for verification
- You won't be charged unless you explicitly upgrade
- This demo stays within the free tier of Agent Engine if you clean up resources promptly

✓ Step 3. **Understand the free trial** - Know what's included

- Check [free trial details of Google Cloud](#)
- Review [common questions about the free trial for Google Cloud](#)

💡 **Quick Setup Guide:** Watch this [3-minute setup video](#) for a walkthrough

## 1.2: Import components

Now, import the specific components you'll need for this notebook. This keeps your code organized and ensures we have access to the necessary building blocks.

```
import os
import random
import time
import vertexai
from kaggle_secrets import UserSecretsClient
from vertexai import agent_engines

print("✓ Imports completed successfully")
```

✓ Imports completed successfully

## 1.3: Add Cloud Credentials to Secrets

1. In the top menu bar of the notebook editor, select Add-ons then Google Cloud SDK.
2. Click on Link Account
3. Select your Google Cloud Account
4. Attach to the notebook

This cell retrieves your Google Cloud credentials from Kaggle Secrets and configures them for use. These credentials allow the notebook to authenticate with Google Cloud services like Vertex AI Agent Engine.

```
# Set up Cloud Credentials in Kaggle
user_secrets = UserSecretsClient()
user_credential = user_secrets.get_gcloud_credential()
user_secrets.set_tensorflow_credential(user_credential)

print("✅ Cloud credentials configured")
```

✅ Cloud credentials configured

#### 1.4: Set your PROJECT\_ID

**Important:** Make sure to replace "your-project-id" with your actual Google Cloud project ID.

You can find your project ID in the [Google Cloud Console](#).

```
## Set your PROJECT_ID
PROJECT_ID = "your-project-id" # TODO: Replace with your project ID
os.environ[ "GOOGLE_CLOUD_PROJECT" ] = PROJECT_ID

if PROJECT_ID == "your-project-id" or not PROJECT_ID:
    raise ValueError("⚠️ Please replace 'your-project-id' with your actual
Google Cloud Project ID.")

print(f"✅ Project ID set to: {PROJECT_ID}")
```

✅ Project ID set to: some-google-cloud-project

#### 1.5: Enable Google Cloud APIs

For this tutorial, you'll need to enable the following APIs in the Google Cloud Console.

- Vertex AI API
- Cloud Storage API
- Cloud Logging API
- Cloud Monitoring API
- Cloud Trace API
- Telemetry API

You can [use this link to open the Google Cloud Console](#) and follow the steps there to enable these APIs.

---

## Section 2: Create Your Agent with ADK

Before we deploy, we need a functional agent to host. We'll build a **Weather Assistant** designed to serve as our sample agent.

This agent is optimized for production testing with the following configuration:

- **Model:** Uses gemini-2.5-flash-lite for low latency and cost-efficiency.
- **Tools:** Includes a get\_weather function to demonstrate tool execution.
- **Persona:** Responds conversationally to prove the instruction-following capabilities.

This demonstrates the foundational ADK architecture we are about to package: **Agent + Tools + Instructions**.

We'll create the following files and directory structure:

```
sample_agent/
├── agent.py                  # The logic
├── requirements.txt          # The libraries
├── .env                       # The secrets/config
└── .agent_engine_config.json # The hardware specs
```

## 2.1: Create agent directory

We need a clean workspace to package our agent for deployment. We will create a directory named `sample_agent`.

All necessary files - including the agent code, dependencies, and configuration—will be written into this folder to prepare it for the `adk deploy` command.

```
## Create simple agent - all code for the agent will live in this directory
!mkdir -p sample_agent

print(f"✅ Sample Agent directory created")
```

```
✅ Sample Agent directory created
```

## 2.2: Create requirements file

The Agent Engine builds a dedicated environment for your agent. To ensure it runs correctly, we must declare our dependencies.

We will write a `requirements.txt` file containing the Python packages needed for the agent.

```
%>writefile sample_agent/requirements.txt

google-adk
opentelemetry-instrumentation-google-genai
```

```
Writing sample_agent/requirements.txt
```

## 2.3: Create environment configuration

We need to provide the agent with the necessary cloud configuration settings.

We will write a .env file that sets the cloud location to global and explicitly enables the Vertex AI backend for the ADK SDK.

```
%%writefile sample_agent/.env

#
https://cloud.google.com/vertex-ai/generative-ai/docs/learn/locations#global-endpoint
GOOGLE_CLOUD_LOCATION="global"

# Set to 1 to use Vertex AI, or 0 to use Google AI Studio
GOOGLE_GENAI_USE_VERTEXAI=1
```

```
Writing sample_agent/.env
```

### Configuration explained:

- GOOGLE\_CLOUD\_LOCATION="global" - Uses the global endpoint for Gemini API calls
- GOOGLE\_GENAI\_USE\_VERTEXAI=1 - Configures ADK to use Vertex AI instead of Google AI Studio

## 2.4: Create agent code

We will now generate the agent.py file. This script defines the behavior of our **Weather Assistant**.

Agent Configuration:

- 🧠 Model: Uses gemini-2.5-flash-lite for low latency and cost-efficiency.
- 🔨 Tools: Accesses a get\_weather function to retrieve data.
- 📝 Instructions: Follows a system prompt to identify cities and respond in a friendly tone.

```
%%writefile sample_agent/agent.py
```

```
from google.adk.agents import Agent
import vertexai
import os

vertexai.init(
    project=os.environ["GOOGLE_CLOUD_PROJECT"],
    location=os.environ["GOOGLE_CLOUD_LOCATION"],
)

def get_weather(city: str) -> dict:
    """
    Returns weather information for a given city.

    This is a TOOL that the agent can call when users ask about weather.
    In production, this would call a real weather API (e.g.,
    OpenWeatherMap).

    For this demo, we use mock data.

    Args:
        city: Name of the city (e.g., "Tokyo", "New York")

    Returns:
        dict: Dictionary with status and weather report or error message
    """
    # Mock weather database with structured responses
    weather_data = {
        "san francisco": {"status": "success", "report": "The weather in San Francisco is sunny with a temperature of 72°F (22°C)."},
        "new york": {"status": "success", "report": "The weather in New York is cloudy with a temperature of 65°F (18°C)."},
        "london": {"status": "success", "report": "The weather in London is rainy with a temperature of 58°F (14°C)."},
        "tokyo": {"status": "success", "report": "The weather in Tokyo is clear with a temperature of 70°F (21°C)."},
        "paris": {"status": "success", "report": "The weather in Paris is partly cloudy with a temperature of 68°F (20°C)."}
    }

    city_lower = city.lower()
    if city_lower in weather_data:
```

```

        return weather_data[city_lower]
    else:
        available_cities = ", ".join([c.title() for c in
weather_data.keys()])
        return {
            "status": "error",
            "error_message": f"Weather information for '{city}' is not
available. Try: {available_cities}"
        }

root_agent = Agent(
    name="weather_assistant",
    model="gemini-2.5-flash-lite", # Fast, cost-effective Gemini model
    description="A helpful weather assistant that provides weather
information for cities.",
    instruction="""
    You are a friendly weather assistant. When users ask about the
    weather:

    1. Identify the city name from their question
    2. Use the get_weather tool to fetch current weather information
    3. Respond in a friendly, conversational tone
    4. If the city isn't available, suggest one of the available cities

    Be helpful and concise in your responses.
    """
,
    tools=[get_weather]
)

```

Writing sample\_agent/agent.py

---

## Section 3: Deploy to Agent Engine

ADK supports multiple deployment platforms. Learn more in the [ADK deployment documentation](#).

You'll be deploying to [Vertex AI Agent Engine](#) in this notebook.

## ◆ Vertex AI Agent Engine

- **Fully managed** service specifically for AI agents
- **Auto-scaling** with session management built-in
- **Easy deployment** using [Agent Starter Pack](#)
-  [Deploy to Agent Engine Guide](#)

**Note:** To help you get started with the runtime, Agent Engine offers a monthly free tier, which you can learn more about in the [documentation](#). The agent deployed in this notebook should stay within the free tier if cleaned up promptly. Note that you can incur costs if the agent is left running.

## 🚢 Other Deployment Options

### ◆ Cloud Run

- Serverless, easiest to start
- Perfect for demos and small-to-medium workloads
-  [Deploy to Cloud Run Guide](#)

### ◆ Google Kubernetes Engine (GKE)

- Full control over containerized deployments
- Best for complex multi-agent systems
-  [Deploy to GKE Guide](#)

#### 3.1: Create deployment configuration

The `.agent_engine_config.json` file controls the deployment settings.

```
%%writefile sample_agent/.agent_engine_config.json
{
    "min_instances": 0,
    "max_instances": 1,
    "resource_limits": {"cpu": "1", "memory": "1Gi"}
}
```

```
Writing sample_agent/.agent_engine_config.json
```

### Configuration explained:

- "min\_instances": 0 - Scales down to zero when not in use (saves costs)
- "max\_instances": 1 - Maximum of 1 instance running (sufficient for this demo)
- "cpu": "1" - 1 CPU core per instance
- "memory": "1Gi" - 1 GB of memory per instance

These settings keep costs minimal while providing adequate resources for our weather agent.

### 3.2: Select deployment region

Agent Engine is available in specific regions. We'll randomly select one for this demo.

```
regions_list = ["europe-west1", "europe-west4", "us-east4", "us-west1"]
deployed_region = random.choice(regions_list)

print(f"✓ Selected deployment region: {deployed_region}")
```

```
✓ Selected deployment region: europe-west4
```

### About regions:

Agent Engine is available in multiple regions. For production:

- Choose a region close to your users for lower latency
- Consider data residency requirements
- Check the [Agent Engine locations documentation](#)

### 3.3: Deploy the agent

This uses the ADK CLI to deploy your agent to Agent Engine.

```
!adk deploy agent_engine --project=$PROJECT_ID --region=$deployed_region
sample_agent
--agent_engine_config_file=sample_agent/.agent_engine_config.json
```

```
Staging all files in: /kaggle/working/sample_agent_tmp20251113_213458
Copying agent source code...
Copying agent source code complete.
Resolving files and dependencies...
Reading agent engine config from sample_agent/.agent_engine_config.json
Reading environment variables from /kaggle/working/sample_agent/.env
Ignoring GOOGLE_CLOUD_LOCATION in .env as '--region' was explicitly passed
and takes precedence
Initializing Vertex AI...
Vertex AI initialized.
Created sample_agent_tmp20251113_213458/agent_engine_app.py
Files and dependencies resolved
Deploying to agent engine...
INFO:vertexai_genai.agentengines:Creating in-memory tarfile of
source_packages
INFO:vertexai_genai.agentengines:Using agent framework: google-adk
INFO:vertexai_genai.agentengines:View progress and logs at
https://console.cloud.google.com/logs/query?project=koverholt-devrel-35571
6.
INFO:vertexai_genai.agentengines:Agent Engine created. To use it in
another session:
INFO:vertexai_genai.agentengines:agent_engine=client.agent_engines.get(nam
e='projects/964731510884/locations/europe-west4/reasoningEngines/830389884
1057329152')
✓ Created agent engine:
projects/964731510884/locations/europe-west4/reasoningEngines/830389884105
7329152
```

```
Cleaning up the temp folder: sample_agent_tmp20251113_213458
```

### What just happened:

The adk deploy agent\_engine command:

1. Packages your agent code (sample\_agent/ directory)
2. Uploads it to Agent Engine
3. Creates a containerized deployment
4. Outputs a resource name like:  
projects/PROJECT\_NUMBER/locations/REGION/reasoningEngines/ID

**Note:** Deployment typically takes 2-5 minutes.

---



## Section 4: Retrieve and Test Your Deployed Agent

### 4.1: Retrieve the deployed agent

After deploying with the CLI, we need to retrieve the agent object to interact with it.

```
# Initialize Vertex AI
vertexai.init(project=PROJECT_ID, location=deployed_region)

# Get the most recently deployed agent
agents_list = list(agent_engines.list())
if agents_list:
    remote_agent = agents_list[0] # Get the first (most recent) agent
    client = agent_engines
    print(f"✅ Connected to deployed agent: {remote_agent.resource_name}")
else:
    print("❌ No agents found. Please deploy first.")
```

✓ Connected to deployed agent:  
projects/964731510884/locations/europe-west4/reasoningEngines/830389884105  
7329152

### What happened:

This cell retrieves your deployed agent:

1. Initializes the Vertex AI SDK with your project and region
2. Lists all deployed agents in that region
3. Gets the first one (most recently deployed)
4. Stores it as `remote_agent` for testing

### 4.2: Test the deployed agent

Now let's send a query to your deployed agent!

```
async for item in remote_agent.async_stream_query(  
    message="What is the weather in Tokyo?",  
    user_id="user_42",  
):  
    print(item)
```

```
{'model_version': 'gemini-2.5-flash-lite', 'content': {'parts':  
[{'function_call': {'id': 'adk-c0e6f605-1b75-48ff-b4cb-d3755723106f',  
'args': {'city': 'Tokyo'}, 'name': 'get_weather'}}], 'role': 'model'},  
'finish_reason': 'STOP', 'usage_metadata': {'candidates_token_count': 5,  
'candidates_tokens_details': [ {'modality': 'TEXT', 'token_count': 5}],  
'prompt_token_count': 232, 'prompt_tokens_details': [ {'modality': 'TEXT',  
'token_count': 232}], 'total_token_count': 237, 'traffic_type':  
'ON_DEMAND'}, 'avg_logprobs': -0.07416906952857971, 'invocation_id':  
'e-e222dfffb-7198-4cbf-a77a-e991dc76e1a1', 'author': 'weather_assistant',  
'actions': {'state_delta': {}, 'artifact_delta': {}},  
'requested_auth_configs': {}, 'requested_tool_confirmations': {}},  
'long_running_tool_ids': [], 'id': '640dd587-9557-4e93-95e1-5683c05e6016',  
'timestamp': 1763069860.457819}
```

```
{'content': {'parts': [{}{'function_response': {'id': 'adk-c0e6f605-1b75-48ff-b4cb-d3755723106f', 'name': 'get_weather', 'response': {'status': 'success', 'report': 'The weather in Tokyo is clear with a temperature of 70°F (21°C).'}}]}, 'role': 'user'}, 'invocation_id': 'e-e222dfffb-7198-4cbf-a77a-e991dc76e1a1', 'author': 'weather_assistant', 'actions': {'state_delta': {}, 'artifact_delta': {}}, 'requested_auth_configs': {}, 'requested_tool_confirmations': {}}, 'id': '19464e3b-00b8-47e4-a7f6-3ade87d629d5', 'timestamp': 1763069860.859922}{'model_version': 'gemini-2.5-flash-lite', 'content': {'parts': [{}{'text': 'The weather in Tokyo is clear with a temperature of 70°F (21°C).'}]}, 'role': 'model'}, 'finish_reason': 'STOP', 'usage_metadata': {'candidates_token_count': 21, 'candidates_tokens_details': [{}{'modality': 'TEXT', 'token_count': 21}], 'prompt_token_count': 264, 'prompt_tokens_details': [{}{'modality': 'TEXT', 'token_count': 264}], 'total_token_count': 285, 'traffic_type': 'ON_DEMAND'}, 'avg_logprobs': -0.005634209939411708, 'invocation_id': 'e-e222dfffb-7198-4cbf-a77a-e991dc76e1a1', 'author': 'weather_assistant', 'actions': {'state_delta': {}, 'artifact_delta': {}}, 'requested_auth_configs': {}, 'requested_tool_confirmations': {}}, 'id': '23a6a463-e090-4d19-b64c-1b388ae60db2', 'timestamp': 1763069861.018271}
```

### What happened:

This cell tests your deployed agent:

1. Sends the query "What is the weather in Tokyo?"
2. Streams the response from the agent

### Understanding the output:

You'll see multiple items printed:

1. **Function call** - Agent decides to call get\_weather tool
  2. **Function response** - Result from the tool (weather data)
  3. **Final response** - Agent's natural language answer
-

## Section 5: Long-Term Memory with Vertex AI Memory Bank

### What Problem Does Memory Bank Solve?

Your deployed agent has **session memory** - it remembers the conversation while you're chatting.

But once the session ends, it forgets everything. Each new conversation starts from scratch.

#### The problem:

- User tells agent "I prefer Celsius" today
- Tomorrow, user asks about weather → Agent gives Fahrenheit (forgot preference)
- User has to repeat preferences every time

#### What is Vertex AI Memory Bank?

Memory Bank gives your agent **long-term memory across sessions**:

Session Memory	Memory Bank
Single conversation	All conversations
Forgets when session ends	Remembers permanently
"What did I just say?"	"What's my favorite city?"

#### How it works:

1. **During conversations** - Agent uses memory tools to search past facts
2. **After conversations** - Agent Engine extracts key information ("User prefers Celsius")
3. **Next session** - Agent automatically recalls and uses that information

#### Example:

- **Session 1:** User: "I prefer Celsius"
- **Session 2 (days later):** User: "Weather in Tokyo?" → Agent responds in Celsius automatically ✨

## Memory Bank & Your Deployment

Your Agent Engine deployment **provides the infrastructure** for Memory Bank, but it's not enabled by default.

#### To use Memory Bank:

1. Add memory tools to your agent code (`PreloadMemoryTool`)
2. Add a callback to save conversations to Memory Bank
3. Redeploy your agent

Once configured, Memory Bank works automatically - no additional infrastructure needed!

## Learn More

- [\*\*ADK Memory Guide\*\*](#) - Complete guide with code examples
- [\*\*Memory Tools\*\*](#) - `PreloadMemory` and `LoadMemory` documentation
- [\*\*Get started with Memory Bank on ADK\*\*](#) - Sample notebook that demonstrates how to build ADK agents with memory

---

## Section 6: Cleanup

 **IMPORTANT:** Prevent unexpected charges: Always delete resources when done testing!

## Cost Reminders

As a reminder, leaving the agent running can incur costs. Agent Engine offers a monthly free tier, which you can learn more about in the [documentation](#).

### Always delete resources when done testing!

When you're done testing and querying your deployed agent, it's recommended to delete your remote agent to avoid incurring additional costs:

```
agent_engines.delete(resource_name=remote_agent.resource_name, force=True)  
print("✅ Agent successfully deleted")
```

```
INFO:vertexai.agent_engines:Deleting AgentEngine resource:  
projects/964731510884/locations/europe-west4/reasoningEngines/830389884105  
7329152  
INFO:vertexai.agent_engines:Delete AgentEngine backing LR0:  
projects/964731510884/locations/europe-west4/operations/821463237821621862  
4  
INFO:vertexai.agent_engines:AgentEngine resource deleted:  
projects/964731510884/locations/europe-west4/reasoningEngines/830389884105  
7329152
```

✅ Agent successfully deleted

### What happened:

This cell deletes your deployed agent:

- `resource_name=remote_agent.resource_name` - Identifies which agent to delete
- `force=True` - Forces deletion even if the agent is running

The deletion process typically takes 1-2 minutes. You can verify deletion in the [Agent Engine Console](#).

---



## Congratulations! You're Ready for Production Deployment

You've successfully learned how to deploy ADK agents to Vertex AI Agent Engine - taking your agents from development to production!

You now know how to deploy agents with enterprise-grade infrastructure, manage costs, and test production deployments.



### Learn More

Refer to the following documentation to learn more:

- [ADK Documentation](#)
- [Agent Engine Documentation](#)
- [ADK Deployment Guide](#)

### Other Deployment Options:

- [Cloud Run Deployment](#)
- [GKE Deployment](#)

### Production Best Practices:

- Delete test deployments when finished to avoid costs
- Enable tracing (enable\_tracing=True) for debugging
- Monitor via [Vertex AI Console](#)
- Follow [security best practices](#)

## Course Recap: Your 5-Day Journey

Over the past 5 days, you've learned:

- **Day 1:** Agent fundamentals - Building your first agent with tools and instructions
- **Day 2:** Advanced tools - Custom tools, built-in tools, and best practices
- **Day 3:** Sessions & Memory - Managing conversations and long-term knowledge storage
- **Day 4:** Observability & Evaluation - Monitoring agents and measuring performance
- **Day 5:** Production Deployment - Taking your agents live with Agent Engine

You now have the complete toolkit to build, test, and deploy production-ready AI agents!

## What's Next?

**Thank you for completing the 5-day AI Agents course!**

Now it's your turn to build:

- Start creating your own AI agents with ADK
- Share your projects with the community on [Kaggle Discord](#)
- Explore advanced patterns in the [ADK documentation](#)

**Happy building!** 

Authors
<a href="#">Lavi Nigam</a>