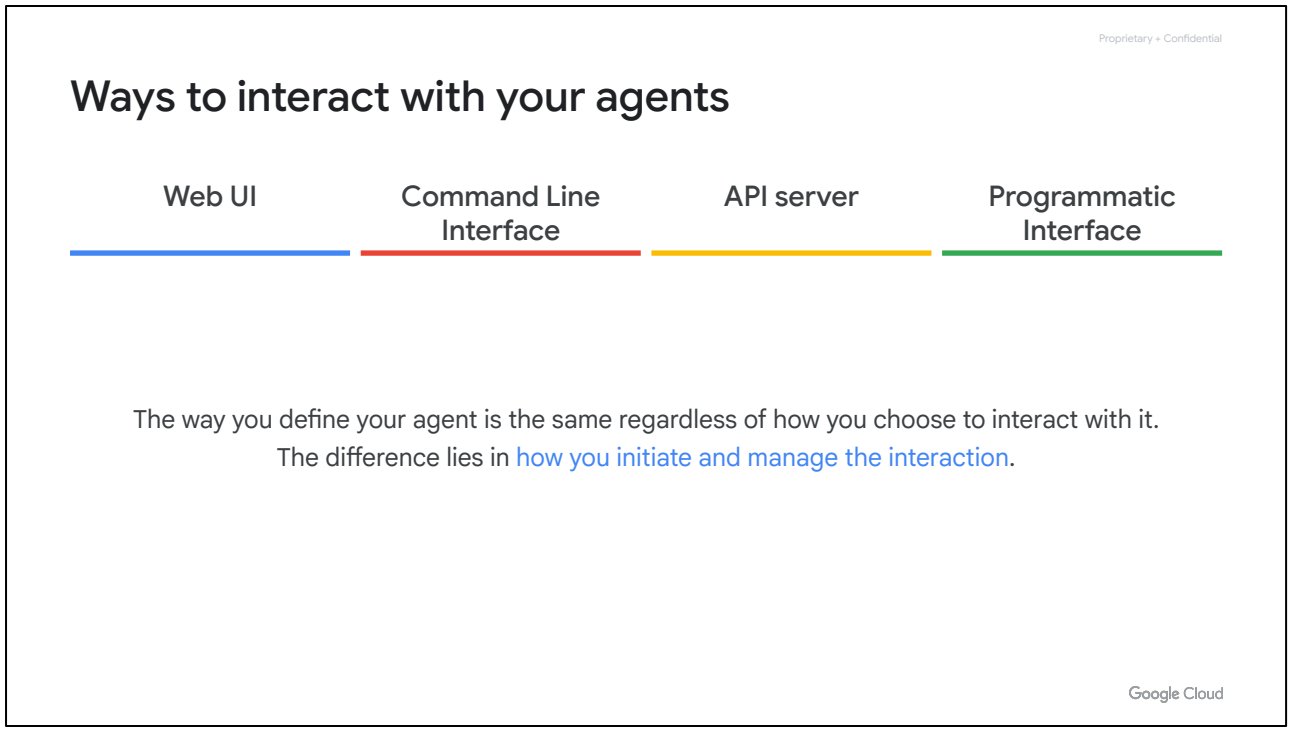
There are different methods for interacting with your agents, each of which offers different levels of control and integration.

Let’s take a look.



ADK offers four primary ways to interact with your agents: These are, through:

- Web UI,
- A Command-Line Interface (or CLI),
- An API Server, or
- The Programmatic Interface.

The way you define your agent (the core logic within agent dot py) is the same regardless of how you choose to interact with it. The difference lies in how you initiate and manage the interaction.

Ways to interact with your agents

Web UI | Command Line Interface | API server | Programmatic Interface

- Interact with your agent through a user-friendly web browser
- Use for visual interaction and monitoring agent behavior
- Only use for local testing - not suitable for a production environment

With Web UI, you can interact with your agent through a user-friendly web browser.

Web UI is a good option for visual interaction while developing your agent and monitoring agent behavior.

It should only be used for local testing, it is not suitable for a production environment.

Ways to interact with your agents

Web UI | Command Line Interface | API server | Programmatic Interface

Get started with Web UI

1. Open your terminal
2. Use cd to navigate to the directory
3. Run this command from your project folder to start a local web server and user interface.

```
adk web
```


To get started with Web UI,:

- Open your terminal, and
- Use cd to navigate to the directory containing your agent folder.
- Next, run this command to start a local web server and open a new tab in your browser, providing a visual interface for interacting with your agent.

Ways to interact with your agents

Web UI | **Command Line Interface** | API server | Programmatic Interface

- Use terminal commands to interact directly with your agent
- Use for quick tasks, scripting, automation, and developers comfortable with terminal commands
- Only use for local testing - not suitable for a production environment

With the CLI, you can use terminal commands to interact directly with your agent.

The CLI is a good option for quick tasks, scripting, automation, and developers comfortable with terminal commands.

This should also only be used for local testing, it is not suitable for a production environment.

Ways to interact with your agents

Web UI | Command Line Interface | API server | Programmatic Interface

Get started with CLI

1. Open your terminal
2. Use cd to navigate to the directory
3. Run this command to start the agent

```
adk run my_google_search_agent # Replace with your agent's folder name
```


To get started with the CLI:

- Open your terminal, and use cd to navigate to the directory containing your agent folder.
- Next, run this command to start the agent, and then you can interact with it directly in the terminal.

Ways to interact with your agents

Web UI | Command Line Interface | **API server** | Programmatic Interface

- Run your agent as a REST API, allowing other applications to communicate with it
- Use for integration with other applications, building services that use the agent, and remote access to the agent
- Use for production environments

To interact with your agent through the API server, run your agent as a REST API, allowing other applications to communicate with it.

Running an API server is a good option for integration with other applications, building services that use the agent, and remote access to the agent.

This approach can be used for production environments.

Ways to interact with your agents

Web UI | Command Line Interface | **API server** | Programmatic Interface

Get started with API server

1. Open your terminal
2. Use cd to navigate to the directory
3. Run this command to start a local API server

```
adk api_server my_google_search_agent # Replace with your agent's folder name
```


To get started with an API server:

- Open your terminal, and use cd to navigate to the directory containing your agent folder.
- Next, run this command to start a local API server, using Flask, on port 8000. You can then interact with your agent through REST API calls.

Ways to interact with your agents

Web UI | Command Line Interface | API server | **Programmatic Interface**

- Integrate ADK directly into your Python applications, or interactive notebooks e.g. Jupyter, Colab
- Use a Session and Runner, and define and interact with your agent within the same file or notebook cell
- Provides deep integration within applications, custom workflows, notebooks, and fine-grained control over agent execution
- Use for production environments

The Programmatic Interface allows you to integrate ADK directly into your Python applications, or interactive notebooks (like Jupyter and Colab).

Unlike the CLI, Web UI, and API server, you don't need the specific project structure, as previously described.

Instead you'll be using a Session and Runner. You can define and interact with your agent within the same file or notebook cell. The programmatic interface provides deep integration within applications, custom workflows, notebooks, and fine-grained control over agent execution. This approach can be used for production environments.

Ways to interact with your agents

Web UI | Command Line Interface | API server | **Programmatic Interface**

Get started with Programmatic Interface

1. Set up Memory.
   - `InMemorySessionService`
   - `InMemoryArtifactService`
2. Create a new session.
3. Prepare content for the agent.
4. Use a Runner to execute the agent's logic.
5. Process the event stream to get the final response.

The Programmatic Interface requires that you:

1. Handle setting up memory, including the in-memory session service and the in-memory artifact service.
2. You also need to create a new session,
3. Prepare content, such as the user query, for the agent
4. Use a Runner to execute the agent’s logic, and
5. Process the event stream to get the final response.

Invoking ADK agents programmatically

```
root_agent = LLMAgent(...)

runner = InMemoryRunner(agent=root_agent)

session = runner.session_service.create_session(
    app_name=runner.app_name, user_id="test_user"
)

user_input = “My prompt”
content = UserContent(parts=[Part(text=user_input)])
for event in runner.run(
    user_id=session.user_id, session_id=session.id,
    new_message=content
):
    for part in event.content.parts:
        print(part.text)
```

Create your agent

Create a runner

Create a session

Content prepared

Run and iterate through response

To run an agent programmatically as part of an application, a runner is needed to handle conversation sessions. In this example, you create an in-memory runner.

Sessions store the conversational history, the agent's internal state, including variables, and other data related to a specific interaction. It's ephemeral in this in-memory implementation. The Memory Artifact stores files, or data generated or used by the agent. This could be text files, images, or any other kind of data. Like sessions, these are lost when the program ends in the in-memory version.

A new session is used to create a track of each conversation.

Content is prepared, encapsulating a user’s text query into Parts, and into a Content object, from the Google Gen AI dot types package. A “Role” is also added, defining who is sending the message.

Flask

- **Flask** is a micro web framework for Python.
    - It is called a "microframework" because it provides a **lightweight, minimalist core** for building web apps and **APIs**, including URL routing and a templating engine (Jinja2).
- Unlike full-stack frameworks (like **Django**),
    - **Flask** doesn't include built-in features for things like database management or user authentication, offering developers maximum **flexibility** to choose their own tools and extensions.
- **Flask** is often used for smaller, simpler projects and **RESTful APIs**.

Runner Types

1. The Base Runner Class

- **Runner** (i.e., google.adk.runners.Runner): This is the abstract class that defines the core functionality for orchestrating an agent's execution, managing the event loop, and coordinating with services for session and state management.

2. The InMemoryRunner Class

- The runner used in the shown code.

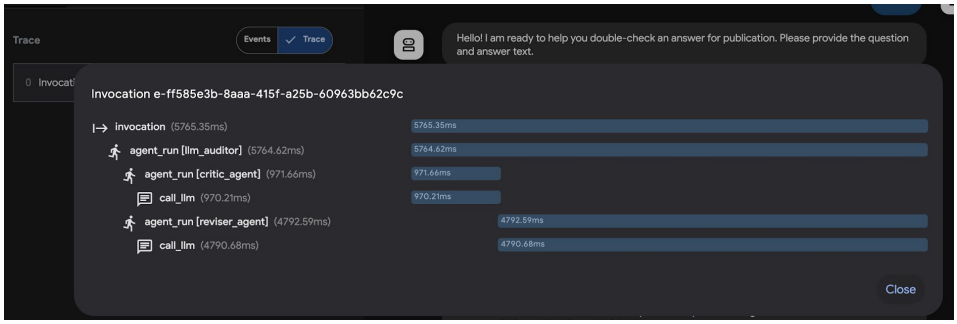
3. Deployment-Specific Runtime Environments

In production environments, the role of the Runner (especially session management) is often fulfilled or managed by a cloud service that can provide persistence, scalability, and managed execution. These environments support the ADK agent but typically replace the simple InMemorySessionService that InMemoryRunner uses:

- **Custom Runner Implementation (with Persistence):** The base Runner can be instantiated with different services, effectively creating "other types" of runners focused on persistence:
    - A runner using **DatabaseSessionService**: Persists session history and state to a SQL database (e.g., MySQL, PostgreSQL) using SQLAlchemy, making the agent stateful and durable.
    - A runner using **VertexAiSessionService**: Leverages Google Cloud's managed services for session storage, which is required for production deployments on the Google Cloud ecosystem.
- **Vertex AI Agent Engine:** This is a managed Google Cloud service designed to deploy, host, and scale ADK agents. When an agent is deployed here, the engine's runtime handles the orchestration, event streaming, and session persistence (using VertexAiSessionService), effectively acting as the production-grade, scaled-up "runner" environment.
- **Cloud Run / GKE:** When deploying ADK agents as independent microservices (often following the **Agent-to-Agent (A2A)** protocol), the **Runner** (with a suitable SessionService) is typically integrated into the web server (like a FastAPI app hosted on Cloud Run or GKE) that exposes the agent's /run endpoint.

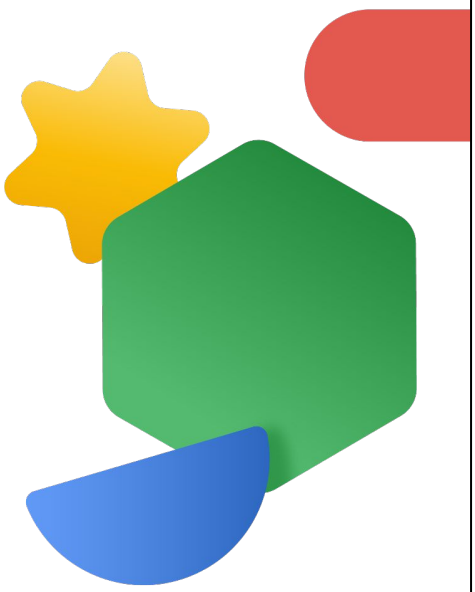
Trace agent runs

In the Dev UI, you can follow the overall and individual spans of each step of your agents' invocations.



In the Dev UI, you can follow the overall and individual spans of each step of your agents' invocations.

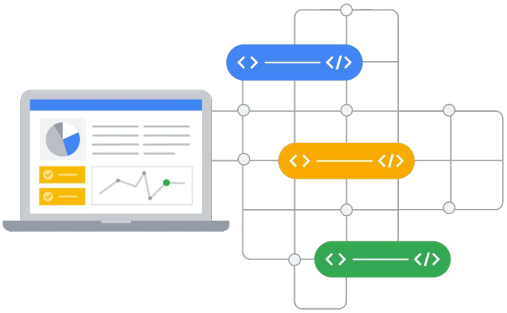
Callbacks



Next, let's discuss Callbacks.

Callbacks

- Callbacks allow you to interrupt the agent's execution lifecycle to observe, modify, or augment the agent's operations directly.
- ADK provides different types of callbacks that trigger at various stages of an agent's execution - defined on the Base Agent and the more specialized LLM Agent classes.
- To use callbacks effectively, understand when each callback fires and what context it receives.



Callbacks are a fundamental mechanism within Agent Development Kit. They allow you to interrupt the agent's execution lifecycle, enabling you to observe, modify, or augment the agent's operations directly.

You define a Python function that takes specific parameters according to which callback hook you are using, then register them with an agent to be called at the appropriate time. The framework automatically invokes these functions at predefined points during an agent's run. Think of them as designated interception or interrupt points, where you can inject your custom code.

The framework provides different types of callbacks that trigger at various stages of an agent's execution. These are primarily defined on the Base Agent and the more specialized LLM Agent classes.

Understanding when each callback fires and what context it receives is key to using them effectively.

Agent lifecycle callbacks

`before_agent_callback` `after_agent_callback`

Agent lifecycle callbacks are available on any agent that inherits from the Base Agent (including LLM Agent, Sequential Agent, Parallel Agent, or Loop Agent).

Agent lifecycle callbacks

`before_agent_callback`

- Setting up resources or state needed only for this specific agent's run
- Performing validation checks on the session state (callback_context.state) before execution
- Logging the entry point of the agent's activity
- Modifying session state before the agent uses it
- Returning content to skip the agent's execution

`after_agent_callback`

Before Agent Callback, is called immediately before the agent's run method is executed.

- It is ideal for setting up resources or state needed only for this specific agent's run, performing validation checks on the session state, before execution starts, logging the entry point of the agent's activity, or potentially modifying the session state before the agent uses it.
- It can also be used to skip the agent's run if it returns a default response or cached content that replaces the agent's execution.

Agent lifecycle callbacks

`before_agent_callback`

- Setting up resources or state needed only for this specific agent's run
- Performing validation checks on the session state (callback_context.state) before execution
- Logging the entry point of the agent's activity
- Modifying session state before the agent uses it
- Returning content to skip the agent's execution

`after_agent_callback`

- Cleanup tasks
- Post-execution validation
- Logging the completion of an agent's activity
- Modifying final state
- Augmenting/replacing the agent's final output

After Agent Callback, is called immediately after the agent's run method successfully completes. It does not run if the agent was skipped due to 'before agent callback' returning content, or if 'end invocation' was set during the agent's run.

- This is useful for cleanup tasks, post-execution validation, logging the completion of an agent's activity, modifying final state, or augmenting or replacing the agent's final output.

LLM interaction callbacks

`before_model_callback`

`after_model_callback`

LLM interaction callbacks are specific to LLM agents, and provide hooks around the interaction with the Large Language Model.

LLM interaction callbacks

`before_model_callback`

- Inspection and modification of the request going to the LLM
  - Adding dynamic instructions
  - Injecting few-shot examples based on state
  - Modifying model config.
  - Implementing guardrails (like profanity filters)
  - Implementing request-level caching

`after_model_callback`

Before Model Callback is called just before the generate content request is sent to the LLM within an LLM Agent's flow.

- It allows inspection and modification of the request going to the LLM. Use cases include adding dynamic instructions, injecting few-shot examples based on state, modifying model config, implementing guardrails (like profanity filters), or implementing request-level caching.
- If the callback returns an LLM Response object, then the call to the LLM is skipped, and the returned LLM Response is used directly as if it came from the model. This is powerful for implementing guardrails or caching.

LLM interaction callbacks

`before_model_callback`

- Inspection and modification of the request going to the LLM
  - Adding dynamic instructions
  - Injecting few-shot examples based on state
  - Modifying model config.
  - Implementing guardrails (like profanity filters)
  - Implementing request-level caching

`after_model_callback`

- Inspection or modification of the raw LLM response
  - Logging model outputs
  - Reformatting responses
  - Censoring sensitive information generated by the model
  - Parsing structured data from the LLM response and storing it
  - Handling specific error codes

After Model Callback is called just after a response is received from the LLM, before it's processed further by the invoking agent.

- It allows inspection or modification of the raw LLM response. Use cases include logging model outputs, reformatting responses, censoring sensitive information generated by the model, parsing structured data from the LLM response and storing it in callback_context.state, or handling specific error codes.

Tool execution callbacks

`before_tool_callback`

`after_tool_callback`

Tool execution callbacks are also specific to the LLM Agent, and trigger around the execution of tools (including Function Tool, and Agent Tool) that the LLM might request.

Tool execution callbacks

`before_tool_callback`

- Inspection and modification of tool arguments
- Performing authorization checks before execution
- Logging tool usage attempts
- Implementing tool-level caching

`after_tool_callback`

Before Tool Callback, is called just before a specific tool's run method is invoked, after the LLM has generated a function call for it.

- It allows inspection and modification of tool arguments, performing authorization checks before execution, logging tool usage attempts, or implementing tool-level caching.
- If a dictionary is returned, the tool's run_async method is skipped. The returned dictionary is used directly as the result of the tool call. This is useful for caching or overriding tool behavior.

Tool execution callbacks

`before_tool_callback`

- Inspection and modification of tool arguments
- Performing authorization checks before execution
- Logging tool usage attempts
- Implementing tool-level caching

`after_tool_callback`

- Inspection and modification of the tool's result before it's sent back to the LLM (potentially after summarization)
  - Logging tool results
  - Post-processing or formatting results
  - Saving specific parts of the result to the session state

After Tool Callback is called just after the tool's run_async method completes successfully.

- It allows inspection and modification of the tool's result before it's sent back to the LLM (potentially after summarization). It’s useful for logging tool results, post-processing or formatting results, or saving specific parts of the result to the session state.
- If a new dictionary is returned, it replaces the original tool response. This allows modifying or filtering the result viewed by the LLM.

Cleanup Tasks (for after_agent_callback)

- Cleanup focuses on **teardown and resource management** after the agent has successfully finished its useful work.
- **Action:** Deleting temporary files, closing database connections opened specifically for the agent's turn, or removing ephemeral data from the session state.
- **Flow Control:** Must return **None** to avoid replacing the agent's final answer.

Request-Level Caching (for before_model_callback)

- **Goal:** To increase efficiency by serving instant responses for repeated or known queries without incurring LLM cost.
- **Mechanism:** The callback generates a unique key (a hash) based on the entire LLM request, including (prompt, config, etc.). It checks a cache for that key. If a match is found (a "cache hit"), it retrieves the stored LLM r esponse and returns it, **skipping** the expensive LLM call.

Handling specific error codes (after_model_callback)

- The primary purpose of using the ADK after_model_callback to handle errors is to **intercept and replace a non-ideal or erroneous response** from the underlying LLM with a **graceful, custom, and user-friendly response**.
- An example of an error is a rate limit response from the model API.

Parsing structured data from the LLM response and storing it in callback_context.state (after_model_callback)

- The callback_context.state is simply a **mutable dictionary-like object** (State object) that is part of the session context.
- To store data during the after_model_callback, you directly assign a key-value pair to it, and the ADK framework handles the persistence.
    - callback_context.state['my_extracted_data'] = parsed_value
- The data stored in callback_context.state is **Session-Scoped**, meaning it is persisted for the duration of that conversation thread and is accessible by various components across the agent's subsequent steps.

Function Tool vs. Agent Tool

- **Function Tool** is like calling a utility function in a script.
- **Agent Tool** wraps one complete agent as a utility for another agent.