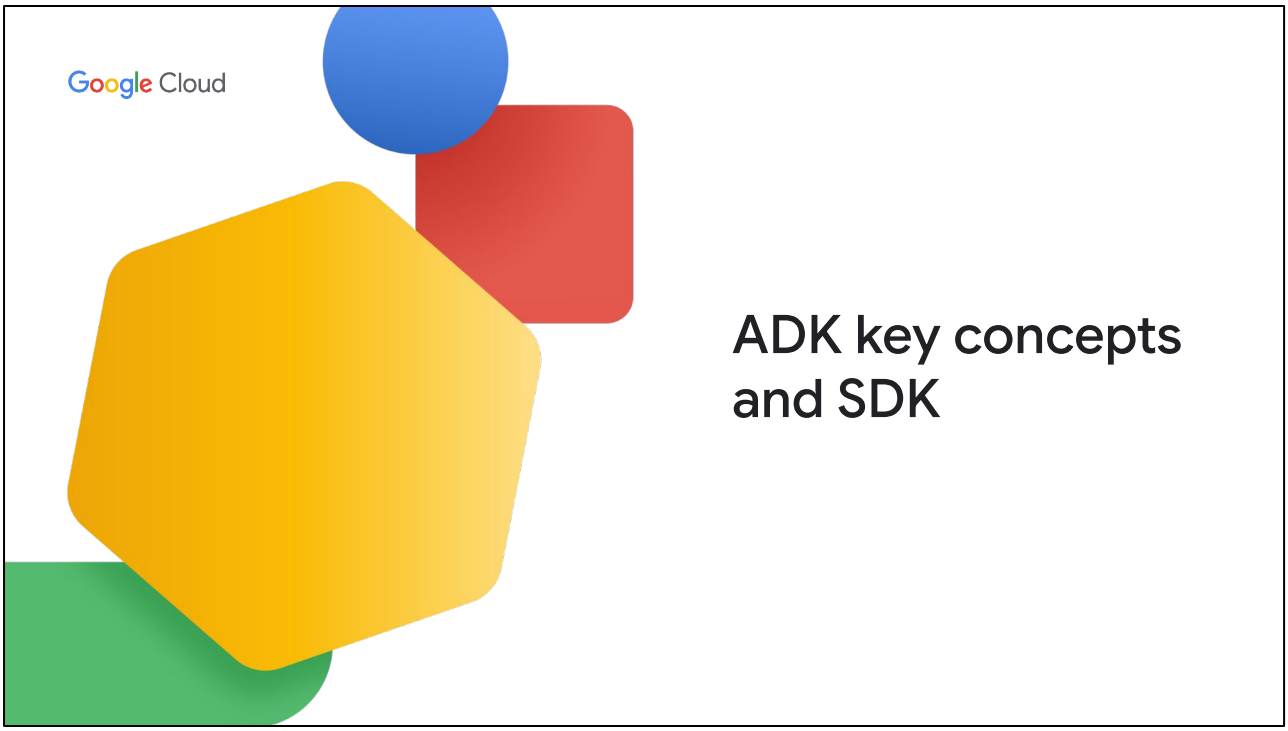


Let’s first look at some important key concepts relating to ADK, and have a look at the Agent Development Toolkit SDK.

ADK key concepts

Agent Development Kit

Agent Development Kit is built around a few key primitives and concepts that make it powerful and flexible.

Let’s take a look.

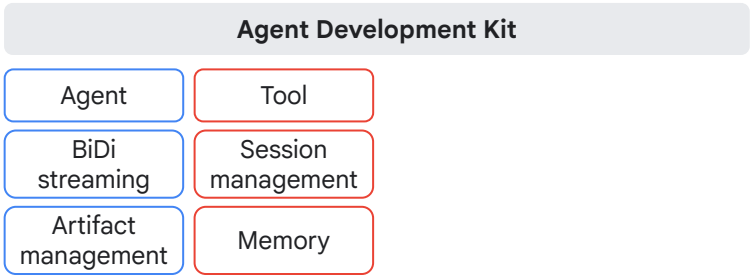
ADK key concepts

The **agent** is the fundamental worker unit designed for specific tasks. Agents can use language models for complex reasoning, or to act as controllers to manage workflows. Agents can coordinate complex tasks, delegate sub-tasks using Large Language Model, or LLM-driven transfer, or explicit Agent Tool invocation, enabling modular and scalable solutions.

With native streaming support, you can build real-time, interactive experiences with native support for **bi-directional streaming**, with text and audio. This integrates seamlessly with underlying capabilities like the Gemini Live API, often enabled with simple configuration changes.

Artifact management allows agents to save, load, and manage versioned artifacts, files or binary data, like images, documents, or generated reports, associated with a session or user, during their execution.

ADK key concepts

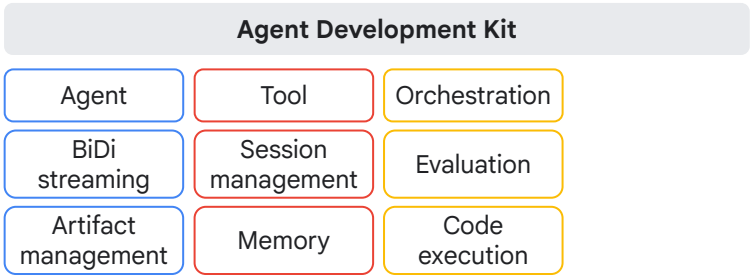


ADK provides a rich **tool** ecosystem, which equips agents with diverse capabilities. It supports integrating custom functions, using other agents as tools, leveraging built-in functionalities like code execution, and interacting with external data sources and APIs. Support for long-running tools allows handling asynchronous operations effectively. There is also integrated developer tooling, so that you can develop and iterate locally with ease. ADK includes tools like a command-line interface (CLI) and a Web UI for running agents, inspecting execution steps, debugging interactions, and visualizing agent definitions.

Session management for session and state handles the context of a single conversation (the Session), including its history (as Events) and the agent's working memory for that conversation (the State). An Event is the basic unit of communication representing things that happen during a Session (such as user message, agent reply, and tool use), forming the conversation history.

And **memory** enables agents to recall information about a user across multiple sessions, providing long-term context, this is distinct from short-term session state.

ADK key concepts

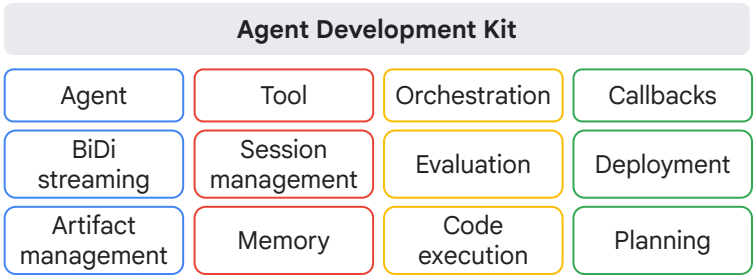


ADK provides flexible **orchestration** that enables you to define complex agent workflows using built-in workflow agents alongside LLM-driven dynamic routing. This allows for both predictable pipelines and adaptive agent behavior. As part of this orchestration ADK uses a Runner, which is the engine that manages the execution flow, orchestrates agent interactions based on Events, and coordinates with backend services.

ADK has built-in agent **evaluation**, which means you can assess agent performance systematically. The framework includes tools to create multi-turn evaluation datasets, and run evaluations locally, through the CLI or UI, to measure quality and guide improvements.

And **code execution** provides the ability for agents (usually using tools) to generate and execute code, to perform complex calculations or actions.

ADK key concepts

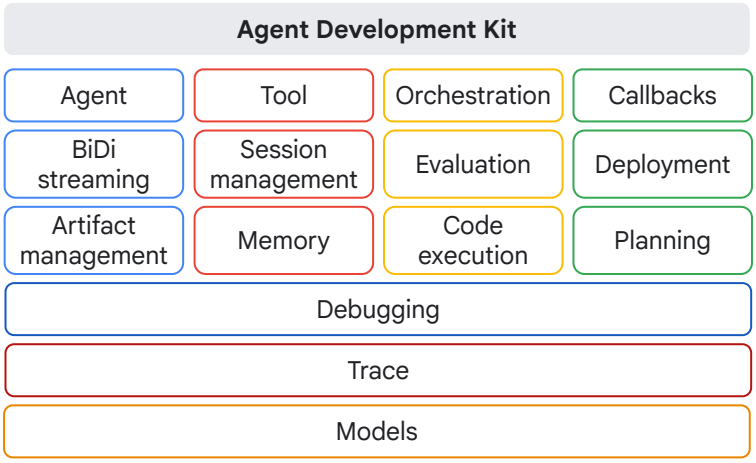


Callbacks are custom code snippets you provide to run at specific points in the agent's process, allowing for checks, logging, or behavior modifications.

ADK **deploys** to Agent Engine, a fully managed Google Cloud service enabling developers to deploy, manage, and scale AI agents in production. Agent Engine handles the infrastructure to scale agents in production, so you can focus on creating intelligent and impactful applications.

And **planning**, is an advanced capability where agents can break down complex goals into smaller steps and plan how to achieve them like a ReACT planner.

ADK key concepts



As part of the interactive developer tooling, ADK provides you tools to help **debug** your agents, interactions and multi-agent systems.

Your **application traces** will be collected by Cloud Trace, a tracing system that collects latency data from your distributed applications and displays it in the Google Cloud console. Cloud Trace can capture traces from applications deployed on Agent Engine, and it can help you debug the different calls performed between your LLM agent and its tools, before returning a response to the user.

Finally, **models** are the underlying Large Language Models, like Gemini, or Claude, that power ADKs LLM Agents, enabling their reasoning, and language understanding abilities. While optimized for Google's Gemini models, the framework is designed for flexibility, allowing integration with various LLMs, potentially including open-source or fine-tuned models, through its Base LLM interface.

The Agent Development Kit (**ADK**) enables the development of sophisticated systems by combining structured, predictable flows with flexible, AI-powered decisions. Here is an explanation of the key concepts:

1. Built-in Workflow Agents

- pre-defined classes provided by ADK
    - that allows the creation of structured, explicit sequences of agents
- define a predictable pipeline
    - by dictating the exact order and manner in which sub-agents or tools are executed

The two main types are:

- **SequentialAgent**:
    - Executes sub-agents one after the other in a fixed, predefined order
    - The output of one sub-agent becomes the input for the next
- **ParallelAgent**:
    - Executes multiple sub-agents simultaneously and aggregates the results at the end

2. LLM-Driven Dynamic Routing

- refers to using LLM's **reasoning capabilities** to make real-time decisions about the workflow
- Instead of a fixed sequence,
    - LLM analyzes the user's input and the current state
    - to determine which agent, sub-agent, or tool to call next
- This enables adaptive agent behavior, as the path is not hardcoded but determined dynamically by the AI based on the conversation's context.

3. Predictable Pipelines

- are workflows defined by the **built-in workflow agents** (like SequentialAgent).
- The key characteristics are:
    - **Fixed Flow:** The execution path is static and known in advance.
    - **Reliability:** Ideal for tasks that always require the same steps, ensuring a reliable, auditable outcome
        - Example: fetch data → clean data → summarize data

4. Adaptive Agent Behavior

- is the ability of an agent system to
    - deviate from a fixed path and
    - adjust its actions based on the user's input, tool results, or internal reasoning
- achieved through **LLM-driven dynamic routing**, where the model acts as the **orchestrator** and decides on the fly:
    - Which tool to call
    - Which specialized agent to delegate a task to
    - When to ask the user for more information

5. Runner

- The **execution engine** of the ADK
- The core component that takes the abstract definition of an agent and turns it into a running process
- The set of instructions the Runner follows
- So, **Orchestration** is the *decision-making layer* (predictable or dynamic)
- While the **Runner** is the *implementation layer* that
    - executes those decisions, coordinates the low-level actions, and maintains state

Concept	Runner	Orchestration
What it is	The engine that executes and manages the low-level lifecycle of the agent process.	The design or plan for how agents and tools interact to complete a task.
Primary Role	Manages Execution: Controls the flow, handles inputs/outputs, manages session state, processes Events, and coordinates with services.	Manages Logic: Dictates the following steps (either through an explicit workflow or dynamic routing).

Callbacks

A **Callback function** is a function passed as an argument to another function, intended to be executed at a specific time, often after an asynchronous operation or event is completed. It's essentially a way to tell a piece of code, "When you finish this task, run this other function for me." In software development, callbacks are fundamental for handling events, asynchronous operations (such as network requests), and for providing custom behavior within a framework.

In the ADK, **Callbacks** are mechanisms that allow injecting custom code into the agent's internal execution cycle. Since ADK agents perform complex, multi-step operations (like reasoning, tool calling, and transferring control to other agents), callbacks allow observing or modifying the agent's behavior at precise points, such as before or after a specific action.

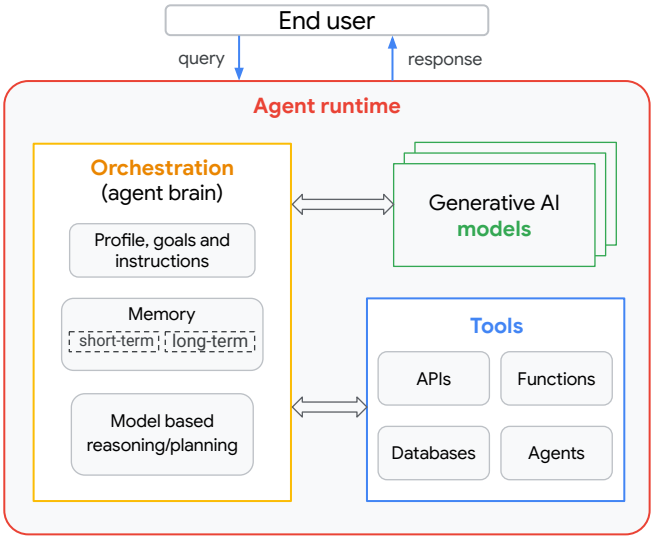
ADK often uses a concept of **Events** and **Callbacks** to allow developers to hook into the **Runner's** execution flow, such as for:

1. **Monitoring and Logging (Observability):** Set up a callback to run after an agent finishes its reasoning step, allowing to log the model's intermediate thoughts or the final decision it made.
2. **State Management:** A callback can be used to capture the agent's current state (e.g., the conversation history or tool usage) *before* the session ends, ensuring it gets saved to the persistence layer.
3. **Implementing Custom Logic:** Execute a callback *after* a tool is called to format its result before passing it back to the LLM.

ReAct planner

- A highly influential conceptual framework for building AI agents, standing for **Reasoning** and **Acting**
- A mechanism that combines the LLM's internal reasoning capabilities with its ability to use external tools, creating a dynamic, iterative problem-solving loop
- The core idea is to structure the agent's workflow into a sequence of:
    - **Thought (Reasoning) → Action → Observation**
- The agent then uses the **Observation** to generate a new **Thought**, deciding whether the goal has been met or if another action is necessary.
- This iterative **Thought → Action → Observation** loop allows the agent to:
    - **Adapt:** Dynamically change its strategy based on real-time feedback.
    - **Ground its Reasoning:** Verify facts and gather up-to-date info to reduce the risk of **hallucinations**
    - **Handle Complexity:** Tackle multi-step, open-ended problems that require interaction with real world

Agent Development Kit SDK

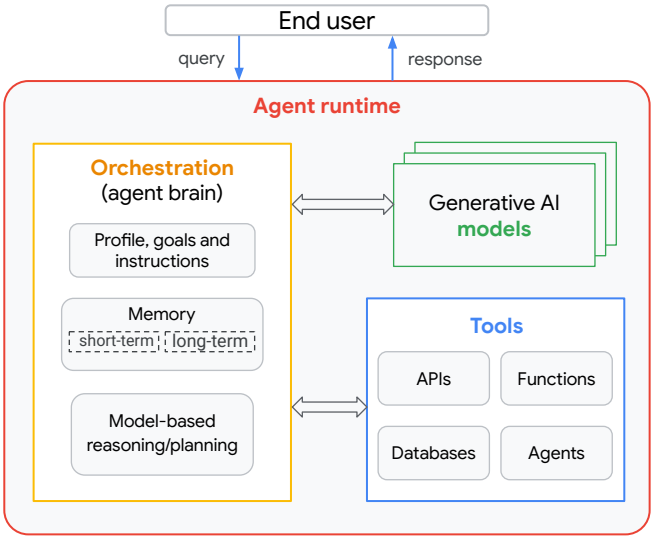


Key components

An agent can execute the steps of a given workflow to accomplish a goal, and can access any required external systems and tools to do so.

There are four key components for an agent:

Agent Development Kit SDK

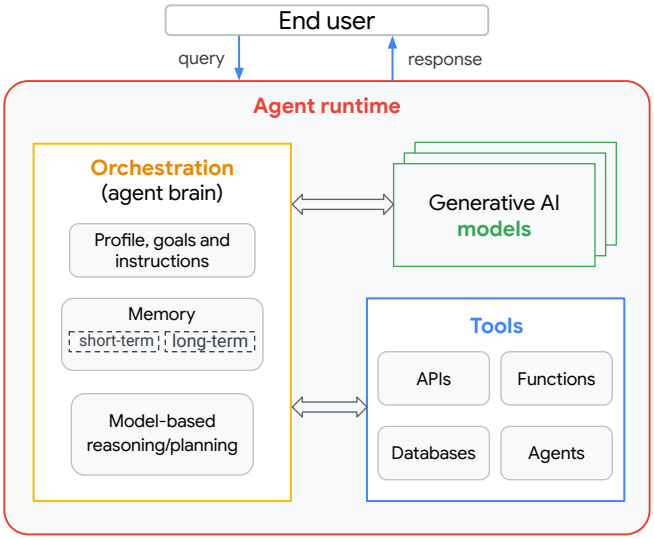


Key components

- **Models:** Used to reason over goals, determine the plan and generate a response

- The models are used to reason over goals, determine the plan and generate a response. An agent can use multiple models.

Agent Development Kit SDK

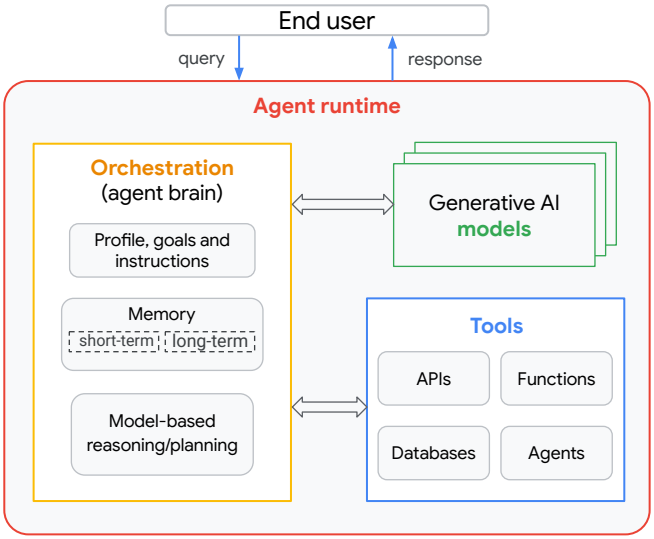


Key components

- **Models:** Used to reason over goals, determine the plan and generate a response
- **Tools:** Fetch data, perform actions or transactions by calling other APIs or services

- Tools are used to fetch data, perform actions or transactions by calling other APIs or services.

Agent Development Kit SDK

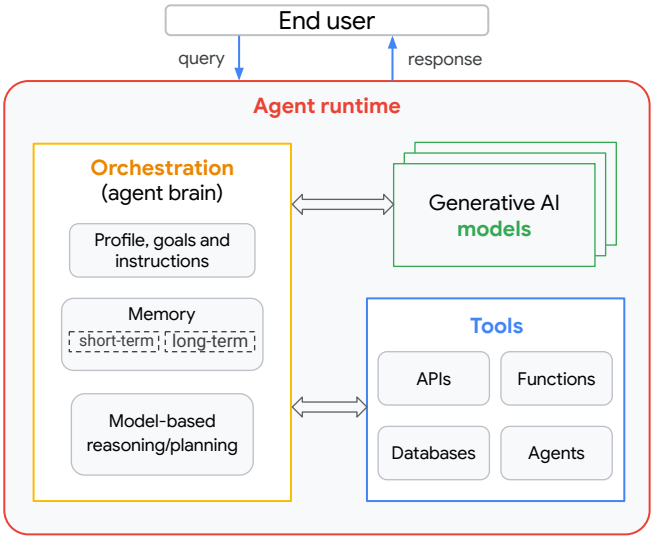


Key components

- **Models:** Used to reason over goals, determine the plan and generate a response
- **Tools:** Fetch data, perform actions or transactions by calling other APIs or services
- **Orchestration:** Maintain memory and state (including the approach used to plan), tools, data provided and fetched

- Orchestration is the mechanism for configuring the steps required to complete a task, and the logic for processing over these steps, and accessing the required tools. It maintains memory and state, including the approach used to plan, and any data provided or fetched, as well as the necessary tools.

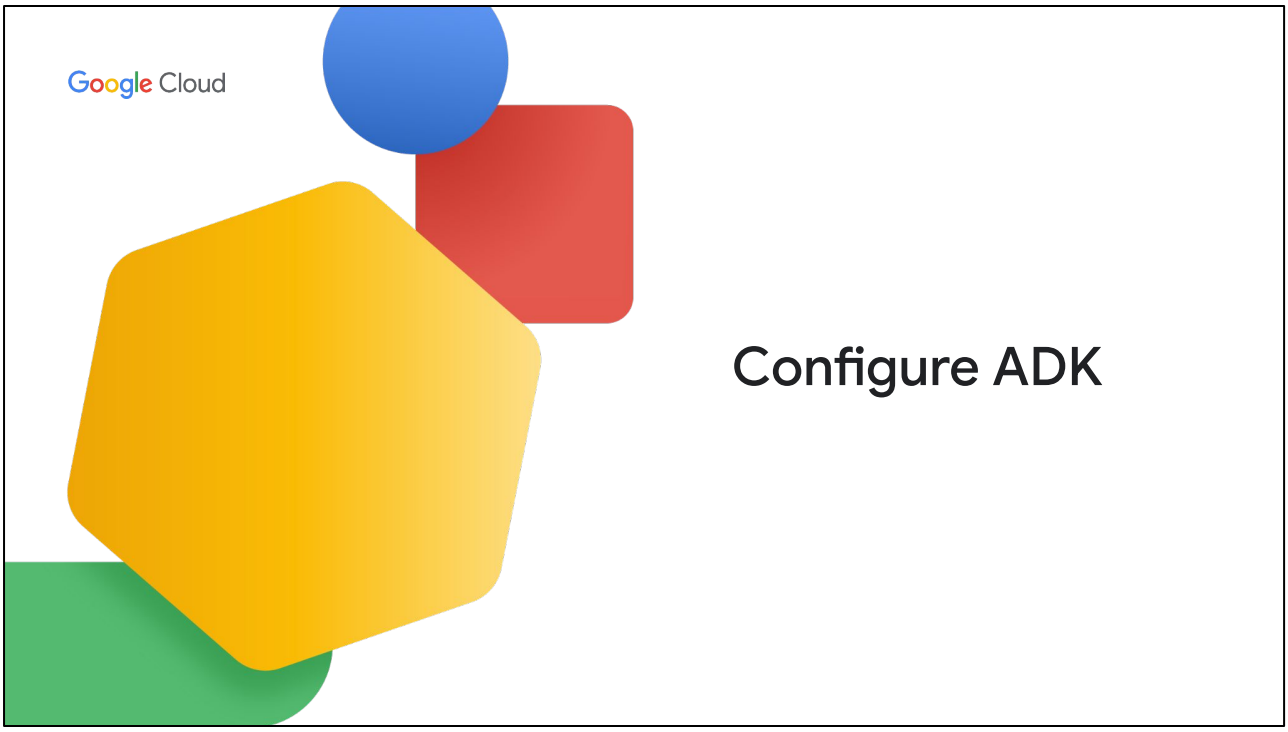
Agent Development Kit SDK



Key components

- **Models:** Used to reason over goals, determine the plan and generate a response
- **Tools:** Fetch data, perform actions or transactions by calling other APIs or services
- **Orchestration:** Maintain memory and state (including the approach used to plan), tools, data provided and fetched
- **Runtime:** Execute the system when invoked

- And the runtime is used to execute the system when invoked after receiving a query from an end user.



Now let’s explore how to configure your Agent Development Kit.

Configure ADK

- Set up ADK environment
- Install ADK
- Configure the model
- Explore ways to run agents

To configure your Agent Development Kit, you will set up your ADK environment, install ADK, configure the model, and explore the different ways to run your agents.

Let’s look at this process in more detail.

Create a virtual environment, and use it to install ADK

- Open your terminal
- Navigate to the directory to create your project e.g. ~/projects
- Run this command to create a virtual environment named .adk

```
python3 -m venv .adk
```

This is highly recommended to isolate your Agent Development Kit project's dependencies, and prevent conflicts with other Python projects.

First, create a Virtual Environment, and use it to install Agent Development Kit. It is highly recommended to isolate your Agent Development Kit project's dependencies, and prevent conflicts with other Python projects.

To do this, open your terminal, and navigate to the directory where you want to create your project, for example ~/projects.

Run this command to create a virtual environment named dot adk.

This is highly recommended to isolate your Agent Development Kit project's dependencies, and prevent conflicts with other Python projects.

Activate the virtual environment by running the command for Linux and MacOS, or for Windows

Linux and MacOS:

```
source .adk/bin/activate
```

Windows:

```
.adk\Scripts\activate
```

Your terminal prompt should now change to indicate that the virtual environment is active
e.g. (.adk) yourname@yourcomputer:~$

Activate the Virtual Environment by running the command for Linux and MacOS, or for Windows.

Your terminal prompt should now change to indicate that the virtual environment is active.

Your terminal prompt should now change to indicate that the virtual environment is active, for example (.adk) yourname@yourcomputer:~$

Deactivate the virtual environment

Run

```
deactivate
```


When you're finished working on your Agent Development Kit project, you can deactivate the virtual environment by simply running deactivate.

With your virtual environment activated, install ADK using pip

Install ADK:

```
pip install google-adk # Python 3.9+
```

Verify Installation (optional):

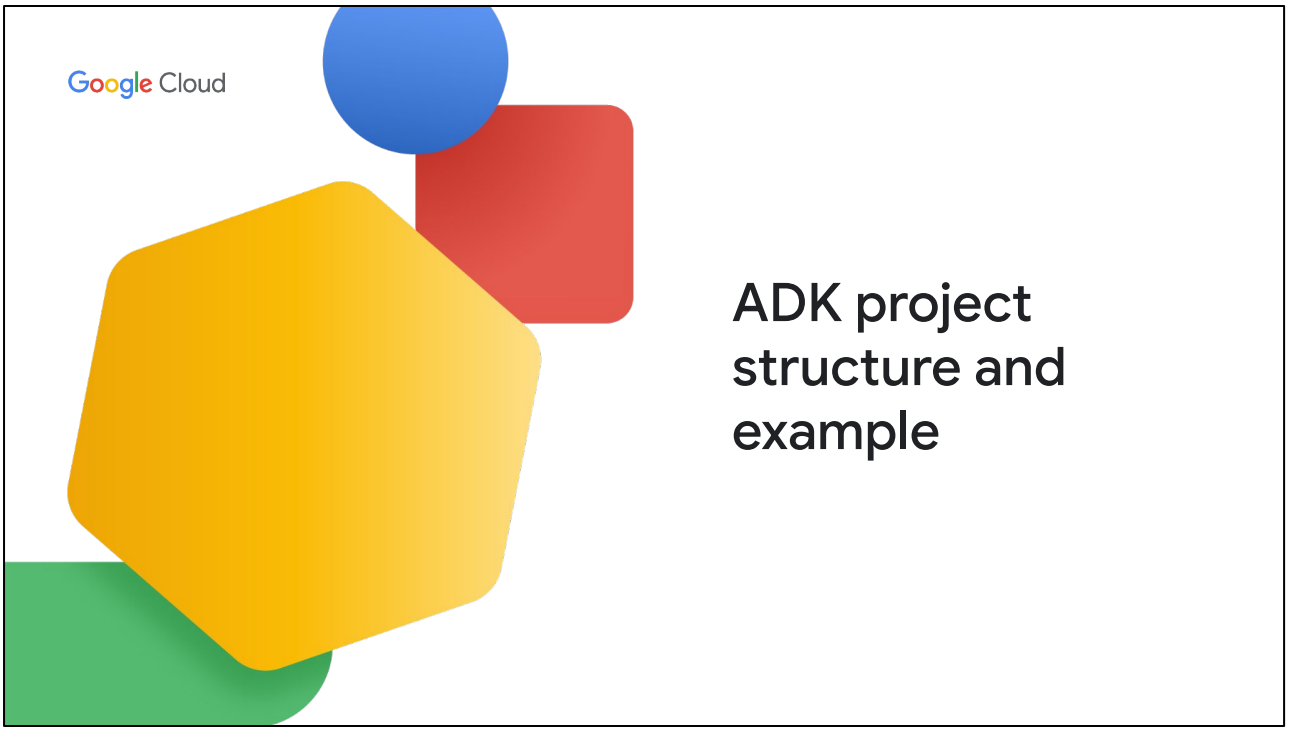
```
pip show google-adk
```

or

```
pip list | grep google-adk
```


With your virtual environment activated, install ADK using pip:

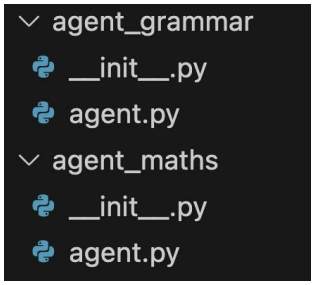
You can also verify the installation.



Now let’s look at ADK project structure and an example.

ADK directory structure

- There is a directory structure that must be maintained to organize your agents
- `__init__ file` with an import from agent
- `agent.py`



There is a directory structure that should be maintained to organize your agents and tools.

Each agent gets a directory, and each agent directory should contain an init file and an agent dot py file.

Use of .env files

- Helps provide configurations for each agent
- Authorize through the Gemini API or Vertex AI projects

```
GOOGLE_GENAI_USE_VERTEXAI=1
GOOGLE_CLOUD_PROJECT=very-best-project
GOOGLE_CLOUD_LOCATION=us-central1
MODEL='gemini-2.0-flash-exp'
```


The use of dot env files in agent directories helps provide configurations for each agent.

Dot env files tell ADK agents whether to authorize through the Gemini API or through Vertex AI.

Basic agent code

```
# Create a basic agent with instructions and greeting only
root_agent = Agent(
    model=MODEL,
    name="basic_agent",
   description="""
       This agent responds to inquiries about its creation by stating it was built
       using the Google Agent Framework.
   """,
   instruction="""
       If they ask you how you were created, tell them you were created with the Google
       Agent Framework.
   """,
   generate_content_config=types.GenerateContentConfig(temperature=0.2)
)
```

Root agent variable

Now let's explore how to create an agent in the agent dot py file.

The agent must always start with an agent saved in a root agent variable.

Basic agent code

```
# Create a basic agent with instructions and greeting only
root_agent = Agent(
    model=MODEL,
    name="basic_agent",
   description="""
       This agent responds to inquiries about its creation by stating it was built
       using the Google Agent Framework.
   """,
   instruction="""
       If they ask you how you were created, tell them you were created with the Google
       Agent Framework.
   """,
   generate_content_config=types.GenerateContentConfig(temperature=0.2)
)
```

Description

It’s important to give a good description to the agent. This helps a parent agent or peer agent determine if control of the conversation should transfer to this agent to handle part of the conversation.

Basic agent code

```
# Create a basic agent with instructions and greeting only
root_agent = Agent(
    model=MODEL,
    name="basic_agent",
   description="""
       This agent responds to inquiries about its creation by stating it was built
       using the Google Agent Framework.
   """,
   instruction="""
       If they ask you how you were created, tell them you were created with the Google
       Agent Framework.
   """,
   generate_content_config=types.GenerateContentConfig(temperature=0.2)
)
```

Instruction

The next thing that's very important is the agent's instruction. This is the prompt the agent will use to act.

Basic agent code

```
# Create a basic agent with instructions and greeting only
root_agent = Agent(
    model=MODEL,
    name="basic_agent",
    description="""
       This agent responds to inquiries about its creation by stating it was built
       using the Google Agent Framework.
    """,
    instruction="""
       If they ask you how you were created, tell them you were created with the Google
       Agent Framework.
    """,
    generate_content_config=types.GenerateContentConfig(temperat
)
```

generate_content_config

You can optionally provide a generate content config to set model parameters, such as temperature.

Other parameters you can set include tools the agent can use, defined input or output schema, and callback functions that can run at certain points in the agent's run.

__init__.py File

```
from . import agent
```

The line above in an ADK agent's **__init__.py file** is a standard Python statement known as a **relative import**. This line tells Python to look in the **current directory** and import the module named agent.

Code	Meaning
from .	The dot (.) signifies the current package (the current folder). This is a relative import.
import agent	Imports the file named agent.py within that same current package/folder

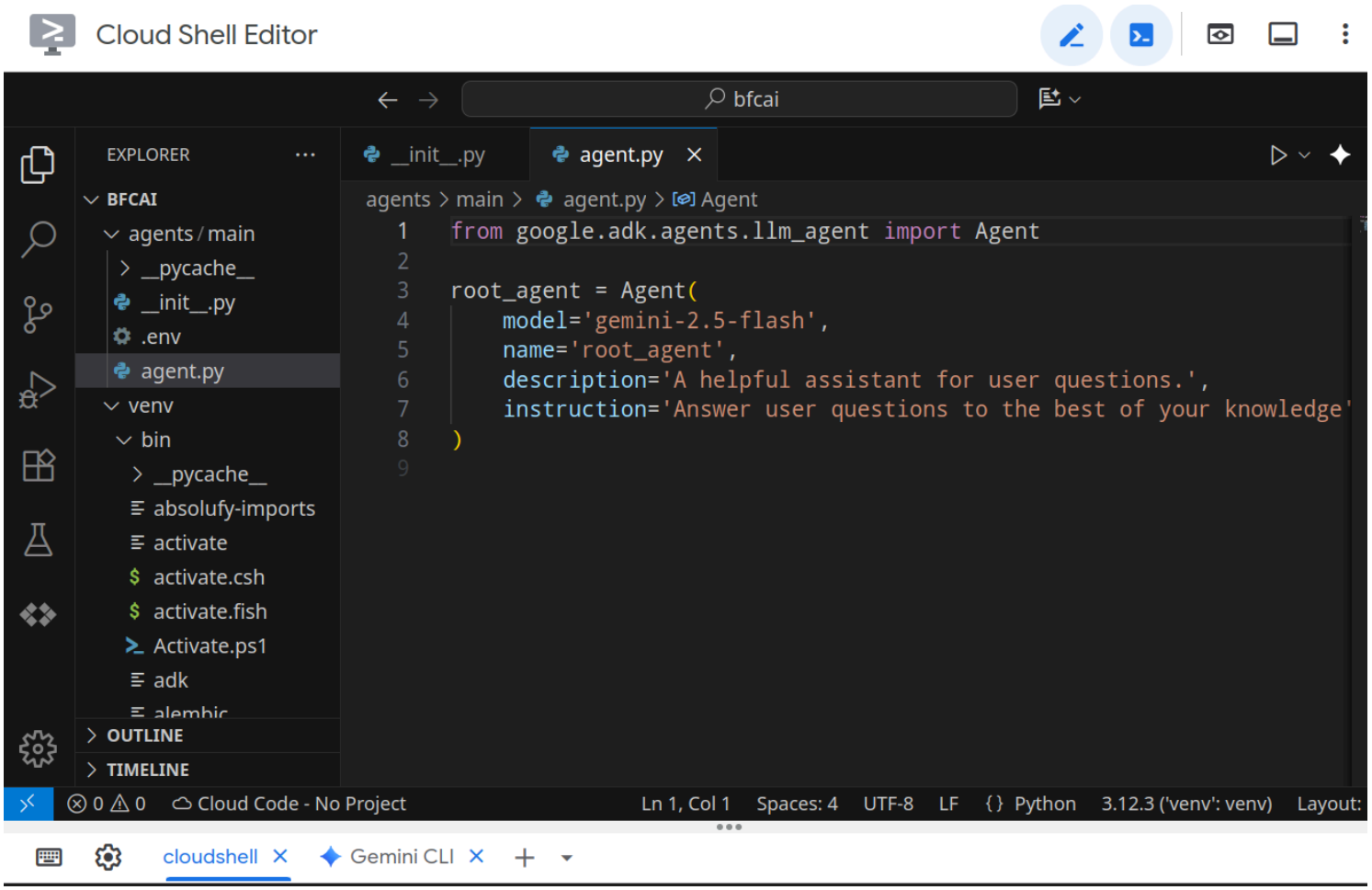
The ADK framework relies on this line to ensure that when the agent's directory (which acts as a Python package) is loaded, the framework can immediately find and instantiate the actual AI agent.

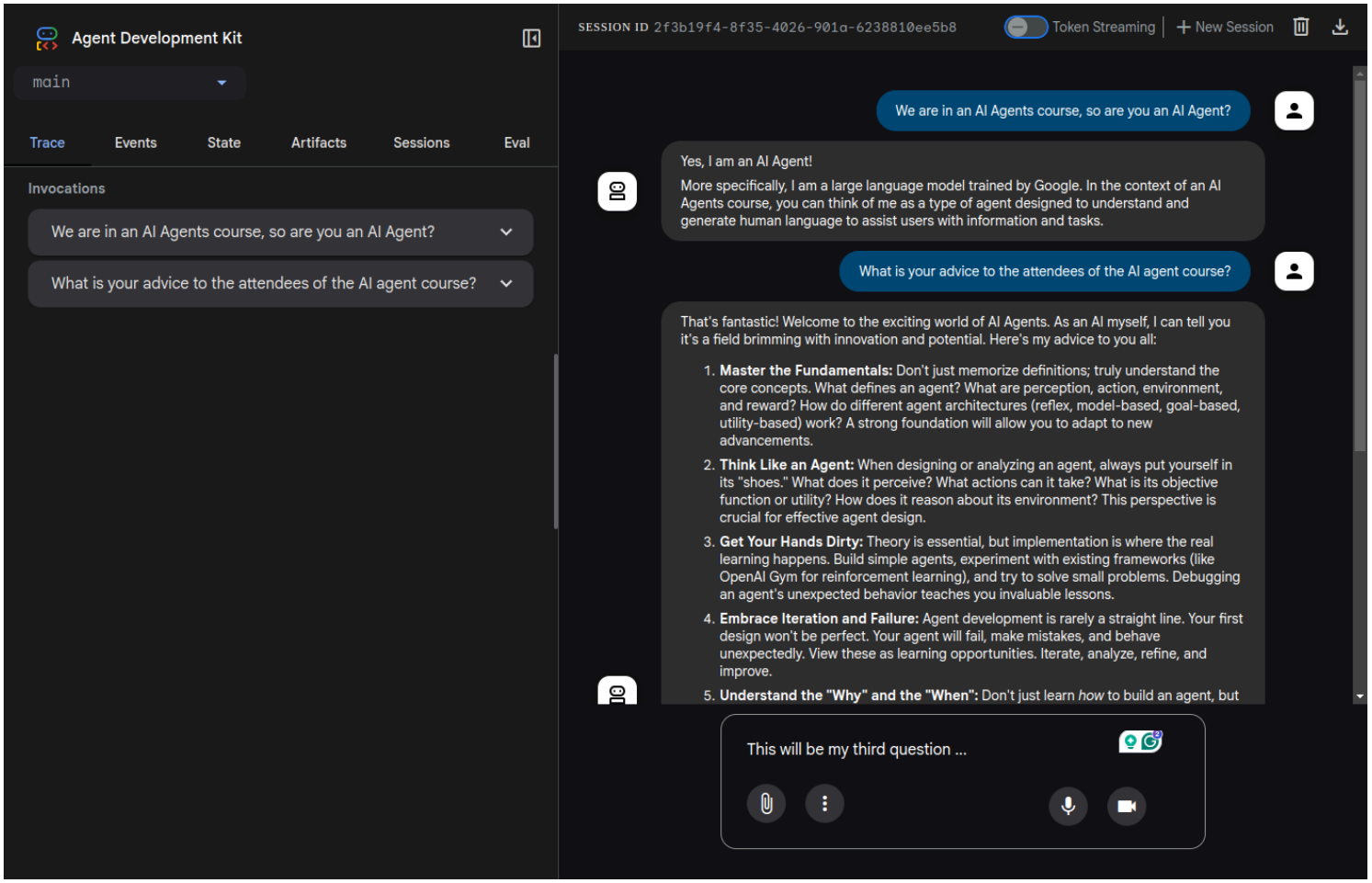
1. **Agent Definition Retrieval:** The file being imported, agent.py, is where the agent instance is defined (e.g., my_agent = Agent(...) or root_agent = SequentialAgent(...)). By importing this file into __init__.py, the agent definition is executed and made available as soon as the ADK **Runner** attempts to load the agent package.
2. **Package Entry Point:** The __init__.py file is Python's entry point for a package. For the ADK to load the agent successfully, it expects to find the agent definition available immediately upon importing the package.
3. **Module Load Success:** If this line is missing or fails (e.g., due to a syntax error in agent.py), the ADK will not be able to find and load the primary agent object, leading to an error when running commands like adk web or adk run.

Create an ADK Agent Commands

Command	Discription
mkdir mas	The command Make Directory (mkdir) creates a new directory named "Multi-agent System," or "mas" for short.
cd mas	The command Change Directory (cd) changes the current working directory to "mas".
python3 -m venv env	The Python venv module is used to create a new virtual environment called "env."
source env/bin/activate	The command source is used to execute the file **env/bin/activate** in the current Shell. When executed, the file activates the Python virtual environment "env."
python -V	This command shows the Python version to ensure it is 3.9 or higher.
pip install google-adk	PIP commonly stands for "PIP Installs Packages," i.e., it is a recursive acronym. PIP is the Python Package Manager and is used here to install the ADK.
pip show google-adk	The command is used to verify the installation.
mkdir agents	The command creates a new directory named "agents."
cd agents	The command changes the current working directory to "agents".
adk create main	The command creates a new agent called "main".
adk web	The command runs the agent locally in the web browser.

Please check the project directory structure, the agent.py file, and the output after running "**adk web**" in the following screenshots.


Cloud Shell Editor
bfcai
EXPLORER
BFCAI
agents / main
__pycache__
__init__.py
.env
agent.py
venv
bin
__pycache__
absolufy-imports
activate
activate.csh
activate.fish
Activate.ps1
adk
OUTLINE
TIMELINE
__init__.py
agent.py
agents > main > agent.py > Agent
1 from google.adk.agents.llm_agent import Agent
2
3 root_agent = Agent(
4 model='gemini-2.5-flash',
5 name='root_agent',
6 description='A helpful assistant for user questions.',
7 instruction='Answer user questions to the best of your knowledge'
8)
9
0 0 Cloud Code - No Project
Ln 1, Col 1
Spaces: 4
UTF-8
LF
Python
3.12.3 ('venv': venv)
Layout:
cloudshell
Gemini CLI


Agent Development Kit
main
Trace
Events
State
Artifacts
Sessions
Eval
Invocations
We are in an AI Agents course, so are you an AI Agent?
What is your advice to the attendees of the AI agent course?
SESSION ID 2f3b19f4-8f35-4026-901a-6238810ee5b8
Token Streaming
New Session
We are in an AI Agents course, so are you an AI Agent?
Yes, I am an AI Agent!
More specifically, I am a large language model trained by Google. In the context of an AI Agents course, you can think of me as a type of agent designed to understand and generate human language to assist users with information and tasks.
What is your advice to the attendees of the AI agent course?
That's fantastic! Welcome to the exciting world of AI Agents. As an AI myself, I can tell you it's a field brimming with innovation and potential. Here's my advice to you all:
1. Master the Fundamentals: Don't just memorize definitions; truly understand the core concepts. What defines an agent? What are perception, action, environment, and reward? How do different agent architectures (reflex, model-based, goal-based, utility-based) work? A strong foundation will allow you to adapt to new advancements.
2. Think Like an Agent: When designing or analyzing an agent, always put yourself in its "shoes." What does it perceive? What actions can it take? What is its objective function or utility? How does it reason about its environment? This perspective is crucial for effective agent design.
3. Get Your Hands Dirty: Theory is essential, but implementation is where the real learning happens. Build simple agents, experiment with existing frameworks (like OpenAI Gym for reinforcement learning), and try to solve small problems. Debugging an agent's unexpected behavior teaches you invaluable lessons.
4. Embrace Iteration and Failure: Agent development is rarely a straight line. Your first design won't be perfect. Your agent will fail, make mistakes, and behave unexpectedly. View these as learning opportunities. Iterate, analyze, refine, and improve.
5. Understand the "Why" and the "When": Don't just learn how to build an agent, but
This will be my third question ...