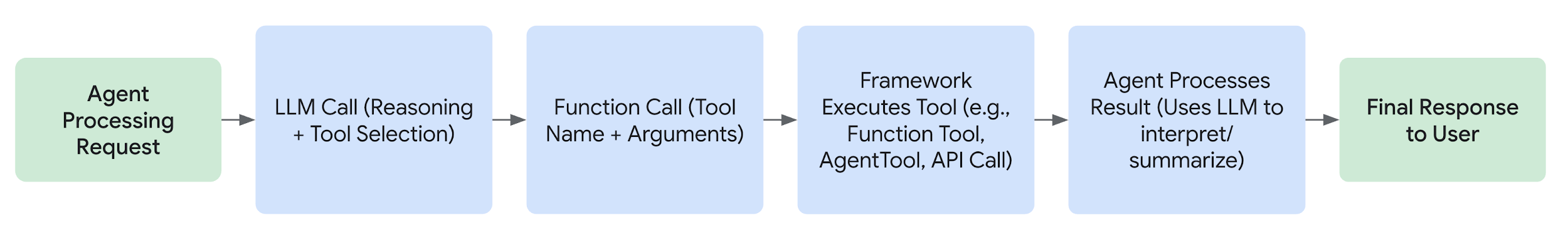
# Tools[¶](https://google.github.io/adk-docs/tools/#tools)

## What is a Tool?[¶](https://google.github.io/adk-docs/tools/#what-is-a-tool)

In the context of ADK, a Tool represents a specific capability provided to an AI agent, enabling it to perform actions and interact with the world beyond its core text generation and reasoning abilities. What distinguishes capable agents from basic language models is often their effective use of tools.

Technically, a tool is typically a modular code component—**like a Python/ Java function**, a class method, or even another specialized agent—designed to execute a distinct, predefined task. These tasks often involve interacting with external systems or data.



### Key Characteristics[¶](https://google.github.io/adk-docs/tools/#key-characteristics)

**Action-Oriented:** Tools perform specific actions, such as:

* Querying databases
* Making API requests (e.g., fetching weather data, booking systems)
* Searching the web
* Executing code snippets
* Retrieving information from documents (RAG)
* Interacting with other software or services

**Extends Agent capabilities:** They empower agents to access real-time information, affect external systems, and overcome the knowledge limitations inherent in their training data.

**Execute predefined logic:** Crucially, tools execute specific, developer-defined logic. They do not possess their own independent reasoning capabilities like the agent's core Large Language Model (LLM). The LLM reasons about which tool to use, when, and with what inputs, but the tool itself just executes its designated function.

## How Agents Use Tools[¶](https://google.github.io/adk-docs/tools/#how-agents-use-tools)

Agents leverage tools dynamically through mechanisms often involving function calling. The process generally follows these steps:

1. **Reasoning:** The agent's LLM analyzes its system instruction, conversation history, and user request.
2. **Selection:** Based on the analysis, the LLM decides on which tool, if any, to execute, based on the tools available to the agent and the docstrings that describes each tool.
3. **Invocation:** The LLM generates the required arguments (inputs) for the selected tool and triggers its execution.
4. **Observation:** The agent receives the output (result) returned by the tool.
5. **Finalization:** The agent incorporates the tool's output into its ongoing reasoning process to formulate the next response, decide the subsequent step, or determine if the goal has been achieved.

Think of the tools as a specialized toolkit that the agent's intelligent core (the LLM) can access and utilize as needed to accomplish complex tasks.

## Tool Types in ADK[¶](https://google.github.io/adk-docs/tools/#tool-types-in-adk)

ADK offers flexibility by supporting several types of tools:

1. [**Function Tools**](https://google.github.io/adk-docs/tools/function-tools/)**:** Tools created by you, tailored to your specific application's needs.
   * [**Functions/Methods**](https://google.github.io/adk-docs/tools/function-tools/#1-function-tool)**:** Define standard synchronous functions or methods in your code (e.g., Python def).
   * [**Agents-as-Tools**](https://google.github.io/adk-docs/tools/function-tools/#3-agent-as-a-tool)**:** Use another, potentially specialized, agent as a tool for a parent agent.
   * [**Long Running Function Tools**](https://google.github.io/adk-docs/tools/function-tools/#2-long-running-function-tool)**:** Support for tools that perform asynchronous operations or take significant time to complete.
2. [**Built-in Tools**](https://google.github.io/adk-docs/tools/built-in-tools/)**:** Ready-to-use tools provided by the framework for common tasks. Examples: Google Search, Code Execution, Retrieval-Augmented Generation (RAG).
3. [**Third-Party Tools**](https://google.github.io/adk-docs/tools/third-party-tools/)**:** Integrate tools seamlessly from popular external libraries. Examples: LangChain Tools, CrewAI Tools.

Navigate to the respective documentation pages linked above for detailed information and examples for each tool type.

## Referencing Tool in Agent’s Instructions[¶](https://google.github.io/adk-docs/tools/#referencing-tool-in-agents-instructions)

Within an agent's instructions, you can directly reference a tool by using its **function name.** If the tool's **function name** and **docstring** are sufficiently descriptive, your instructions can primarily focus on **when the Large Language Model (LLM) should utilize the tool**. This promotes clarity and helps the model understand the intended use of each tool.

It is **crucial to clearly instruct the agent on how to handle different return values** that a tool might produce. For example, if a tool returns an error message, your instructions should specify whether the agent should retry the operation, give up on the task, or request additional information from the user.

Furthermore, ADK supports the sequential use of tools, where the output of one tool can serve as the input for another. When implementing such workflows, it's important to **describe the intended sequence of tool usage** within the agent's instructions to guide the model through the necessary steps.

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

The following example showcases how an agent can use tools by **referencing their function names in its instructions**. It also demonstrates how to guide the agent to **handle different return values from tools**, such as success or error messages, and how to orchestrate the **sequential use of multiple tools** to accomplish a task.

**[Python](https://google.github.io/adk-docs/tools/#python)**

**[Java](https://google.github.io/adk-docs/tools/#java)**

from google.adk.agents import Agent

from google.adk.tools import FunctionTool

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

APP\_NAME="weather\_sentiment\_agent"

USER\_ID="user1234"

SESSION\_ID="1234"

MODEL\_ID="gemini-2.0-flash"

# Tool 1

def get\_weather\_report(city: str) -> dict:

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

Returns:

dict: A dictionary containing the weather information with a 'status' key ('success' or 'error') and a 'report' key with the weather details if successful, or an 'error\_message' if an error occurred.

"""

if city.lower() == "london":

return {"status": "success", "report": "The current weather in London is cloudy with a temperature of 18 degrees Celsius and a chance of rain."}

elif city.lower() == "paris":

return {"status": "success", "report": "The weather in Paris is sunny with a temperature of 25 degrees Celsius."}

else:

return {"status": "error", "error\_message": f"Weather information for '{city}' is not available."}

weather\_tool = FunctionTool(func=get\_weather\_report)

# Tool 2

def analyze\_sentiment(text: str) -> dict:

"""Analyzes the sentiment of the given text.

Returns:

dict: A dictionary with 'sentiment' ('positive', 'negative', or 'neutral') and a 'confidence' score.

"""

if "good" in text.lower() or "sunny" in text.lower():

return {"sentiment": "positive", "confidence": 0.8}

elif "rain" in text.lower() or "bad" in text.lower():

return {"sentiment": "negative", "confidence": 0.7}

else:

return {"sentiment": "neutral", "confidence": 0.6}

sentiment\_tool = FunctionTool(func=analyze\_sentiment)

# Agent

weather\_sentiment\_agent = Agent(

model=MODEL\_ID,

name='weather\_sentiment\_agent',

instruction="""You are a helpful assistant that provides weather information and analyzes the sentiment of user feedback.

\*\*If the user asks about the weather in a specific city, use the 'get\_weather\_report' tool to retrieve the weather details.\*\*

\*\*If the 'get\_weather\_report' tool returns a 'success' status, provide the weather report to the user.\*\*

\*\*If the 'get\_weather\_report' tool returns an 'error' status, inform the user that the weather information for the specified city is not available and ask if they have another city in mind.\*\*

\*\*After providing a weather report, if the user gives feedback on the weather (e.g., 'That's good' or 'I don't like rain'), use the 'analyze\_sentiment' tool to understand their sentiment.\*\* Then, briefly acknowledge their sentiment.

You can handle these tasks sequentially if needed.""",

tools=[weather\_tool, sentiment\_tool]

)

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=weather\_sentiment\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

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

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

for event in events:

if event.is\_final\_response():

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

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

call\_agent("weather in london?")

## Tool Context[¶](https://google.github.io/adk-docs/tools/#tool-context)

For more advanced scenarios, ADK allows you to access additional contextual information within your tool function by including the special parameter tool\_context: ToolContext. By including this in the function signature, ADK will **automatically** provide an **instance of the ToolContext** class when your tool is called during agent execution.

The **ToolContext** provides access to several key pieces of information and control levers:

* state: State: Read and modify the current session's state. Changes made here are tracked and persisted.
* actions: EventActions: Influence the agent's subsequent actions after the tool runs (e.g., skip summarization, transfer to another agent).
* function\_call\_id: str: The unique identifier assigned by the framework to this specific invocation of the tool. Useful for tracking and correlating with authentication responses. This can also be helpful when multiple tools are called within a single model response.
* function\_call\_event\_id: str: This attribute provides the unique identifier of the **event** that triggered the current tool call. This can be useful for tracking and logging purposes.
* auth\_response: Any: Contains the authentication response/credentials if an authentication flow was completed before this tool call.
* Access to Services: Methods to interact with configured services like Artifacts and Memory.

Note that you shouldn't include the tool\_context parameter in the tool function docstring. Since ToolContext is automatically injected by the ADK framework *after* the LLM decides to call the tool function, it is not relevant for the LLM's decision-making and including it can confuse the LLM.

### **State Management**[¶](https://google.github.io/adk-docs/tools/#state-management)

The tool\_context.state attribute provides direct read and write access to the state associated with the current session. It behaves like a dictionary but ensures that any modifications are tracked as deltas and persisted by the session service. This enables tools to maintain and share information across different interactions and agent steps.

* **Reading State**: Use standard dictionary access (tool\_context.state['my\_key']) or the .get() method (tool\_context.state.get('my\_key', default\_value)).
* **Writing State**: Assign values directly (tool\_context.state['new\_key'] = 'new\_value'). These changes are recorded in the state\_delta of the resulting event.
* **State Prefixes**: Remember the standard state prefixes:
  + app:\*: Shared across all users of the application.
  + user:\*: Specific to the current user across all their sessions.
  + (No prefix): Specific to the current session.
  + temp:\*: Temporary, not persisted across invocations (useful for passing data within a single run call but generally less useful inside a tool context which operates between LLM calls).

**[Python](https://google.github.io/adk-docs/tools/#python_1)**

**[Java](https://google.github.io/adk-docs/tools/#java_1)**

from google.adk.tools import ToolContext, FunctionTool

def update\_user\_preference(preference: str, value: str, tool\_context: ToolContext):

"""Updates a user-specific preference."""

user\_prefs\_key = "user:preferences"

# Get current preferences or initialize if none exist

preferences = tool\_context.state.get(user\_prefs\_key, {})

preferences[preference] = value

# Write the updated dictionary back to the state

tool\_context.state[user\_prefs\_key] = preferences

print(f"Tool: Updated user preference '{preference}' to '{value}'")

return {"status": "success", "updated\_preference": preference}

pref\_tool = FunctionTool(func=update\_user\_preference)

# In an Agent:

# my\_agent = Agent(..., tools=[pref\_tool])

# When the LLM calls update\_user\_preference(preference='theme', value='dark', ...):

# The tool\_context.state will be updated, and the change will be part of the

# resulting tool response event's actions.state\_delta.

### **Controlling Agent Flow**[¶](https://google.github.io/adk-docs/tools/#controlling-agent-flow)

The tool\_context.actions attribute (ToolContext.actions() in Java) holds an **EventActions** object. Modifying attributes on this object allows your tool to influence what the agent or framework does after the tool finishes execution.

* **skip\_summarization: bool**: (Default: False) If set to True, instructs the ADK to bypass the LLM call that typically summarizes the tool's output. This is useful if your tool's return value is already a user-ready message.
* **transfer\_to\_agent: str**: Set this to the name of another agent. The framework will halt the current agent's execution and **transfer control of the conversation to the specified agent**. This allows tools to dynamically hand off tasks to more specialized agents.
* **escalate: bool**: (Default: False) Setting this to True signals that the current agent cannot handle the request and should pass control up to its parent agent (if in a hierarchy). In a LoopAgent, setting **escalate=True** in a sub-agent's tool will terminate the loop.

#### **Example**[**¶**](https://google.github.io/adk-docs/tools/#example_1)

**[Python](https://google.github.io/adk-docs/tools/#python_2)**

**[Java](https://google.github.io/adk-docs/tools/#java_2)**

from google.adk.agents import Agent

from google.adk.tools import FunctionTool

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools import ToolContext

from google.genai import types

APP\_NAME="customer\_support\_agent"

USER\_ID="user1234"

SESSION\_ID="1234"

def check\_and\_transfer(query: str, tool\_context: ToolContext) -> str:

"""Checks if the query requires escalation and transfers to another agent if needed."""

if "urgent" in query.lower():

print("Tool: Detected urgency, transferring to the support agent.")

tool\_context.actions.transfer\_to\_agent = "support\_agent"

return "Transferring to the support agent..."

else:

return f"Processed query: '{query}'. No further action needed."

escalation\_tool = FunctionTool(func=check\_and\_transfer)

main\_agent = Agent(

model='gemini-2.0-flash',

name='main\_agent',

instruction="""You are the first point of contact for customer support of an analytics tool. Answer general queries. If the user indicates urgency, use the 'check\_and\_transfer' tool.""",

tools=[check\_and\_transfer]

)

support\_agent = Agent(

model='gemini-2.0-flash',

name='support\_agent',

instruction="""You are the dedicated support agent. Mentioned you are a support handler and please help the user with their urgent issue."""

)

main\_agent.sub\_agents = [support\_agent]

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=main\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

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

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

for event in events:

if event.is\_final\_response():

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

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

call\_agent("this is urgent, i cant login")

##### **Explanation**[**¶**](https://google.github.io/adk-docs/tools/#explanation)

* We define two agents: main\_agent and support\_agent. The main\_agent is designed to be the initial point of contact.
* The check\_and\_transfer tool, when called by main\_agent, examines the user's query.
* If the query contains the word "urgent", the tool accesses the tool\_context, specifically **tool\_context.actions**, and sets the transfer\_to\_agent attribute to support\_agent.
* This action signals to the framework to **transfer the control of the conversation to the agent named support\_agent**.
* When the main\_agent processes the urgent query, the check\_and\_transfer tool triggers the transfer. The subsequent response would ideally come from the support\_agent.
* For a normal query without urgency, the tool simply processes it without triggering a transfer.

This example illustrates how a tool, through EventActions in its ToolContext, can dynamically influence the flow of the conversation by transferring control to another specialized agent.

### **Authentication**[¶](https://google.github.io/adk-docs/tools/#authentication)

python_only

ToolContext provides mechanisms for tools interacting with authenticated APIs. If your tool needs to handle authentication, you might use the following:

* **auth\_response**: Contains credentials (e.g., a token) if authentication was already handled by the framework before your tool was called (common with RestApiTool and OpenAPI security schemes).
* **request\_credential(auth\_config: dict)**: Call this method if your tool determines authentication is needed but credentials aren't available. This signals the framework to start an authentication flow based on the provided auth\_config.
* **get\_auth\_response()**: Call this in a subsequent invocation (after request\_credential was successfully handled) to retrieve the credentials the user provided.

For detailed explanations of authentication flows, configuration, and examples, please refer to the dedicated Tool Authentication documentation page.

### **Context-Aware Data Access Methods**[¶](https://google.github.io/adk-docs/tools/#context-aware-data-access-methods)

These methods provide convenient ways for your tool to interact with persistent data associated with the session or user, managed by configured services.

* **list\_artifacts()** (or **listArtifacts()** in Java): Returns a list of filenames (or keys) for all artifacts currently stored for the session via the artifact\_service. Artifacts are typically files (images, documents, etc.) uploaded by the user or generated by tools/agents.
* **load\_artifact(filename: str)**: Retrieves a specific artifact by its filename from the **artifact\_service**. You can optionally specify a version; if omitted, the latest version is returned. Returns a google.genai.types.Part object containing the artifact data and mime type, or None if not found.
* **save\_artifact(filename: str, artifact: types.Part)**: Saves a new version of an artifact to the artifact\_service. Returns the new version number (starting from 0).
* **search\_memory(query: str)** python_only  
  Queries the user's long-term memory using the configured memory\_service. This is useful for retrieving relevant information from past interactions or stored knowledge. The structure of the **SearchMemoryResponse** depends on the specific memory service implementation but typically contains relevant text snippets or conversation excerpts.

#### **Example**[**¶**](https://google.github.io/adk-docs/tools/#example_2)

**[Python](https://google.github.io/adk-docs/tools/#python_3)**

**[Java](https://google.github.io/adk-docs/tools/#java_3)**

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

from google.adk.tools import ToolContext, FunctionTool

from google.genai import types

def process\_document(

document\_name: str, analysis\_query: str, tool\_context: ToolContext

) -> dict:

"""Analyzes a document using context from memory."""

# 1. Load the artifact

print(f"Tool: Attempting to load artifact: {document\_name}")

document\_part = tool\_context.load\_artifact(document\_name)

if not document\_part:

return {"status": "error", "message": f"Document '{document\_name}' not found."}

document\_text = document\_part.text # Assuming it's text for simplicity

print(f"Tool: Loaded document '{document\_name}' ({len(document\_text)} chars).")

# 2. Search memory for related context

print(f"Tool: Searching memory for context related to: '{analysis\_query}'")

memory\_response = tool\_context.search\_memory(

f"Context for analyzing document about {analysis\_query}"

)

memory\_context = "\n".join(

[

m.events[0].content.parts[0].text

for m in memory\_response.memories

if m.events and m.events[0].content

]

) # Simplified extraction

print(f"Tool: Found memory context: {memory\_context[:100]}...")

# 3. Perform analysis (placeholder)

analysis\_result = f"Analysis of '{document\_name}' regarding '{analysis\_query}' using memory context: [Placeholder Analysis Result]"

print("Tool: Performed analysis.")

# 4. Save the analysis result as a new artifact

analysis\_part = types.Part.from\_text(text=analysis\_result)

new\_artifact\_name = f"analysis\_{document\_name}"

version = await tool\_context.save\_artifact(new\_artifact\_name, analysis\_part)

print(f"Tool: Saved analysis result as '{new\_artifact\_name}' version {version}.")

return {

"status": "success",

"analysis\_artifact": new\_artifact\_name,

"version": version,

}

doc\_analysis\_tool = FunctionTool(func=process\_document)

# In an Agent:

# Assume artifact 'report.txt' was previously saved.

# Assume memory service is configured and has relevant past data.

# my\_agent = Agent(..., tools=[doc\_analysis\_tool], artifact\_service=..., memory\_service=...)

By leveraging the **ToolContext**, developers can create more sophisticated and context-aware custom tools that seamlessly integrate with ADK's architecture and enhance the overall capabilities of their agents.

## Defining Effective Tool Functions[¶](https://google.github.io/adk-docs/tools/#defining-effective-tool-functions)

When using a method or function as an ADK Tool, how you define it significantly impacts the agent's ability to use it correctly. The agent's Large Language Model (LLM) relies heavily on the function's **name**, **parameters (arguments)**, **type hints**, and **docstring** / **source code comments** to understand its purpose and generate the correct call.

Here are key guidelines for defining effective tool functions:

* **Function Name:**
  + Use descriptive, verb-noun based names that clearly indicate the action (e.g., get\_weather, searchDocuments, schedule\_meeting).
  + Avoid generic names like run, process, handle\_data, or overly ambiguous names like doStuff. Even with a good description, a name like do\_stuff might confuse the model about when to use the tool versus, for example, cancelFlight.
  + The LLM uses the function name as a primary identifier during tool selection.
* **Parameters (Arguments):**
  + Your function can have any number of parameters.
  + Use clear and descriptive names (e.g., city instead of c, search\_query instead of q).
  + **Provide type hints in Python** for all parameters (e.g., city: str, user\_id: int, items: list[str]). This is essential for ADK to generate the correct schema for the LLM.
  + Ensure all parameter types are **JSON serializable**. All java primitives as well as standard Python types like str, int, float, bool, list, dict, and their combinations are generally safe. Avoid complex custom class instances as direct parameters unless they have a clear JSON representation.
  + **Do not set default values** for parameters. E.g., def my\_func(param1: str = "default"). Default values are not reliably supported or used by the underlying models during function call generation. All necessary information should be derived by the LLM from the context or explicitly requested if missing.
* **Return Type:**
  + The function's return value **must be a dictionary (dict)** in Python or a **Map** in Java.
  + If your function returns a non-dictionary type (e.g., a string, number, list), the ADK framework will automatically wrap it into a dictionary/Map like {'result': your\_original\_return\_value} before passing the result back to the model.
  + Design the dictionary/Map keys and values to be **descriptive and easily understood *by the LLM***. Remember, the model reads this output to decide its next step.
  + Include meaningful keys. For example, instead of returning just an error code like 500, return {'status': 'error', 'error\_message': 'Database connection failed'}.
  + It's a **highly recommended practice** to include a status key (e.g., 'success', 'error', 'pending', 'ambiguous') to clearly indicate the outcome of the tool execution for the model.
* **Docstring / Source Code Comments:**
  + **This is critical.** The docstring is the primary source of descriptive information for the LLM.
  + **Clearly state what the tool *does*.** Be specific about its purpose and limitations.
  + **Explain *when* the tool should be used.** Provide context or example scenarios to guide the LLM's decision-making.
  + **Describe *each parameter* clearly.** Explain what information the LLM needs to provide for that argument.
  + Describe the **structure and meaning of the expected dict return value**, especially the different status values and associated data keys.
  + **Do not describe the injected ToolContext parameter**. Avoid mentioning the optional tool\_context: ToolContext parameter within the docstring description since it is not a parameter the LLM needs to know about. ToolContext is injected by ADK, *after* the LLM decides to call it.
* **Example of a good definition:**

**[Python](https://google.github.io/adk-docs/tools/#python_4)**

**[Java](https://google.github.io/adk-docs/tools/#java_4)**

def lookup\_order\_status(order\_id: str) -> dict:

"""Fetches the current status of a customer's order using its ID.

Use this tool ONLY when a user explicitly asks for the status of

a specific order and provides the order ID. Do not use it for

general inquiries.

Args:

order\_id: The unique identifier of the order to look up.

Returns:

A dictionary containing the order status.

Possible statuses: 'shipped', 'processing', 'pending', 'error'.

Example success: {'status': 'shipped', 'tracking\_number': '1Z9...'}

Example error: {'status': 'error', 'error\_message': 'Order ID not found.'}

"""

# ... function implementation to fetch status ...

if status := fetch\_status\_from\_backend(order\_id):

return {"status": status.state, "tracking\_number": status.tracking} # Example structure

else:

return {"status": "error", "error\_message": f"Order ID {order\_id} not found."}

* **Simplicity and Focus:**
  + **Keep Tools Focused:** Each tool should ideally perform one well-defined task.
  + **Fewer Parameters are Better:** Models generally handle tools with fewer, clearly defined parameters more reliably than those with many optional or complex ones.
  + **Use Simple Data Types:** Prefer basic types (str, int, bool, float, List[str], in **Python**, or int, byte, short, long, float, double, boolean and char in **Java**) over complex custom classes or deeply nested structures as parameters when possible.
  + **Decompose Complex Tasks:** Break down functions that perform multiple distinct logical steps into smaller, more focused tools. For instance, instead of a single update\_user\_profile(profile: ProfileObject) tool, consider separate tools like update\_user\_name(name: str), update\_user\_address(address: str), update\_user\_preferences(preferences: list[str]), etc. This makes it easier for the LLM to select and use the correct capability.

By adhering to these guidelines, you provide the LLM with the clarity and structure it needs to effectively utilize your custom function tools, leading to more capable and reliable agent behavior.

## Toolsets: Grouping and Dynamically Providing Tools python_only[¶](https://google.github.io/adk-docs/tools/#toolsets-grouping-and-dynamically-providing-tools)

Beyond individual tools, ADK introduces the concept of a **Toolset** via the BaseToolset interface (defined in google.adk.tools.base\_toolset). A toolset allows you to manage and provide a collection of BaseTool instances, often dynamically, to an agent.

This approach is beneficial for:

* **Organizing Related Tools:** Grouping tools that serve a common purpose (e.g., all tools for mathematical operations, or all tools interacting with a specific API).
* **Dynamic Tool Availability:** Enabling an agent to have different tools available based on the current context (e.g., user permissions, session state, or other runtime conditions). The get\_tools method of a toolset can decide which tools to expose.
* **Integrating External Tool Providers:** Toolsets can act as adapters for tools coming from external systems, like an OpenAPI specification or an MCP server, converting them into ADK-compatible BaseTool objects.

### The BaseToolset Interface[¶](https://google.github.io/adk-docs/tools/#the-basetoolset-interface)

Any class acting as a toolset in ADK should implement the BaseToolset abstract base class. This interface primarily defines two methods:

* **async def get\_tools(...) -> list[BaseTool]:** This is the core method of a toolset. When an ADK agent needs to know its available tools, it will call get\_tools() on each BaseToolset instance provided in its tools list.
  + It receives an optional readonly\_context (an instance of ReadonlyContext). This context provides read-only access to information like the current session state (readonly\_context.state), agent name, and invocation ID. The toolset can use this context to dynamically decide which tools to return.
  + It **must** return a list of BaseTool instances (e.g., FunctionTool, RestApiTool).
* **async def close(self) -> None:** This asynchronous method is called by the ADK framework when the toolset is no longer needed, for example, when an agent server is shutting down or the Runner is being closed. Implement this method to perform any necessary cleanup, such as closing network connections, releasing file handles, or cleaning up other resources managed by the toolset.

### Using Toolsets with Agents[¶](https://google.github.io/adk-docs/tools/#using-toolsets-with-agents)

You can include instances of your BaseToolset implementations directly in an LlmAgent's tools list, alongside individual BaseTool instances.

When the agent initializes or needs to determine its available capabilities, the ADK framework will iterate through the tools list:

* If an item is a BaseTool instance, it's used directly.
* If an item is a BaseToolset instance, its get\_tools() method is called (with the current ReadonlyContext), and the returned list of BaseTools is added to the agent's available tools.

### Example: A Simple Math Toolset[¶](https://google.github.io/adk-docs/tools/#example-a-simple-math-toolset)

Let's create a basic example of a toolset that provides simple arithmetic operations.

# 1. Define the individual tool functions

def add\_numbers(a: int, b: int, tool\_context: ToolContext) -> Dict[str, Any]:

"""Adds two integer numbers.

Args:

a: The first number.

b: The second number.

Returns:

A dictionary with the sum, e.g., {'status': 'success', 'result': 5}

"""

print(f"Tool: add\_numbers called with a={a}, b={b}")

result = a + b

# Example: Storing something in tool\_context state

tool\_context.state["last\_math\_operation"] = "addition"

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

def subtract\_numbers(a: int, b: int) -> Dict[str, Any]:

"""Subtracts the second number from the first.

Args:

a: The first number.

b: The second number.

Returns:

A dictionary with the difference, e.g., {'status': 'success', 'result': 1}

"""

print(f"Tool: subtract\_numbers called with a={a}, b={b}")

return {"status": "success", "result": a - b}

# 2. Create the Toolset by implementing BaseToolset

class SimpleMathToolset(BaseToolset):

def \_\_init\_\_(self, prefix: str = "math\_"):

self.prefix = prefix

# Create FunctionTool instances once

self.\_add\_tool = FunctionTool(

func=add\_numbers,

name=f"{self.prefix}add\_numbers", # Toolset can customize names

)

self.\_subtract\_tool = FunctionTool(

func=subtract\_numbers, name=f"{self.prefix}subtract\_numbers"

)

print(f"SimpleMathToolset initialized with prefix '{self.prefix}'")

async def get\_tools(

self, readonly\_context: Optional[ReadonlyContext] = None

) -> List[BaseTool]:

print(f"SimpleMathToolset.get\_tools() called.")

# Example of dynamic behavior:

# Could use readonly\_context.state to decide which tools to return

# For instance, if readonly\_context.state.get("enable\_advanced\_math"):

# return [self.\_add\_tool, self.\_subtract\_tool, self.\_multiply\_tool]

# For this simple example, always return both tools

tools\_to\_return = [self.\_add\_tool, self.\_subtract\_tool]

print(f"SimpleMathToolset providing tools: {[t.name for t in tools\_to\_return]}")

return tools\_to\_return

async def close(self) -> None:

# No resources to clean up in this simple example

print(f"SimpleMathToolset.close() called for prefix '{self.prefix}'.")

await asyncio.sleep(0) # Placeholder for async cleanup if needed

# 3. Define an individual tool (not part of the toolset)

def greet\_user(name: str = "User") -> Dict[str, str]:

"""Greets the user."""

print(f"Tool: greet\_user called with name={name}")

return {"greeting": f"Hello, {name}!"}

greet\_tool = FunctionTool(func=greet\_user)

# 4. Instantiate the toolset

math\_toolset\_instance = SimpleMathToolset(prefix="calculator\_")

# 5. Define an agent that uses both the individual tool and the toolset

calculator\_agent = LlmAgent(

name="CalculatorAgent",

model="gemini-2.0-flash", # Replace with your desired model

instruction="You are a helpful calculator and greeter. "

"Use 'greet\_user' for greetings. "

"Use 'calculator\_add\_numbers' to add and 'calculator\_subtract\_numbers' to subtract. "

"Announce the state of 'last\_math\_operation' if it's set.",

tools=[greet\_tool, math\_toolset\_instance], # Individual tool # Toolset instance

)

In this example:

* SimpleMathToolset implements BaseToolset and its get\_tools() method returns FunctionTool instances for add\_numbers and subtract\_numbers. It also customizes their names using a prefix.
* The calculator\_agent is configured with both an individual greet\_tool and an instance of SimpleMathToolset.
* When calculator\_agent is run, ADK will call math\_toolset\_instance.get\_tools(). The agent's LLM will then have access to greet\_user, calculator\_add\_numbers, and calculator\_subtract\_numbers to handle user requests.
* The add\_numbers tool demonstrates writing to tool\_context.state, and the agent's instruction mentions reading this state.
* The close() method is called to ensure any resources held by the toolset are released.

Toolsets offer a powerful way to organize, manage, and dynamically provide collections of tools to your ADK agents, leading to more modular, maintainable, and adaptable agentic applications.

# Function tools[¶](https://google.github.io/adk-docs/tools/function-tools/#function-tools)

## What are function tools?[¶](https://google.github.io/adk-docs/tools/function-tools/#what-are-function-tools)

When out-of-the-box tools don't fully meet specific requirements, developers can create custom function tools. This allows for **tailored functionality**, such as connecting to proprietary databases or implementing unique algorithms.

*For example,* a function tool, "myfinancetool", might be a function that calculates a specific financial metric. ADK also supports long running functions, so if that calculation takes a while, the agent can continue working on other tasks.

ADK offers several ways to create functions tools, each suited to different levels of complexity and control:

1. Function Tool
2. Long Running Function Tool
3. Agents-as-a-Tool

## 1. Function Tool[¶](https://google.github.io/adk-docs/tools/function-tools/#1-function-tool)

Transforming a function into a tool is a straightforward way to integrate custom logic into your agents. In fact, when you assign a function to an agent’s tools list, the framework will automatically wrap it as a Function Tool for you. This approach offers flexibility and quick integration.

### Parameters[¶](https://google.github.io/adk-docs/tools/function-tools/#parameters)

Define your function parameters using standard **JSON-serializable types** (e.g., string, integer, list, dictionary). It's important to avoid setting default values for parameters, as the language model (LLM) does not currently support interpreting them.

### Return Type[¶](https://google.github.io/adk-docs/tools/function-tools/#return-type)

The preferred return type for a Function Tool is a **dictionary** in Python or **Map** in Java. This allows you to structure the response with key-value pairs, providing context and clarity to the LLM. If your function returns a type other than a dictionary, the framework automatically wraps it into a dictionary with a single key named **"result"**.

Strive to make your return values as descriptive as possible. *For example,* instead of returning a numeric error code, return a dictionary with an "error\_message" key containing a human-readable explanation. **Remember that the LLM**, not a piece of code, needs to understand the result. As a best practice, include a "status" key in your return dictionary to indicate the overall outcome (e.g., "success", "error", "pending"), providing the LLM with a clear signal about the operation's state.

### Docstring / Source code comments[¶](https://google.github.io/adk-docs/tools/function-tools/#docstring-source-code-comments)

The docstring (or comments above) your function serve as the tool's description and is sent to the LLM. Therefore, a well-written and comprehensive docstring is crucial for the LLM to understand how to use the tool effectively. Clearly explain the purpose of the function, the meaning of its parameters, and the expected return values.

**Example**

### Best Practices[¶](https://google.github.io/adk-docs/tools/function-tools/#best-practices)

While you have considerable flexibility in defining your function, remember that simplicity enhances usability for the LLM. Consider these guidelines:

* **Fewer Parameters are Better:** Minimize the number of parameters to reduce complexity.
* **Simple Data Types:** Favor primitive data types like str and int over custom classes whenever possible.
* **Meaningful Names:** The function's name and parameter names significantly influence how the LLM interprets and utilizes the tool. Choose names that clearly reflect the function's purpose and the meaning of its inputs. Avoid generic names like do\_stuff() or beAgent().

## 2. Long Running Function Tool[¶](https://google.github.io/adk-docs/tools/function-tools/#2-long-running-function-tool)

Designed for tasks that require a significant amount of processing time without blocking the agent's execution. This tool is a subclass of FunctionTool.

When using a LongRunningFunctionTool, your function can initiate the long-running operation and optionally return an **initial result**\*\* (e.g. the long-running operation id). Once a long running function tool is invoked the agent runner will pause the agent run and let the agent client to decide whether to continue or wait until the long-running operation finishes. The agent client can query the progress of the long-running operation and send back an intermediate or final response. The agent can then continue with other tasks. An example is the human-in-the-loop scenario where the agent needs human approval before proceeding with a task.

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

In Python, you wrap a function with LongRunningFunctionTool. In Java, you pass a Method name to LongRunningFunctionTool.create().

1. **Initiation:** When the LLM calls the tool, your function starts the long-running operation.
2. **Initial Updates:** Your function should optionally return an initial result (e.g. the long-running operaiton id). The ADK framework takes the result and sends it back to the LLM packaged within a FunctionResponse. This allows the LLM to inform the user (e.g., status, percentage complete, messages). And then the agent run is ended / paused.
3. **Continue or Wait:** After each agent run is completed. Agent client can query the progress of the long-running operation and decide whether to continue the agent run with an intermediate response (to update the progress) or wait until a final response is retrieved. Agent client should send the intermediate or final response back to the agent for the next run.
4. **Framework Handling:** The ADK framework manages the execution. It sends the intermediate or final FunctionResponse sent by agent client to the LLM to generate a user friendly message.

### Creating the Tool[¶](https://google.github.io/adk-docs/tools/function-tools/#creating-the-tool)

Define your tool function and wrap it using the LongRunningFunctionTool class:

**[Python](https://google.github.io/adk-docs/tools/function-tools/#python_1)**

**[Java](https://google.github.io/adk-docs/tools/function-tools/#java_1)**

from google.adk.tools import LongRunningFunctionTool

# Define your long running function (see example below)

def ask\_for\_approval(

purpose: str, amount: float, tool\_context: ToolContext

) -> dict[str, Any]:

"""Ask for approval for the reimbursement."""

# create a ticket for the approval

# Send a notification to the approver with the link of the ticket

return {'status': 'pending', 'approver': 'Sean Zhou', 'purpose' : purpose, 'amount': amount, 'ticket-id': 'approval-ticket-1'}

# Wrap the function

approve\_tool = LongRunningFunctionTool(func=ask\_for\_approval)

### Intermediate / Final result Updates[¶](https://google.github.io/adk-docs/tools/function-tools/#intermediate-final-result-updates)

Agent client received an event with long running function calls and check the status of the ticket. Then Agent client can send the intermediate or final response back to update the progress. The framework packages this value (even if it's None) into the content of the FunctionResponse sent back to the LLM.

**Applies to only Java ADK**

When passing ToolContext with Function Tools, ensure that one of the following is true:

* The Schema is passed with the ToolContext parameter in the function signature, like:

@com.google.adk.tools.Annotations.Schema(name = "toolContext") ToolContext toolContext

* OR
* The following -parameters flag is set to the mvn compiler plugin

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.14.0</version> <!-- or newer -->

<configuration>

<compilerArgs>

<arg>-parameters</arg>

</compilerArgs>

</configuration>

</plugin>

</plugins>

</build>

This constraint is temporary and will be removed.

**[Python](https://google.github.io/adk-docs/tools/function-tools/#python_2)**

**[Java](https://google.github.io/adk-docs/tools/function-tools/#java_2)**

# runner = Runner(...)

# session = await session\_service.create\_session(...)

# content = types.Content(...) # User's initial query

def get\_long\_running\_function\_call(event: Event) -> types.FunctionCall:

# Get the long running function call from the event

if not event.long\_running\_tool\_ids or not event.content or not event.content.parts:

return

for part in event.content.parts:

if (

part

and part.function\_call

and event.long\_running\_tool\_ids

and part.function\_call.id in event.long\_running\_tool\_ids

):

return part.function\_call

def get\_function\_response(event: Event, function\_call\_id: str) -> types.FunctionResponse:

# Get the function response for the fuction call with specified id.

if not event.content or not event.content.parts:

return

for part in event.content.parts:

if (

part

and part.function\_response

and part.function\_response.id == function\_call\_id

):

return part.function\_response

print("\nRunning agent...")

events\_async = runner.run\_async(

session\_id=session.id, user\_id='user', new\_message=content

)

long\_running\_function\_call, long\_running\_function\_response, ticket\_id = None, None, None

async for event in events\_async:

# Use helper to check for the specific auth request event

if not long\_running\_function\_call:

long\_running\_function\_call = get\_long\_running\_function\_call(event)

else:

long\_running\_function\_response = get\_function\_response(event, long\_running\_function\_call.id)

if long\_running\_function\_response:

ticket\_id = long\_running\_function\_response.response['ticket\_id']

if event.content and event.content.parts:

if text := ''.join(part.text or '' for part in event.content.parts):

print(f'[{event.author}]: {text}')

if long\_running\_function\_response:

# query the status of the correpsonding ticket via tciket\_id

# send back an intermediate / final response

updated\_response = long\_running\_function\_response.model\_copy(deep=True)

updated\_response.response = {'status': 'approved'}

async for event in runner.run\_async(

session\_id=session.id, user\_id='user', new\_message=types.Content(parts=[types.Part(function\_response = updated\_response)], role='user')

):

if event.content and event.content.parts:

if text := ''.join(part.text or '' for part in event.content.parts):

print(f'[{event.author}]: {text}')

**Example: File Processing Simulation**

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

import asyncio

from typing import Any

from google.adk.agents import Agent

from google.adk.events import Event

from google.adk.runners import Runner

from google.adk.tools import LongRunningFunctionTool

from google.adk.sessions import InMemorySessionService

from google.genai import types

# 1. Define the long running function

def ask\_for\_approval(

purpose: str, amount: float

) -> dict[str, Any]:

"""Ask for approval for the reimbursement."""

# create a ticket for the approval

# Send a notification to the approver with the link of the ticket

return {'status': 'pending', 'approver': 'Sean Zhou', 'purpose' : purpose, 'amount': amount, 'ticket-id': 'approval-ticket-1'}

def reimburse(purpose: str, amount: float) -> str:

"""Reimburse the amount of money to the employee."""

# send the reimbrusement request to payment vendor

return {'status': 'ok'}

# 2. Wrap the function with LongRunningFunctionTool

long\_running\_tool = LongRunningFunctionTool(func=ask\_for\_approval)

# 3. Use the tool in an Agent

file\_processor\_agent = Agent(

# Use a model compatible with function calling

model="gemini-2.0-flash",

name='reimbursement\_agent',

instruction="""

You are an agent whose job is to handle the reimbursement process for

the employees. If the amount is less than $100, you will automatically

approve the reimbursement.

If the amount is greater than $100, you will

ask for approval from the manager. If the manager approves, you will

call reimburse() to reimburse the amount to the employee. If the manager

rejects, you will inform the employee of the rejection.

""",

tools=[reimburse, long\_running\_tool]

)

APP\_NAME = "human\_in\_the\_loop"

USER\_ID = "1234"

SESSION\_ID = "session1234"

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=file\_processor\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

async def call\_agent(query):

def get\_long\_running\_function\_call(event: Event) -> types.FunctionCall:

# Get the long running function call from the event

if not event.long\_running\_tool\_ids or not event.content or not event.content.parts:

return

for part in event.content.parts:

if (

part

and part.function\_call

and event.long\_running\_tool\_ids

and part.function\_call.id in event.long\_running\_tool\_ids

):

return part.function\_call

def get\_function\_response(event: Event, function\_call\_id: str) -> types.FunctionResponse:

# Get the function response for the fuction call with specified id.

if not event.content or not event.content.parts:

return

for part in event.content.parts:

if (

part

and part.function\_response

and part.function\_response.id == function\_call\_id

):

return part.function\_response

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

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

print("\nRunning agent...")

events\_async = runner.run\_async(

session\_id=session.id, user\_id=USER\_ID, new\_message=content

)

long\_running\_function\_call, long\_running\_function\_response, ticket\_id = None, None, None

async for event in events\_async:

# Use helper to check for the specific auth request event

if not long\_running\_function\_call:

long\_running\_function\_call = get\_long\_running\_function\_call(event)

else:

long\_running\_function\_response = get\_function\_response(event, long\_running\_function\_call.id)

if long\_running\_function\_response:

ticket\_id = long\_running\_function\_response.response['ticket-id']

if event.content and event.content.parts:

if text := ''.join(part.text or '' for part in event.content.parts):

print(f'[{event.author}]: {text}')

if long\_running\_function\_response:

# query the status of the correpsonding ticket via tciket\_id

# send back an intermediate / final response

updated\_response = long\_running\_function\_response.model\_copy(deep=True)

updated\_response.response = {'status': 'approved'}

async for event in runner.run\_async(

session\_id=session.id, user\_id=USER\_ID, new\_message=types.Content(parts=[types.Part(function\_response = updated\_response)], role='user')

):

if event.content and event.content.parts:

if text := ''.join(part.text or '' for part in event.content.parts):

print(f'[{event.author}]: {text}')

# reimbursement that doesn't require approval

asyncio.run(call\_agent("Please reimburse 50$ for meals"))

# reimbursement that requires approval

asyncio.run(call\_agent("Please reimburse 200$ for meals"))

#### **Key aspects of this example**[**¶**](https://google.github.io/adk-docs/tools/function-tools/#key-aspects-of-this-example)

* **LongRunningFunctionTool**: Wraps the supplied method/function; the framework handles sending yielded updates and the final return value as sequential FunctionResponses.
* **Agent instruction**: Directs the LLM to use the tool and understand the incoming FunctionResponse stream (progress vs. completion) for user updates.
* **Final return**: The function returns the final result dictionary, which is sent in the concluding FunctionResponse to indicate completion.

## 3. Agent-as-a-Tool[¶](https://google.github.io/adk-docs/tools/function-tools/#3-agent-as-a-tool)

This powerful feature allows you to leverage the capabilities of other agents within your system by calling them as tools. The Agent-as-a-Tool enables you to invoke another agent to perform a specific task, effectively **delegating responsibility**. This is conceptually similar to creating a Python function that calls another agent and uses the agent's response as the function's return value.

### Key difference from sub-agents[¶](https://google.github.io/adk-docs/tools/function-tools/#key-difference-from-sub-agents)

It's important to distinguish an Agent-as-a-Tool from a Sub-Agent.

* **Agent-as-a-Tool:** When Agent A calls Agent B as a tool (using Agent-as-a-Tool), Agent B's answer is **passed back** to Agent A, which then summarizes the answer and generates a response to the user. Agent A retains control and continues to handle future user input.
* **Sub-agent:** When Agent A calls Agent B as a sub-agent, the responsibility of answering the user is completely **transferred to Agent B**. Agent A is effectively out of the loop. All subsequent user input will be answered by Agent B.

### Usage[¶](https://google.github.io/adk-docs/tools/function-tools/#usage)

To use an agent as a tool, wrap the agent with the AgentTool class.

**[Python](https://google.github.io/adk-docs/tools/function-tools/#python_3)**

**[Java](https://google.github.io/adk-docs/tools/function-tools/#java_3)**

tools=[AgentTool(agent=agent\_b)]

### Customization[¶](https://google.github.io/adk-docs/tools/function-tools/#customization)

The AgentTool class provides the following attributes for customizing its behavior:

* **skip\_summarization: bool:** If set to True, the framework will **bypass the LLM-based summarization** of the tool agent's response. This can be useful when the tool's response is already well-formatted and requires no further processing.

**Example**

**[Python](https://google.github.io/adk-docs/tools/function-tools/#python_4)**

**[Java](https://google.github.io/adk-docs/tools/function-tools/#java_4)**

from google.adk.agents import Agent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools.agent\_tool import AgentTool

from google.genai import types

APP\_NAME="summary\_agent"

USER\_ID="user1234"

SESSION\_ID="1234"

summary\_agent = Agent(

model="gemini-2.0-flash",

name="summary\_agent",

instruction="""You are an expert summarizer. Please read the following text and provide a concise summary.""",

description="Agent to summarize text",

)

root\_agent = Agent(

model='gemini-2.0-flash',

name='root\_agent',

instruction="""You are a helpful assistant. When the user provides a text, use the 'summarize' tool to generate a summary. Always forward the user's message exactly as received to the 'summarize' tool, without modifying or summarizing it yourself. Present the response from the tool to the user.""",

tools=[AgentTool(agent=summary\_agent)]

)

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=root\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

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

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

for event in events:

if event.is\_final\_response():

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

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

long\_text = """Quantum computing represents a fundamentally different approach to computation,

leveraging the bizarre principles of quantum mechanics to process information. Unlike classical computers

that rely on bits representing either 0 or 1, quantum computers use qubits which can exist in a state of superposition - effectively

being 0, 1, or a combination of both simultaneously. Furthermore, qubits can become entangled,

meaning their fates are intertwined regardless of distance, allowing for complex correlations. This parallelism and

interconnectedness grant quantum computers the potential to solve specific types of incredibly complex problems - such

as drug discovery, materials science, complex system optimization, and breaking certain types of cryptography - far

faster than even the most powerful classical supercomputers could ever achieve, although the technology is still largely in its developmental stages."""

call\_agent(long\_text)

### How it works[¶](https://google.github.io/adk-docs/tools/function-tools/#how-it-works_1)

1. When the main\_agent receives the long text, its instruction tells it to use the 'summarize' tool for long texts.
2. The framework recognizes 'summarize' as an AgentTool that wraps the summary\_agent.
3. Behind the scenes, the main\_agent will call the summary\_agent with the long text as input.
4. The summary\_agent will process the text according to its instruction and generate a summary.
5. **The response from the summary\_agent is then passed back to the main\_agent.**
6. The main\_agent can then take the summary and formulate its final response to the user (e.g., "Here's a summary of the text: ...")

# Built-in tools[¶](https://google.github.io/adk-docs/tools/built-in-tools/#built-in-tools)

These built-in tools provide ready-to-use functionality such as Google Search or code executors that provide agents with common capabilities. For instance, an agent that needs to retrieve information from the web can directly use the **google\_search** tool without any additional setup.

## How to Use[¶](https://google.github.io/adk-docs/tools/built-in-tools/#how-to-use)

1. **Import:** Import the desired tool from the tools module. This is agents.tools in Python or com.google.adk.tools in Java.
2. **Configure:** Initialize the tool, providing required parameters if any.
3. **Register:** Add the initialized tool to the **tools** list of your Agent.

Once added to an agent, the agent can decide to use the tool based on the **user prompt** and its **instructions**. The framework handles the execution of the tool when the agent calls it. Important: check the ***Limitations*** section of this page.

## Available Built-in tools[¶](https://google.github.io/adk-docs/tools/built-in-tools/#available-built-in-tools)

Note: Java only supports Google Search and Code Execition tools currently.

### Google Search[¶](https://google.github.io/adk-docs/tools/built-in-tools/#google-search)

The google\_search tool allows the agent to perform web searches using Google Search. The google\_search tool is only compatible with Gemini 2 models.

**Additional requirements when using the google\_search tool**

When you use grounding with Google Search, and you receive Search suggestions in your response, you must display the Search suggestions in production and in your applications. For more information on grounding with Google Search, see Grounding with Google Search documentation for [Google AI Studio](https://ai.google.dev/gemini-api/docs/grounding/search-suggestions) or [Vertex AI](https://cloud.google.com/vertex-ai/generative-ai/docs/grounding/grounding-search-suggestions). The UI code (HTML) is returned in the Gemini response as renderedContent, and you will need to show the HTML in your app, in accordance with the policy.

**[Python](https://google.github.io/adk-docs/tools/built-in-tools/#python)**

**[Java](https://google.github.io/adk-docs/tools/built-in-tools/#java)**

from google.adk.agents import Agent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools import google\_search

from google.genai import types

APP\_NAME="google\_search\_agent"

USER\_ID="user1234"

SESSION\_ID="1234"

root\_agent = Agent(

name="basic\_search\_agent",

model="gemini-2.0-flash",

description="Agent to answer questions using Google Search.",

instruction="I can answer your questions by searching the internet. Just ask me anything!",

# google\_search is a pre-built tool which allows the agent to perform Google searches.

tools=[google\_search]

)

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=root\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

"""

Helper function to call the agent with a query.

"""

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

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

for event in events:

if event.is\_final\_response():

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

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

call\_agent("what's the latest ai news?")

### Code Execution[¶](https://google.github.io/adk-docs/tools/built-in-tools/#code-execution)

The built\_in\_code\_execution tool enables the agent to execute code, specifically when using Gemini 2 models. This allows the model to perform tasks like calculations, data manipulation, or running small scripts.

**[Python](https://google.github.io/adk-docs/tools/built-in-tools/#python_1)**

**[Java](https://google.github.io/adk-docs/tools/built-in-tools/#java_1)**

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

import asyncio

from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.code\_executors import BuiltInCodeExecutor

from google.genai import types

AGENT\_NAME = "calculator\_agent"

APP\_NAME = "calculator"

USER\_ID = "user1234"

SESSION\_ID = "session\_code\_exec\_async"

GEMINI\_MODEL = "gemini-2.0-flash"

# Agent Definition

code\_agent = LlmAgent(

name=AGENT\_NAME,

model=GEMINI\_MODEL,

executor=[BuiltInCodeExecutor],

instruction="""You are a calculator agent.

When given a mathematical expression, write and execute Python code to calculate the result.

Return only the final numerical result as plain text, without markdown or code blocks.

""",

description="Executes Python code to perform calculations.",

)

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(

app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID

)

runner = Runner(agent=code\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction (Async)

async def call\_agent\_async(query):

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

print(f"\n--- Running Query: {query} ---")

final\_response\_text = "No final text response captured."

try:

# Use run\_async

async for event in runner.run\_async(

user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content

):

print(f"Event ID: {event.id}, Author: {event.author}")

# --- Check for specific parts FIRST ---

has\_specific\_part = False

if event.content and event.content.parts:

for part in event.content.parts: # Iterate through all parts

if part.executable\_code:

# Access the actual code string via .code

print(

f" Debug: Agent generated code:\n```python\n{part.executable\_code.code}\n```"

)

has\_specific\_part = True

elif part.code\_execution\_result:

# Access outcome and output correctly

print(

f" Debug: Code Execution Result: {part.code\_execution\_result.outcome} - Output:\n{part.code\_execution\_result.output}"

)

has\_specific\_part = True

# Also print any text parts found in any event for debugging

elif part.text and not part.text.isspace():

print(f" Text: '{part.text.strip()}'")

# Do not set has\_specific\_part=True here, as we want the final response logic below

# --- Check for final response AFTER specific parts ---

# Only consider it final if it doesn't have the specific code parts we just handled

if not has\_specific\_part and event.is\_final\_response():

if (

event.content

and event.content.parts

and event.content.parts[0].text

):

final\_response\_text = event.content.parts[0].text.strip()

print(f"==> Final Agent Response: {final\_response\_text}")

else:

print("==> Final Agent Response: [No text content in final event]")

except Exception as e:

print(f"ERROR during agent run: {e}")

print("-" \* 30)

# Main async function to run the examples

async def main():

await call\_agent\_async("Calculate the value of (5 + 7) \* 3")

await call\_agent\_async("What is 10 factorial?")

# Execute the main async function

try:

asyncio.run(main())

except RuntimeError as e:

# Handle specific error when running asyncio.run in an already running loop (like Jupyter/Colab)

if "cannot be called from a running event loop" in str(e):

print("\nRunning in an existing event loop (like Colab/Jupyter).")

print("Please run `await main()` in a notebook cell instead.")

# If in an interactive environment like a notebook, you might need to run:

# await main()

else:

raise e # Re-raise other runtime errors

### Vertex AI Search[¶](https://google.github.io/adk-docs/tools/built-in-tools/#vertex-ai-search)

The vertex\_ai\_search\_tool uses Google Cloud's Vertex AI Search, enabling the agent to search across your private, configured data stores (e.g., internal documents, company policies, knowledge bases). This built-in tool requires you to provide the specific data store ID during configuration.

import asyncio

from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

from google.adk.tools import VertexAiSearchTool

# Replace with your actual Vertex AI Search Datastore ID

# Format: projects/<PROJECT\_ID>/locations/<LOCATION>/collections/default\_collection/dataStores/<DATASTORE\_ID>

# e.g., "projects/12345/locations/us-central1/collections/default\_collection/dataStores/my-datastore-123"

YOUR\_DATASTORE\_ID = "YOUR\_DATASTORE\_ID\_HERE"

# Constants

APP\_NAME\_VSEARCH = "vertex\_search\_app"

USER\_ID\_VSEARCH = "user\_vsearch\_1"

SESSION\_ID\_VSEARCH = "session\_vsearch\_1"

AGENT\_NAME\_VSEARCH = "doc\_qa\_agent"

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

# Tool Instantiation

# You MUST provide your datastore ID here.

vertex\_search\_tool = VertexAiSearchTool(data\_store\_id=YOUR\_DATASTORE\_ID)

# Agent Definition

doc\_qa\_agent = LlmAgent(

name=AGENT\_NAME\_VSEARCH,

model=GEMINI\_2\_FLASH, # Requires Gemini model

tools=[vertex\_search\_tool],

instruction=f"""You are a helpful assistant that answers questions based on information found in the document store: {YOUR\_DATASTORE\_ID}.

Use the search tool to find relevant information before answering.

If the answer isn't in the documents, say that you couldn't find the information.

""",

description="Answers questions using a specific Vertex AI Search datastore.",

)

# Session and Runner Setup

session\_service\_vsearch = InMemorySessionService()

runner\_vsearch = Runner(

agent=doc\_qa\_agent, app\_name=APP\_NAME\_VSEARCH, session\_service=session\_service\_vsearch

)

session\_vsearch = session\_service\_vsearch.create\_session(

app\_name=APP\_NAME\_VSEARCH, user\_id=USER\_ID\_VSEARCH, session\_id=SESSION\_ID\_VSEARCH

)

# Agent Interaction Function

async def call\_vsearch\_agent\_async(query):

print("\n--- Running Vertex AI Search Agent ---")

print(f"Query: {query}")

if "YOUR\_DATASTORE\_ID\_HERE" in YOUR\_DATASTORE\_ID:

print("Skipping execution: Please replace YOUR\_DATASTORE\_ID\_HERE with your actual datastore ID.")

print("-" \* 30)

return

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

final\_response\_text = "No response received."

try:

async for event in runner\_vsearch.run\_async(

user\_id=USER\_ID\_VSEARCH, session\_id=SESSION\_ID\_VSEARCH, new\_message=content

):

# Like Google Search, results are often embedded in the model's response.

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

final\_response\_text = event.content.parts[0].text.strip()

print(f"Agent Response: {final\_response\_text}")

# You can inspect event.grounding\_metadata for source citations

if event.grounding\_metadata:

print(f" (Grounding metadata found with {len(event.grounding\_metadata.grounding\_attributions)} attributions)")

except Exception as e:

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

print("Ensure your datastore ID is correct and the service account has permissions.")

print("-" \* 30)

# --- Run Example ---

async def run\_vsearch\_example():

# Replace with a question relevant to YOUR datastore content

await call\_vsearch\_agent\_async("Summarize the main points about the Q2 strategy document.")

await call\_vsearch\_agent\_async("What safety procedures are mentioned for lab X?")

# Execute the example

# await run\_vsearch\_example()

# Running locally due to potential colab asyncio issues with multiple awaits

try:

asyncio.run(run\_vsearch\_example())

except RuntimeError as e:

if "cannot be called from a running event loop" in str(e):

print("Skipping execution in running event loop (like Colab/Jupyter). Run locally.")

else:

raise e

## Use Built-in tools with other tools[¶](https://google.github.io/adk-docs/tools/built-in-tools/#use-built-in-tools-with-other-tools)

The following code sample demonstrates how to use multiple built-in tools or how to use built-in tools with other tools by using multiple agents:

**[Python](https://google.github.io/adk-docs/tools/built-in-tools/#python_2)**

**[Java](https://google.github.io/adk-docs/tools/built-in-tools/#java_2)**

from google.adk.tools import agent\_tool

from google.adk.agents import Agent

from google.adk.tools import google\_search

from google.adk.code\_executors import BuiltInCodeExecutor

search\_agent = Agent(

model='gemini-2.0-flash',

name='SearchAgent',

instruction="""

You're a specialist in Google Search

""",

tools=[google\_search],

)

coding\_agent = Agent(

model='gemini-2.0-flash',

name='CodeAgent',

instruction="""

You're a specialist in Code Execution

""",

code\_executor=[BuiltInCodeExecutor],

)

root\_agent = Agent(

name="RootAgent",

model="gemini-2.0-flash",

description="Root Agent",

tools=[agent\_tool.AgentTool(agent=search\_agent), agent\_tool.AgentTool(agent=coding\_agent)],

)

### Limitations[¶](https://google.github.io/adk-docs/tools/built-in-tools/#limitations)

**Warning**

Currently, for each root agent or single agent, only one built-in tool is supported. No other tools of any type can be used in the same agent.

For example, the following approach that uses ***a built-in tool along with other tools*** within a single agent is **not** currently supported:

**[Python](https://google.github.io/adk-docs/tools/built-in-tools/#python_3)**

**[Java](https://google.github.io/adk-docs/tools/built-in-tools/#java_3)**

root\_agent = Agent(

name="RootAgent",

model="gemini-2.0-flash",

description="Root Agent",

tools=[custom\_function],

executor=[BuiltInCodeExecutor] # <-- not supported when used with tools

)

**Warning**

Built-in tools cannot be used within a sub-agent.

For example, the following approach that uses built-in tools within sub-agents is **not** currently supported:

**[Python](https://google.github.io/adk-docs/tools/built-in-tools/#python_4)**

**[Java](https://google.github.io/adk-docs/tools/built-in-tools/#java_4)**

search\_agent = Agent(

model='gemini-2.0-flash',

name='SearchAgent',

instruction="""

You're a specialist in Google Search

""",

tools=[google\_search],

)

coding\_agent = Agent(

model='gemini-2.0-flash',

name='CodeAgent',

instruction="""

You're a specialist in Code Execution

""",

executor=[BuiltInCodeExecutor],

)

root\_agent = Agent(

name="RootAgent",

model="gemini-2.0-flash",

description="Root Agent",

sub\_agents=[

search\_agent,

coding\_agent

],

)

# Third Party Tools[¶](https://google.github.io/adk-docs/tools/third-party-tools/#third-party-tools)

python_only

ADK is designed to be **highly extensible, allowing you to seamlessly integrate tools from other AI Agent frameworks** like CrewAI and LangChain. This interoperability is crucial because it allows for faster development time and allows you to reuse existing tools.

## 1. Using LangChain Tools[¶](https://google.github.io/adk-docs/tools/third-party-tools/#1-using-langchain-tools)

ADK provides the LangchainTool wrapper to integrate tools from the LangChain ecosystem into your agents.

### Example: Web Search using LangChain's Tavily tool[¶](https://google.github.io/adk-docs/tools/third-party-tools/#example-web-search-using-langchains-tavily-tool)

[Tavily](https://tavily.com/) provides a search API that returns answers derived from real-time search results, intended for use by applications like AI agents.

1. Follow [ADK installation and setup](https://google.github.io/adk-docs/get-started/installation/) guide.
2. **Install Dependencies:** Ensure you have the necessary LangChain packages installed. For example, to use the Tavily search tool, install its specific dependencies:

pip install langchain\_community tavily-python

1. Obtain a [Tavily](https://tavily.com/) API KEY and export it as an environment variable.

export TAVILY\_API\_KEY=<REPLACE\_WITH\_API\_KEY>

1. **Import:** Import the LangchainTool wrapper from ADK and the specific LangChain tool you wish to use (e.g, TavilySearchResults).

from google.adk.tools.langchain\_tool import LangchainTool

from langchain\_community.tools import TavilySearchResults

1. **Instantiate & Wrap:** Create an instance of your LangChain tool and pass it to the LangchainTool constructor.

# Instantiate the LangChain tool

tavily\_tool\_instance = TavilySearchResults(

max\_results=5,

search\_depth="advanced",

include\_answer=True,

include\_raw\_content=True,

include\_images=True,

)

# Wrap it with LangchainTool for ADK

adk\_tavily\_tool = LangchainTool(tool=tavily\_tool\_instance)

1. **Add to Agent:** Include the wrapped LangchainTool instance in your agent's tools list during definition.

from google.adk import Agent

# Define the ADK agent, including the wrapped tool

my\_agent = Agent(

name="langchain\_tool\_agent",

model="gemini-2.0-flash",

description="Agent to answer questions using TavilySearch.",

instruction="I can answer your questions by searching the internet. Just ask me anything!",

tools=[adk\_tavily\_tool] # Add the wrapped tool here

)

### Full Example: Tavily Search[¶](https://google.github.io/adk-docs/tools/third-party-tools/#full-example-tavily-search)

Here's the full code combining the steps above to create and run an agent using the LangChain Tavily search tool.

import os

from google.adk import Agent, Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools.langchain\_tool import LangchainTool

from google.genai import types

from langchain\_community.tools import TavilySearchResults

# Ensure TAVILY\_API\_KEY is set in your environment

if not os.getenv("TAVILY\_API\_KEY"):

print("Warning: TAVILY\_API\_KEY environment variable not set.")

APP\_NAME = "news\_app"

USER\_ID = "1234"

SESSION\_ID = "session1234"

# Instantiate LangChain tool

tavily\_search = TavilySearchResults(

max\_results=5,

search\_depth="advanced",

include\_answer=True,

include\_raw\_content=True,

include\_images=True,

)

# Wrap with LangchainTool

adk\_tavily\_tool = LangchainTool(tool=tavily\_search)

# Define Agent with the wrapped tool

my\_agent = Agent(

name="langchain\_tool\_agent",

model="gemini-2.0-flash",

description="Agent to answer questions using TavilySearch.",

instruction="I can answer your questions by searching the internet. Just ask me anything!",

tools=[adk\_tavily\_tool] # Add the wrapped tool here

)

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=my\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

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

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

for event in events:

if event.is\_final\_response():

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

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

call\_agent("stock price of GOOG")

## 2. Using CrewAI tools[¶](https://google.github.io/adk-docs/tools/third-party-tools/#2-using-crewai-tools)

ADK provides the CrewaiTool wrapper to integrate tools from the CrewAI library.

### Example: Web Search using CrewAI's Serper API[¶](https://google.github.io/adk-docs/tools/third-party-tools/#example-web-search-using-crewais-serper-api)

[Serper API](https://serper.dev/) provides access to Google Search results programmatically. It allows applications, like AI agents, to perform real-time Google searches (including news, images, etc.) and get structured data back without needing to scrape web pages directly.

1. Follow [ADK installation and setup](https://google.github.io/adk-docs/get-started/installation/) guide.
2. **Install Dependencies:** Install the necessary CrewAI tools package. For example, to use the SerperDevTool:

pip install crewai-tools

1. Obtain a [Serper API KEY](https://serper.dev/) and export it as an environment variable.

export SERPER\_API\_KEY=<REPLACE\_WITH\_API\_KEY>

1. **Import:** Import CrewaiTool from ADK and the desired CrewAI tool (e.g, SerperDevTool).

from google.adk.tools.crewai\_tool import CrewaiTool

from crewai\_tools import SerperDevTool

1. **Instantiate & Wrap:** Create an instance of the CrewAI tool. Pass it to the CrewaiTool constructor. **Crucially, you must provide a name and description** to the ADK wrapper, as these are used by ADK's underlying model to understand when to use the tool.

# Instantiate the CrewAI tool

serper\_tool\_instance = SerperDevTool(

n\_results=10,

save\_file=False,

search\_type="news",

)

# Wrap it with CrewaiTool for ADK, providing name and description

adk\_serper\_tool = CrewaiTool(

name="InternetNewsSearch",

description="Searches the internet specifically for recent news articles using Serper.",

tool=serper\_tool\_instance

)

1. **Add to Agent:** Include the wrapped CrewaiTool instance in your agent's tools list.

from google.adk import Agent

# Define the ADK agent

my\_agent = Agent(

name="crewai\_search\_agent",

model="gemini-2.0-flash",

description="Agent to find recent news using the Serper search tool.",

instruction="I can find the latest news for you. What topic are you interested in?",

tools=[adk\_serper\_tool] # Add the wrapped tool here

)

### Full Example: Serper API[¶](https://google.github.io/adk-docs/tools/third-party-tools/#full-example-serper-api)

Here's the full code combining the steps above to create and run an agent using the CrewAI Serper API search tool.

import os

from google.adk import Agent, Runner

from google.adk.sessions import InMemorySessionService

from google.adk.tools.crewai\_tool import CrewaiTool

from google.genai import types

from crewai\_tools import SerperDevTool

# Constants

APP\_NAME = "news\_app"

USER\_ID = "user1234"

SESSION\_ID = "1234"

# Ensure SERPER\_API\_KEY is set in your environment

if not os.getenv("SERPER\_API\_KEY"):

print("Warning: SERPER\_API\_KEY environment variable not set.")

serper\_tool\_instance = SerperDevTool(

n\_results=10,

save\_file=False,

search\_type="news",

)

adk\_serper\_tool = CrewaiTool(

name="InternetNewsSearch",

description="Searches the internet specifically for recent news articles using Serper.",

tool=serper\_tool\_instance

)

serper\_agent = Agent(

name="basic\_search\_agent",

model="gemini-2.0-flash",

description="Agent to answer questions using Google Search.",

instruction="I can answer your questions by searching the internet. Just ask me anything!",

# Add the Serper tool

tools=[adk\_serper\_tool]

)

# Session and Runner

session\_service = InMemorySessionService()

session = session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)

runner = Runner(agent=serper\_agent, app\_name=APP\_NAME, session\_service=session\_service)

# Agent Interaction

def call\_agent(query):

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

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

for event in events:

if event.is\_final\_response():

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

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

call\_agent("what's the latest news on AI Agents?")

# Google Cloud Tools[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#google-cloud-tools)

python_only

Google Cloud tools make it easier to connect your agents to Google Cloud’s products and services. With just a few lines of code you can use these tools to connect your agents with:

* **Any custom APIs** that developers host in Apigee.
* **100s** of **prebuilt connectors** to enterprise systems such as Salesforce, Workday, and SAP.
* **Automation workflows** built using application integration.
* **Databases** such as Spanner, AlloyDB, Postgres and more using the MCP Toolbox for databases.

Google Cloud Tools

## Apigee API Hub Tools[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#apigee-api-hub-tools)

**ApiHubToolset** lets you turn any documented API from Apigee API hub into a tool with a few lines of code. This section shows you the step by step instructions including setting up authentication for a secure connection to your APIs.

**Prerequisites**

1. [Install ADK](https://google.github.io/adk-docs/get-started/installation/)
2. Install the [Google Cloud CLI](https://cloud.google.com/sdk/docs/install?db=bigtable-docs#installation_instructions).
3. [Apigee API hub](https://cloud.google.com/apigee/docs/apihub/what-is-api-hub) instance with documented (i.e. OpenAPI spec) APIs
4. Set up your project structure and create required files

project\_root\_folder

|

`-- my\_agent

|-- .env

|-- \_\_init\_\_.py

|-- agent.py

`\_\_ tool.py

### Create an API Hub Toolset[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#create-an-api-hub-toolset)

Note: This tutorial includes an agent creation. If you already have an agent, you only need to follow a subset of these steps.

1. Get your access token, so that APIHubToolset can fetch spec from API Hub API. In your terminal run the following command

gcloud auth print-access-token

# Prints your access token like 'ya29....'

1. Ensure that the account used has the required permissions. You can use the pre-defined role roles/apihub.viewer or assign the following permissions:
   1. **apihub.specs.get (required)**
   2. apihub.apis.get (optional)
   3. apihub.apis.list (optional)
   4. apihub.versions.get (optional)
   5. apihub.versions.list (optional)
   6. apihub.specs.list (optional)
2. Create a tool with APIHubToolset. Add the below to tools.py  
   If your API requires authentication, you must configure authentication for the tool. The following code sample demonstrates how to configure an API key. ADK supports token based auth (API Key, Bearer token), service account, and OpenID Connect. We will soon add support for various OAuth2 flows.

from google.adk.tools.openapi\_tool.auth.auth\_helpers import token\_to\_scheme\_credential

from google.adk.tools.apihub\_tool.apihub\_toolset import APIHubToolset

# Provide authentication for your APIs. Not required if your APIs don't required authentication.

auth\_scheme, auth\_credential = token\_to\_scheme\_credential(

"apikey", "query", "apikey", apikey\_credential\_str

)

sample\_toolset\_with\_auth = APIHubToolset(

name="apihub-sample-tool",

description="Sample Tool",

access\_token="...", # Copy your access token generated in step 1

apihub\_resource\_name="...", # API Hub resource name

auth\_scheme=auth\_scheme,

auth\_credential=auth\_credential,

)

1. For production deployment we recommend using a service account instead of an access token. In the code snippet above, use service\_account\_json=service\_account\_cred\_json\_str and provide your security account credentials instead of the token.  
   For apihub\_resource\_name, if you know the specific ID of the OpenAPI Spec being used for your API, use `projects/my-project-id/locations/us-west1/apis/my-api-id/versions/version-id/specs/spec-id`. If you would like the Toolset to automatically pull the first available spec from the API, use `projects/my-project-id/locations/us-west1/apis/my-api-id`
2. Create your agent file [Agent.py](http://agent.py/) and add the created tools to your agent definition:

from google.adk.agents.llm\_agent import LlmAgent

from .tools import sample\_toolset

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='enterprise\_assistant',

instruction='Help user, leverage the tools you have access to',

tools=sample\_toolset.get\_tools(),

)

1. Configure your \_\_init\_\_.py to expose your agent

from . import agent

1. Start the Google ADK Web UI and try your agent:

# make sure to run `adk web` from your project\_root\_folder

adk web

Then go to [http://localhost:8000](http://localhost:8000/) to try your agent from the Web UI.

## Application Integration Tools[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#application-integration-tools)

With **ApplicationIntegrationToolset** you can seamlessly give your agents a secure and governed to enterprise applications using Integration Connector’s 100+ pre-built connectors for systems like Salesforce, ServiceNow, JIRA, SAP, and more. Support for both on-prem and SaaS applications. In addition you can turn your existing Application Integration process automations into agentic workflows by providing application integration workflows as tools to your ADK agents.

**Prerequisites**

1. [Install ADK](https://google.github.io/adk-docs/get-started/installation/)
2. An existing [Application Integration](https://cloud.google.com/application-integration/docs/overview) workflow or [Integrations Connector](https://cloud.google.com/integration-connectors/docs/overview) connection you want to use with your agent
3. To use tool with default credentials: have Google Cloud CLI installed. See [installation guide](https://cloud.google.com/sdk/docs/install#installation_instructions)*.*

*Run:*

gcloud config set project <project-id>

gcloud auth application-default login

gcloud auth application-default set-quota-project <project-id>

1. Set up your project structure and create required files

project\_root\_folder

|-- .env

`-- my\_agent

|-- \_\_init\_\_.py

|-- agent.py

`\_\_ tools.py

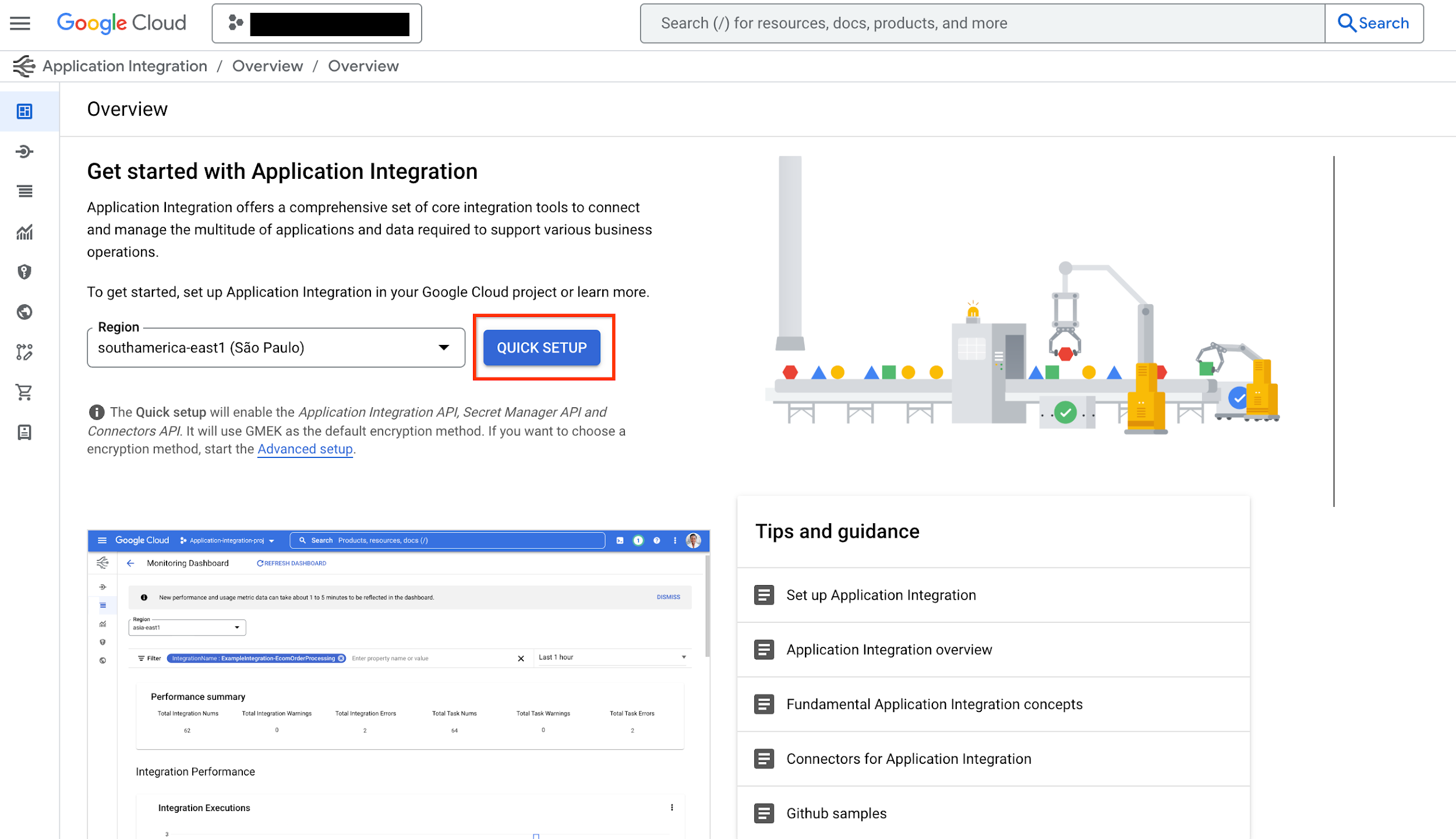
When running the agent, make sure to run adk web in project\_root\_folder

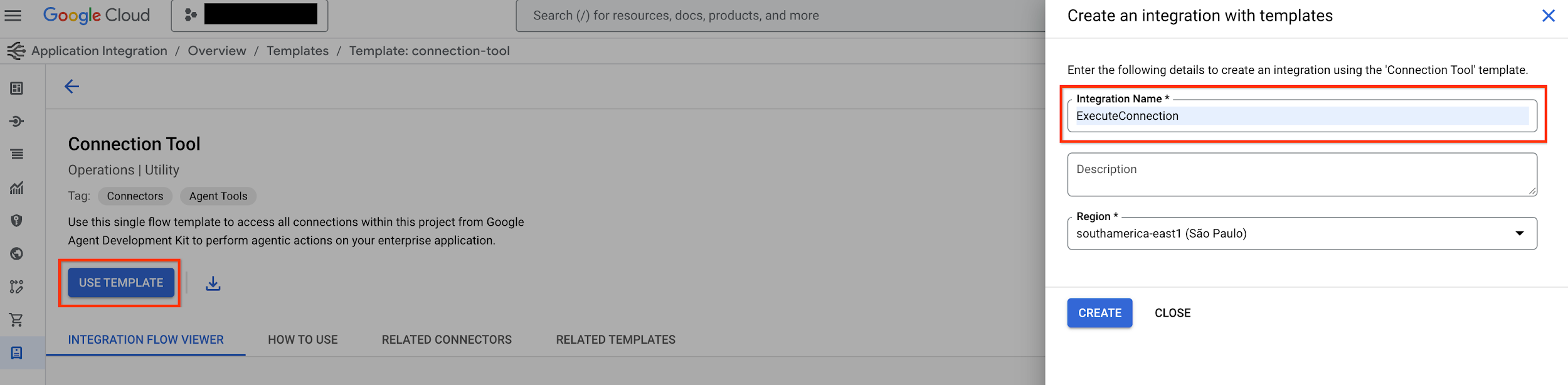
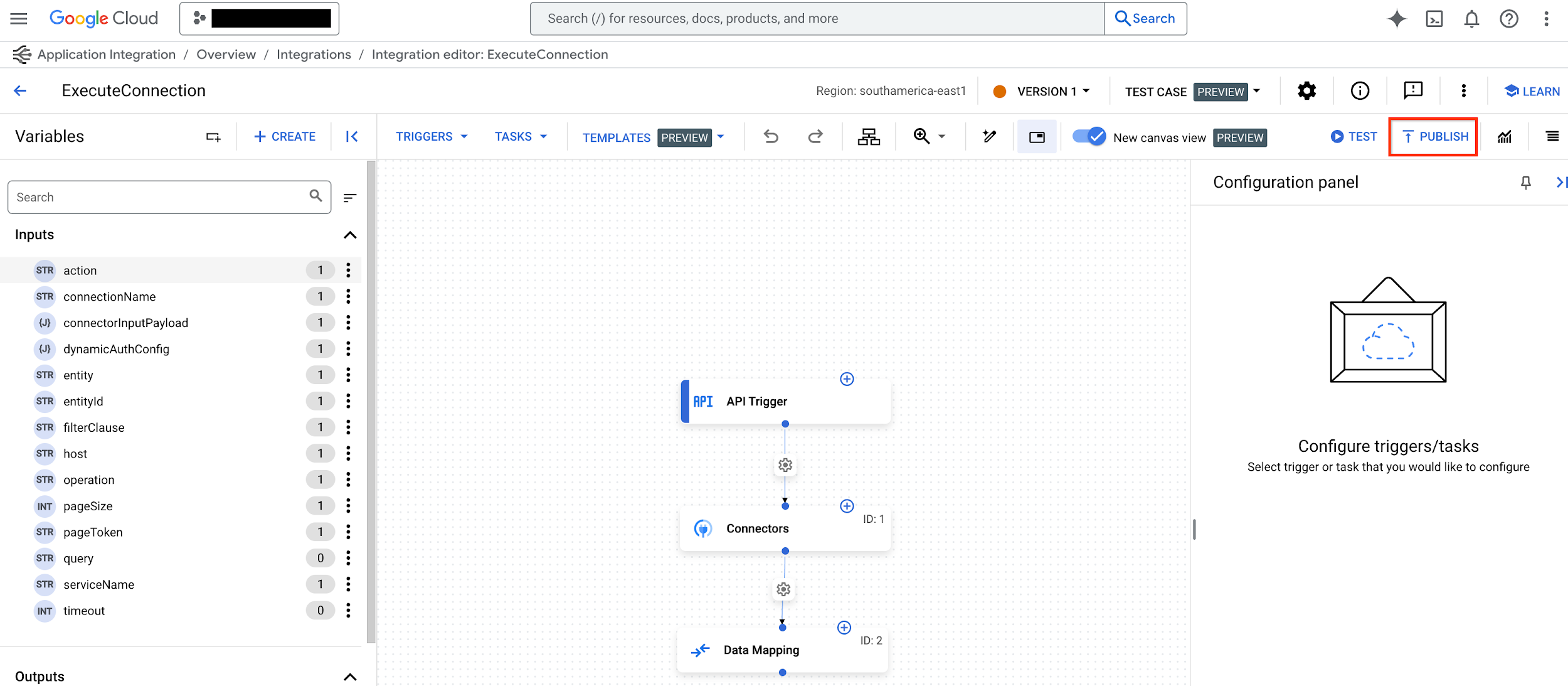
### Use Integration Connectors[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#use-integration-connectors)

Connect your agent to enterprise applications using [Integration Connectors](https://cloud.google.com/integration-connectors/docs/overview).

**Prerequisites**

1. To use a connector from Integration Connectors, you need to [provision](https://console.cloud.google.com/integrations) Application Integration in the same region as your connection by clicking on "QUICK SETUP" button.



1. Go to [Connection Tool](https://console.cloud.google.com/integrations/templates/connection-tool/locations/us-central1) template from the template library and click on "USE TEMPLATE" button.  
   
2. Fill the Integration Name as **ExecuteConnection** (It is mandatory to use this integration name only) and select the region same as the connection region. Click on "CREATE".
3. Publish the integration by using the "PUBLISH" button on the Application Integration Editor.  
   

**Steps:**

1. Create a tool with ApplicationIntegrationToolset within your tools.py file

from google.adk.tools.application\_integration\_tool.application\_integration\_toolset import ApplicationIntegrationToolset

connector\_tool = ApplicationIntegrationToolset(

project="test-project", # TODO: replace with GCP project of the connection

location="us-central1", #TODO: replace with location of the connection

connection="test-connection", #TODO: replace with connection name

entity\_operations={"Entity\_One": ["LIST","CREATE"], "Entity\_Two": []},#empty list for actions means all operations on the entity are supported.

actions=["action1"], #TODO: replace with actions

service\_account\_credentials='{...}', # optional

tool\_name="tool\_prefix2",

tool\_instructions="..."

)

1. Note: - You can provide service account to be used instead of using default credentials. - To find the list of supported entities and actions for a connection, use the connectors apis: [listActions](https://cloud.google.com/integration-connectors/docs/reference/rest/v1/projects.locations.connections.connectionSchemaMetadata/listActions) or [listEntityTypes](https://cloud.google.com/integration-connectors/docs/reference/rest/v1/projects.locations.connections.connectionSchemaMetadata/listEntityTypes)
2. Add the tool to your agent. Update your agent.py file

from google.adk.agents.llm\_agent import LlmAgent

from .tools import connector\_tool

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='connector\_agent',

instruction="Help user, leverage the tools you have access to",

tools=connector\_tool.get\_tools(),

)

1. Configure your \_\_init\_\_.py to expose your agent

from . import agent

1. Start the Google ADK Web UI and try your agent.

# make sure to run `adk web` from your project\_root\_folder

adk web

Then go to [http://localhost:8000](http://localhost:8000/), and choose my\_agent agent (same as the agent folder name)

### Use App Integration Workflows[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#use-app-integration-workflows)

Use existing [Application Integration](https://cloud.google.com/application-integration/docs/overview) workflow as a tool for your agent or create a new one.

**Steps:**

1. Create a tool with ApplicationIntegrationToolset within your tools.py file

integration\_tool = ApplicationIntegrationToolset(

project="test-project", # TODO: replace with GCP project of the connection

location="us-central1", #TODO: replace with location of the connection

integration="test-integration", #TODO: replace with integration name

trigger="api\_trigger/test\_trigger",#TODO: replace with trigger id

service\_account\_credentials='{...}', #optional

tool\_name="tool\_prefix1",

tool\_instructions="..."

)

1. Note: You can provide service account to be used instead of using default credentials
2. Add the tool to your agent. Update your agent.py file

from google.adk.agents.llm\_agent import LlmAgent

from .tools import integration\_tool, connector\_tool

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='integration\_agent',

instruction="Help user, leverage the tools you have access to",

tools=integration\_tool.get\_tools(),

)

1. Configure your `\_\_init\_\_.py` to expose your agent

from . import agent

1. Start the Google ADK Web UI and try your agent.

# make sure to run `adk web` from your project\_root\_folder

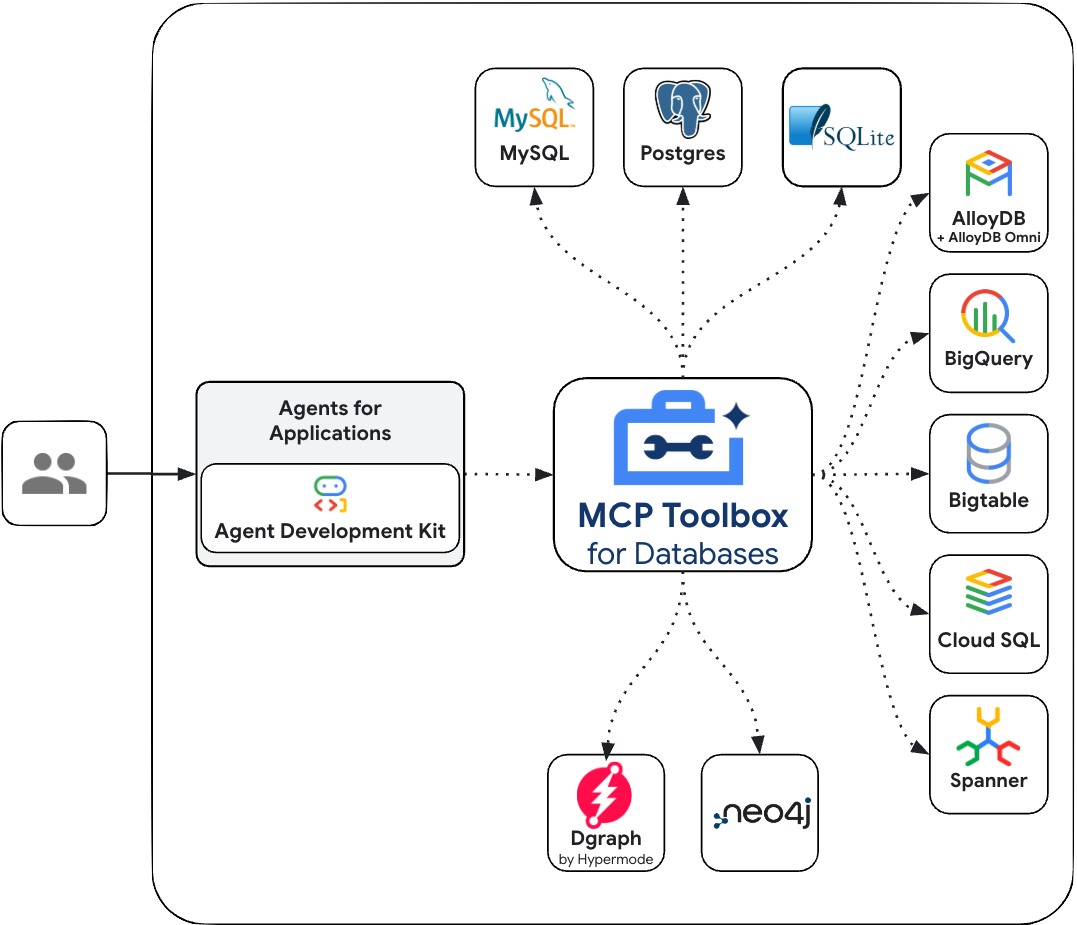
adk web

1. Then go to [http://localhost:8000](http://localhost:8000/), and choose my\_agent agent (same as the agent folder name)

## Toolbox Tools for Databases[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#toolbox-tools-for-databases)

[MCP Toolbox for Databases](https://github.com/googleapis/genai-toolbox) is an open source MCP server for databases. It was designed with enterprise-grade and production-quality in mind. It enables you to develop tools easier, faster, and more securely by handling the complexities such as connection pooling, authentication, and more.

Google’s Agent Development Kit (ADK) has built in support for Toolbox. For more information on [getting started](https://googleapis.github.io/genai-toolbox/getting-started) or [configuring](https://googleapis.github.io/genai-toolbox/getting-started/configure/) Toolbox, see the [documentation](https://googleapis.github.io/genai-toolbox/getting-started/introduction/).



### Configure and deploy[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#configure-and-deploy)

Toolbox is an open source server that you deploy and manage yourself. For more instructions on deploying and configuring, see the official Toolbox documentation:

* [Installing the Server](https://googleapis.github.io/genai-toolbox/getting-started/introduction/#installing-the-server)
* [Configuring Toolbox](https://googleapis.github.io/genai-toolbox/getting-started/configure/)

### Install client SDK[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#install-client-sdk)

ADK relies on the toolbox-core python package to use Toolbox. Install the package before getting started:

pip install toolbox-core

### Loading Toolbox Tools[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#loading-toolbox-tools)

Once you’re Toolbox server is configured and up and running, you can load tools from your server using ADK:

from google.adk.agents import Agent

from toolbox\_core import ToolboxSyncClient

toolbox = ToolboxSyncClient("https://127.0.0.1:5000")

# Load a specific set of tools

tools = toolbox.load\_toolset('my-toolset-name'),

# Load single tool

tools = toolbox.load\_tool('my-tool-name'),

root\_agent = Agent(

...,

tools=tools # Provide the list of tools to the Agent

)

### Advanced Toolbox Features[¶](https://google.github.io/adk-docs/tools/google-cloud-tools/#advanced-toolbox-features)

Toolbox has a variety of features to make developing Gen AI tools for databases. For more information, read more about the following features:

* [Authenticated Parameters](https://googleapis.github.io/genai-toolbox/resources/tools/#authenticated-parameters): bind tool inputs to values from OIDC tokens automatically, making it easy to run sensitive queries without potentially leaking data
* [Authorized Invocations:](https://googleapis.github.io/genai-toolbox/resources/tools/#authorized-invocations) restrict access to use a tool based on the users Auth token
* [OpenTelemetry](https://googleapis.github.io/genai-toolbox/how-to/export_telemetry/): get metrics and tracing from Toolbox with OpenTelemetry

# Model Context Protocol Tools[¶](https://google.github.io/adk-docs/tools/mcp-tools/#model-context-protocol-tools)

This guide walks you through two ways of integrating Model Context Protocol (MCP) with ADK.

## What is Model Context Protocol (MCP)?[¶](https://google.github.io/adk-docs/tools/mcp-tools/#what-is-model-context-protocol-mcp)

The Model Context Protocol (MCP) is an open standard designed to standardize how Large Language Models (LLMs) like Gemini and Claude communicate with external applications, data sources, and tools. Think of it as a universal connection mechanism that simplifies how LLMs obtain context, execute actions, and interact with various systems.

MCP follows a client-server architecture, defining how **data** (resources), **interactive templates** (prompts), and **actionable functions** (tools) are exposed by an **MCP server** and consumed by an **MCP client** (which could be an LLM host application or an AI agent).

This guide covers two primary integration patterns:

1. **Using Existing MCP Servers within ADK:** An ADK agent acts as an MCP client, leveraging tools provided by external MCP servers.
2. **Exposing ADK Tools via an MCP Server:** Building an MCP server that wraps ADK tools, making them accessible to any MCP client.

## Prerequisites[¶](https://google.github.io/adk-docs/tools/mcp-tools/#prerequisites)

Before you begin, ensure you have the following set up:

* **Set up ADK:** Follow the standard ADK [setup instructions](https://google.github.io/adk-docs/get-started/quickstart.md) in the quickstart.
* **Install/update Python/Java:** MCP requires Python version of 3.9 or higher for Python or Java 17+.
* **Setup Node.js and npx:** **(Python only)** Many community MCP servers are distributed as Node.js packages and run using npx. Install Node.js (which includes npx) if you haven't already. For details, see <https://nodejs.org/en>.
* **Verify Installations:** **(Python only)** Confirm adk and npx are in your PATH within the activated virtual environment:

# Both commands should print the path to the executables.

which adk

which npx

## 1. Using MCP servers with ADK agents (ADK as an MCP client) in adk web[¶](https://google.github.io/adk-docs/tools/mcp-tools/#1-using-mcp-servers-with-adk-agents-adk-as-an-mcp-client-in-adk-web)

This section demonstrates how to integrate tools from external MCP (Model Context Protocol) servers into your ADK agents. This is the **most common** integration pattern when your ADK agent needs to use capabilities provided by an existing service that exposes an MCP interface. You will see how the MCPToolset class can be directly added to your agent's tools list, enabling seamless connection to an MCP server, discovery of its tools, and making them available for your agent to use. These examples primarily focus on interactions within the adk web development environment.

### MCPToolset class[¶](https://google.github.io/adk-docs/tools/mcp-tools/#mcptoolset-class)

The MCPToolset class is ADK's primary mechanism for integrating tools from an MCP server. When you include an MCPToolset instance in your agent's tools list, it automatically handles the interaction with the specified MCP server. Here's how it works:

1. **Connection Management:** On initialization, MCPToolset establishes and manages the connection to the MCP server. This can be a local server process (using StdioServerParameters for communication over standard input/output) or a remote server (using SseServerParams for Server-Sent Events). The toolset also handles the graceful shutdown of this connection when the agent or application terminates.
2. **Tool Discovery & Adaptation:** Once connected, MCPToolset queries the MCP server for its available tools (via the list\_tools MCP method). It then converts the schemas of these discovered MCP tools into ADK-compatible BaseTool instances.
3. **Exposure to Agent:** These adapted tools are then made available to your LlmAgent as if they were native ADK tools.
4. **Proxying Tool Calls:** When your LlmAgent decides to use one of these tools, MCPToolset transparently proxies the call (using the call\_tool MCP method) to the MCP server, sends the necessary arguments, and returns the server's response back to the agent.
5. **Filtering (Optional):** You can use the tool\_filter parameter when creating an MCPToolset to select a specific subset of tools from the MCP server, rather than exposing all of them to your agent.

The following examples demonstrate how to use MCPToolset within the adk web development environment. For scenarios where you need more fine-grained control over the MCP connection lifecycle or are not using adk web, refer to the "Using MCP Tools in your own Agent out of adk web" section later in this page.

### Example 1: File System MCP Server[¶](https://google.github.io/adk-docs/tools/mcp-tools/#example-1-file-system-mcp-server)

This example demonstrates connecting to a local MCP server that provides file system operations.

#### **Step 1: Define your Agent with MCPToolset**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-1-define-your-agent-with-mcptoolset)

Create an agent.py file (e.g., in ./adk\_agent\_samples/mcp\_agent/agent.py). The MCPToolset is instantiated directly within the tools list of your LlmAgent.

* **Important:** Replace "/path/to/your/folder" in the args list with the **absolute path** to an actual folder on your local system that the MCP server can access.

# ./adk\_agent\_samples/mcp\_agent/agent.py

import os # Required for path operations

from google.adk.agents import LlmAgent

from google.adk.tools.mcp\_tool.mcp\_toolset import MCPToolset, StdioServerParameters

# It's good practice to define paths dynamically if possible,

# or ensure the user understands the need for an ABSOLUTE path.

# For this example, we'll construct a path relative to this file,

# assuming '/path/to/your/folder' is in the same directory as agent.py.

# REPLACE THIS with an actual absolute path if needed for your setup.

TARGET\_FOLDER\_PATH = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), "/path/to/your/folder")

# Ensure TARGET\_FOLDER\_PATH is an absolute path for the MCP server.

# If you created ./adk\_agent\_samples/mcp\_agent/your\_folder,

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='filesystem\_assistant\_agent',

instruction='Help the user manage their files. You can list files, read files, etc.',

tools=[

MCPToolset(

connection\_params=StdioServerParameters(

command='npx',

args=[

"-y", # Argument for npx to auto-confirm install

"@modelcontextprotocol/server-filesystem",

# IMPORTANT: This MUST be an ABSOLUTE path to a folder the

# npx process can access.

# Replace with a valid absolute path on your system.

# For example: "/Users/youruser/accessible\_mcp\_files"

# or use a dynamically constructed absolute path:

os.path.abspath(TARGET\_FOLDER\_PATH),

],

),

# Optional: Filter which tools from the MCP server are exposed

# tool\_filter=['list\_directory', 'read\_file']

)

],

)

#### **Step 2: Create an \_\_init\_\_.py file**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-2-create-an-__init__py-file)

Ensure you have an \_\_init\_\_.py in the same directory as agent.py to make it a discoverable Python package for ADK.

# ./adk\_agent\_samples/mcp\_agent/\_\_init\_\_.py

from . import agent

#### **Step 3: Run adk web and Interact**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-3-run-adk-web-and-interact)

Navigate to the parent directory of mcp\_agent (e.g., adk\_agent\_samples) in your terminal and run:

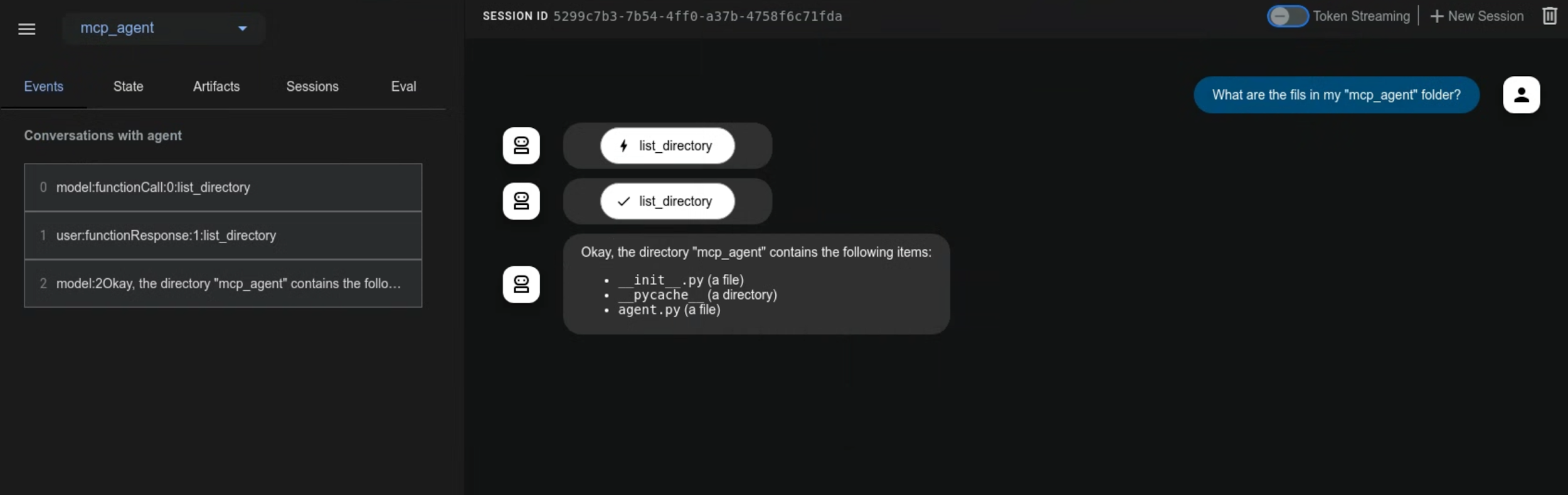
cd ./adk\_agent\_samples # Or your equivalent parent directory

adk web

Once the ADK Web UI loads in your browser:

1. Select the filesystem\_assistant\_agent from the agent dropdown.
2. Try prompts like:
   * "List files in the current directory."
   * "Can you read the file named sample.txt?" (assuming you created it in TARGET\_FOLDER\_PATH).
   * "What is the content of another\_file.md?"

You should see the agent interacting with the MCP file system server, and the server's responses (file listings, file content) relayed through the agent. The adk web console (terminal where you ran the command) might also show logs from the npx process if it outputs to stderr.



### Example 2: Google Maps MCP Server[¶](https://google.github.io/adk-docs/tools/mcp-tools/#example-2-google-maps-mcp-server)

This example demonstrates connecting to the Google Maps MCP server.

#### **Step 1: Get API Key and Enable APIs**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-1-get-api-key-and-enable-apis)

1. **Google Maps API Key:** Follow the directions at [Use API keys](https://developers.google.com/maps/documentation/javascript/get-api-key#create-api-keys) to obtain a Google Maps API Key.
2. **Enable APIs:** In your Google Cloud project, ensure the following APIs are enabled:
   * Directions API
   * Routes API For instructions, see the [Getting started with Google Maps Platform](https://developers.google.com/maps/get-started#enable-api-sdk) documentation.

#### **Step 2: Define your Agent with MCPToolset for Google Maps**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-2-define-your-agent-with-mcptoolset-for-google-maps)

Modify your agent.py file (e.g., in ./adk\_agent\_samples/mcp\_agent/agent.py). Replace YOUR\_GOOGLE\_MAPS\_API\_KEY with the actual API key you obtained.

# ./adk\_agent\_samples/mcp\_agent/agent.py

import os

from google.adk.agents import LlmAgent

from google.adk.tools.mcp\_tool.mcp\_toolset import MCPToolset, StdioServerParameters

# Retrieve the API key from an environment variable or directly insert it.

# Using an environment variable is generally safer.

# Ensure this environment variable is set in the terminal where you run 'adk web'.

# Example: export GOOGLE\_MAPS\_API\_KEY="YOUR\_ACTUAL\_KEY"

google\_maps\_api\_key = os.environ.get("GOOGLE\_MAPS\_API\_KEY")

if not google\_maps\_api\_key:

# Fallback or direct assignment for testing - NOT RECOMMENDED FOR PRODUCTION

google\_maps\_api\_key = "YOUR\_GOOGLE\_MAPS\_API\_KEY\_HERE" # Replace if not using env var

if google\_maps\_api\_key == "YOUR\_GOOGLE\_MAPS\_API\_KEY\_HERE":

print("WARNING: GOOGLE\_MAPS\_API\_KEY is not set. Please set it as an environment variable or in the script.")

# You might want to raise an error or exit if the key is crucial and not found.

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='maps\_assistant\_agent',

instruction='Help the user with mapping, directions, and finding places using Google Maps tools.',

tools=[

MCPToolset(

connection\_params=StdioServerParameters(

command='npx',

args=[

"-y",

"@modelcontextprotocol/server-google-maps",

],

# Pass the API key as an environment variable to the npx process

# This is how the MCP server for Google Maps expects the key.

env={

"GOOGLE\_MAPS\_API\_KEY": google\_maps\_api\_key

}

),

# You can filter for specific Maps tools if needed:

# tool\_filter=['get\_directions', 'find\_place\_by\_id']

)

],

)

#### **Step 3: Ensure \_\_init\_\_.py Exists**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-3-ensure-__init__py-exists)

If you created this in Example 1, you can skip this. Otherwise, ensure you have an \_\_init\_\_.py in the ./adk\_agent\_samples/mcp\_agent/ directory:

# ./adk\_agent\_samples/mcp\_agent/\_\_init\_\_.py

from . import agent

#### **Step 4: Run adk web and Interact**[**¶**](https://google.github.io/adk-docs/tools/mcp-tools/#step-4-run-adk-web-and-interact)

1. **Set Environment Variable (Recommended):** Before running adk web, it's best to set your Google Maps API key as an environment variable in your terminal:

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

1. Replace YOUR\_ACTUAL\_GOOGLE\_MAPS\_API\_KEY with your key.
2. **Run adk web**: Navigate to the parent directory of mcp\_agent (e.g., adk\_agent\_samples) and run:

cd ./adk\_agent\_samples # Or your equivalent parent directory

adk web

1. **Interact in the UI**:
   * Select the maps\_assistant\_agent.
   * Try prompts like:
     + "Get directions from GooglePlex to SFO."
     + "Find coffee shops near Golden Gate Park."
     + "What's the route from Paris, France to Berlin, Germany?"

You should see the agent use the Google Maps MCP tools to provide directions or location-based information.



## 2. Building an MCP server with ADK tools (MCP server exposing ADK)[¶](https://google.github.io/adk-docs/tools/mcp-tools/#2-building-an-mcp-server-with-adk-tools-mcp-server-exposing-adk)

This pattern allows you to wrap existing ADK tools and make them available to any standard MCP client application. The example in this section exposes the ADK load\_web\_page tool through a custom-built MCP server.

### Summary of steps[¶](https://google.github.io/adk-docs/tools/mcp-tools/#summary-of-steps)

You will create a standard Python MCP server application using the mcp library. Within this server, you will:

1. Instantiate the ADK tool(s) you want to expose (e.g., FunctionTool(load\_web\_page)).
2. Implement the MCP server's @app.list\_tools() handler to advertise the ADK tool(s). This involves converting the ADK tool definition to the MCP schema using the adk\_to\_mcp\_tool\_type utility from google.adk.tools.mcp\_tool.conversion\_utils.
3. Implement the MCP server's @app.call\_tool() handler. This handler will:
   * Receive tool call requests from MCP clients.
   * Identify if the request targets one of your wrapped ADK tools.
   * Execute the ADK tool's .run\_async() method.
   * Format the ADK tool's result into an MCP-compliant response (e.g., mcp.types.TextContent).

### Prerequisites[¶](https://google.github.io/adk-docs/tools/mcp-tools/#prerequisites_1)

Install the MCP server library in the same Python environment as your ADK installation:

pip install mcp

### Step 1: Create the MCP Server Script[¶](https://google.github.io/adk-docs/tools/mcp-tools/#step-1-create-the-mcp-server-script)

Create a new Python file for your MCP server, for example, my\_adk\_mcp\_server.py.

### Step 2: Implement the Server Logic[¶](https://google.github.io/adk-docs/tools/mcp-tools/#step-2-implement-the-server-logic)

Add the following code to my\_adk\_mcp\_server.py. This script sets up an MCP server that exposes the ADK load\_web\_page tool.

# my\_adk\_mcp\_server.py

import asyncio

import json

import os

from dotenv import load\_dotenv

# MCP Server Imports

from mcp import types as mcp\_types # Use alias to avoid conflict

from mcp.server.lowlevel import Server, NotificationOptions

from mcp.server.models import InitializationOptions

import mcp.server.stdio # For running as a stdio server

# ADK Tool Imports

from google.adk.tools.function\_tool import FunctionTool

from google.adk.tools.load\_web\_page import load\_web\_page # Example ADK tool

# ADK <-> MCP Conversion Utility

from google.adk.tools.mcp\_tool.conversion\_utils import adk\_to\_mcp\_tool\_type

# --- Load Environment Variables (If ADK tools need them, e.g., API keys) ---

load\_dotenv() # Create a .env file in the same directory if needed

# --- Prepare the ADK Tool ---

# Instantiate the ADK tool you want to expose.

# This tool will be wrapped and called by the MCP server.

print("Initializing ADK load\_web\_page tool...")

adk\_tool\_to\_expose = FunctionTool(load\_web\_page)

print(f"ADK tool '{adk\_tool\_to\_expose.name}' initialized and ready to be exposed via MCP.")

# --- End ADK Tool Prep ---

# --- MCP Server Setup ---

print("Creating MCP Server instance...")

# Create a named MCP Server instance using the mcp.server library

app = Server("adk-tool-exposing-mcp-server")

# Implement the MCP server's handler to list available tools

@app.list\_tools()

async def list\_mcp\_tools() -> list[mcp\_types.Tool]:

"""MCP handler to list tools this server exposes."""

print("MCP Server: Received list\_tools request.")

# Convert the ADK tool's definition to the MCP Tool schema format

mcp\_tool\_schema = adk\_to\_mcp\_tool\_type(adk\_tool\_to\_expose)

print(f"MCP Server: Advertising tool: {mcp\_tool\_schema.name}")

return [mcp\_tool\_schema]

# Implement the MCP server's handler to execute a tool call

@app.call\_tool()

async def call\_mcp\_tool(

name: str, arguments: dict

) -> list[mcp\_types.Content]: # MCP uses mcp\_types.Content

"""MCP handler to execute a tool call requested by an MCP client."""

print(f"MCP Server: Received call\_tool request for '{name}' with args: {arguments}")

# Check if the requested tool name matches our wrapped ADK tool

if name == adk\_tool\_to\_expose.name:

try:

# Execute the ADK tool's run\_async method.

# Note: tool\_context is None here because this MCP server is

# running the ADK tool outside of a full ADK Runner invocation.

# If the ADK tool requires ToolContext features (like state or auth),

# this direct invocation might need more sophisticated handling.

adk\_tool\_response = await adk\_tool\_to\_expose.run\_async(

args=arguments,

tool\_context=None,

)

print(f"MCP Server: ADK tool '{name}' executed. Response: {adk\_tool\_response}")

# Format the ADK tool's response (often a dict) into an MCP-compliant format.

# Here, we serialize the response dictionary as a JSON string within TextContent.

# Adjust formatting based on the ADK tool's output and client needs.

response\_text = json.dumps(adk\_tool\_response, indent=2)

# MCP expects a list of mcp\_types.Content parts

return [mcp\_types.TextContent(type="text", text=response\_text)]

except Exception as e:

print(f"MCP Server: Error executing ADK tool '{name}': {e}")

# Return an error message in MCP format

error\_text = json.dumps({"error": f"Failed to execute tool '{name}': {str(e)}"})

return [mcp\_types.TextContent(type="text", text=error\_text)]

else:

# Handle calls to unknown tools

print(f"MCP Server: Tool '{name}' not found/exposed by this server.")

error\_text = json.dumps({"error": f"Tool '{name}' not implemented by this server."})

return [mcp\_types.TextContent(type="text", text=error\_text)]

# --- MCP Server Runner ---

async def run\_mcp\_stdio\_server():

"""Runs the MCP server, listening for connections over standard input/output."""

# Use the stdio\_server context manager from the mcp.server.stdio library

async with mcp.server.stdio.stdio\_server() as (read\_stream, write\_stream):

print("MCP Stdio Server: Starting handshake with client...")

await app.run(

read\_stream,

write\_stream,

InitializationOptions(

server\_name=app.name, # Use the server name defined above

server\_version="0.1.0",

capabilities=app.get\_capabilities(

# Define server capabilities - consult MCP docs for options

notification\_options=NotificationOptions(),

experimental\_capabilities={},

),

),

)

print("MCP Stdio Server: Run loop finished or client disconnected.")

if \_\_name\_\_ == "\_\_main\_\_":

print("Launching MCP Server to expose ADK tools via stdio...")

try:

asyncio.run(run\_mcp\_stdio\_server())

except KeyboardInterrupt:

print("\nMCP Server (stdio) stopped by user.")

except Exception as e:

print(f"MCP Server (stdio) encountered an error: {e}")

finally:

print("MCP Server (stdio) process exiting.")

# --- End MCP Server ---

### Step 3: Test your Custom MCP Server with an ADK Agent[¶](https://google.github.io/adk-docs/tools/mcp-tools/#step-3-test-your-custom-mcp-server-with-an-adk-agent)

Now, create an ADK agent that will act as a client to the MCP server you just built. This ADK agent will use MCPToolset to connect to your my\_adk\_mcp\_server.py script.

Create an agent.py (e.g., in ./adk\_agent\_samples/mcp\_client\_agent/agent.py):

# ./adk\_agent\_samples/mcp\_client\_agent/agent.py

import os

from google.adk.agents import LlmAgent

from google.adk.tools.mcp\_tool import MCPToolset, StdioServerParameters

# IMPORTANT: Replace this with the ABSOLUTE path to your my\_adk\_mcp\_server.py script

PATH\_TO\_YOUR\_MCP\_SERVER\_SCRIPT = "/path/to/your/my\_adk\_mcp\_server.py" # <<< REPLACE

if PATH\_TO\_YOUR\_MCP\_SERVER\_SCRIPT == "/path/to/your/my\_adk\_mcp\_server.py":

print("WARNING: PATH\_TO\_YOUR\_MCP\_SERVER\_SCRIPT is not set. Please update it in agent.py.")

# Optionally, raise an error if the path is critical

root\_agent = LlmAgent(

model='gemini-2.0-flash',

name='web\_reader\_mcp\_client\_agent',

instruction="Use the 'load\_web\_page' tool to fetch content from a URL provided by the user.",

tools=[

MCPToolset(

connection\_params=StdioServerParameters(

command='python3', # Command to run your MCP server script

args=[PATH\_TO\_YOUR\_MCP\_SERVER\_SCRIPT], # Argument is the path to the script

)

# tool\_filter=['load\_web\_page'] # Optional: ensure only specific tools are loaded

)

],

)

And an \_\_init\_\_.py in the same directory:

# ./adk\_agent\_samples/mcp\_client\_agent/\_\_init\_\_.py

from . import agent

**To run the test:**

1. **Start your custom MCP server (optional, for separate observation):** You can run your my\_adk\_mcp\_server.py directly in one terminal to see its logs:

python3 /path/to/your/my\_adk\_mcp\_server.py

1. It will print "Launching MCP Server..." and wait. The ADK agent (run via adk web) will then connect to this process if the command in StdioServerParameters is set up to execute it. *(Alternatively, MCPToolset will start this server script as a subprocess automatically when the agent initializes).*
2. **Run adk web for the client agent:** Navigate to the parent directory of mcp\_client\_agent (e.g., adk\_agent\_samples) and run:

cd ./adk\_agent\_samples # Or your equivalent parent directory

adk web

1. **Interact in the ADK Web UI:**
   * Select the web\_reader\_mcp\_client\_agent.
   * Try a prompt like: "Load the content from https://example.com"

The ADK agent (web\_reader\_mcp\_client\_agent) will use MCPToolset to start and connect to your my\_adk\_mcp\_server.py. Your MCP server will receive the call\_tool request, execute the ADK load\_web\_page tool, and return the result. The ADK agent will then relay this information. You should see logs from both the ADK Web UI (and its terminal) and potentially from your my\_adk\_mcp\_server.py terminal if you ran it separately.

This example demonstrates how ADK tools can be encapsulated within an MCP server, making them accessible to a broader range of MCP-compliant clients, not just ADK agents.

Refer to the [documentation](https://modelcontextprotocol.io/quickstart/server#core-mcp-concepts), to try it out with Claude Desktop.

## Using MCP Tools in your own Agent out of adk web[¶](https://google.github.io/adk-docs/tools/mcp-tools/#using-mcp-tools-in-your-own-agent-out-of-adk-web)

This section is relevant to you if:

* You are developing your own Agent using ADK
* And, you are **NOT** using adk web,
* And, you are exposing the agent via your own UI

Using MCP Tools requires a different setup than using regular tools, due to the fact that specs for MCP Tools are fetched asynchronously from the MCP Server running remotely, or in another process.

The following example is modified from the "Example 1: File System MCP Server" example above. The main differences are:

1. Your tool and agent are created asynchronously
2. You need to properly manage the exit stack, so that your agents and tools are destructed properly when the connection to MCP Server is closed.

# agent.py (modify get\_tools\_async and other parts as needed)

# ./adk\_agent\_samples/mcp\_agent/agent.py

import asyncio

from dotenv import load\_dotenv

from google.genai import types

from google.adk.agents.llm\_agent import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.adk.artifacts.in\_memory\_artifact\_service import InMemoryArtifactService # Optional

from google.adk.tools.mcp\_tool.mcp\_toolset import MCPToolset, SseServerParams, StdioServerParameters

# Load environment variables from .env file in the parent directory

# Place this near the top, before using env vars like API keys

load\_dotenv('../.env')

# --- Step 1: Agent Definition ---

async def get\_agent\_async():

"""Creates an ADK Agent equipped with tools from the MCP Server."""

tools, exit\_stack = await MCPToolset.from\_server(

# Use StdioServerParameters for local process communication

connection\_params=StdioServerParameters(

command='npx', # Command to run the server

args=["-y", # Arguments for the command

"@modelcontextprotocol/server-filesystem",

# TODO: IMPORTANT! Change the path below to an ABSOLUTE path on your system.

"/path/to/your/folder"],

)

# For remote servers, you would use SseServerParams instead:

# connection\_params=SseServerParams(url="http://remote-server:port/path", headers={...})

)

print(f"Fetched {len(tools)} tools from MCP server.")

root\_agent = LlmAgent(

model='gemini-2.0-flash', # Adjust model name if needed based on availability

name='filesystem\_assistant',

instruction='Help user interact with the local filesystem using available tools.',

tools=tools, # Provide the MCP tools to the ADK agent

)

return root\_agent, exit\_stack

# --- Step 2: Main Execution Logic ---

async def async\_main():

session\_service = InMemorySessionService()

# Artifact service might not be needed for this example

artifacts\_service = InMemoryArtifactService()

session = await session\_service.create\_session(

state={}, app\_name='mcp\_filesystem\_app', user\_id='user\_fs'

)

# TODO: Change the query to be relevant to YOUR specified folder.

# e.g., "list files in the 'documents' subfolder" or "read the file 'notes.txt'"

query = "list files in the tests folder"

print(f"User Query: '{query}'")

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

root\_agent, exit\_stack = await get\_agent\_async()

runner = Runner(

app\_name='mcp\_filesystem\_app',

agent=root\_agent,

artifact\_service=artifacts\_service, # Optional

session\_service=session\_service,

)

print("Running agent...")

events\_async = runner.run\_async(

session\_id=session.id, user\_id=session.user\_id, new\_message=content

)

async for event in events\_async:

print(f"Event received: {event}")

# Crucial Cleanup: Ensure the MCP server process connection is closed.

print("Closing MCP server connection...")

await exit\_stack.aclose()

print("Cleanup complete.")

if \_\_name\_\_ == '\_\_main\_\_':

try:

asyncio.run(async\_main())

except Exception as e:

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

## Key considerations[¶](https://google.github.io/adk-docs/tools/mcp-tools/#key-considerations)

When working with MCP and ADK, keep these points in mind:

* **Protocol vs. Library:** MCP is a protocol specification, defining communication rules. ADK is a Python library/framework for building agents. MCPToolset bridges these by implementing the client side of the MCP protocol within the ADK framework. Conversely, building an MCP server in Python requires using the model-context-protocol library.
* **ADK Tools vs. MCP Tools:**
  + ADK Tools (BaseTool, FunctionTool, AgentTool, etc.) are Python objects designed for direct use within the ADK's LlmAgent and Runner.
  + MCP Tools are capabilities exposed by an MCP Server according to the protocol's schema. MCPToolset makes these look like ADK tools to an LlmAgent.
  + Langchain/CrewAI Tools are specific implementations within those libraries, often simple functions or classes, lacking the server/protocol structure of MCP. ADK offers wrappers (LangchainTool, CrewaiTool) for some interoperability.
* **Asynchronous nature:** Both ADK and the MCP Python library are heavily based on the asyncio Python library. Tool implementations and server handlers should generally be async functions.
* **Stateful sessions (MCP):** MCP establishes stateful, persistent connections between a client and server instance. This differs from typical stateless REST APIs.
  + **Deployment:** This statefulness can pose challenges for scaling and deployment, especially for remote servers handling many users. The original MCP design often assumed client and server were co-located. Managing these persistent connections requires careful infrastructure considerations (e.g., load balancing, session affinity).
  + **ADK MCPToolset:** Manages this connection lifecycle. The exit\_stack pattern shown in the examples is crucial for ensuring the connection (and potentially the server process) is properly terminated when the ADK agent finishes.

## Further Resources[¶](https://google.github.io/adk-docs/tools/mcp-tools/#further-resources)

* [Model Context Protocol Documentation](https://modelcontextprotocol.io/)
* [MCP Specification](https://modelcontextprotocol.io/specification/)
* [MCP Python SDK & Examples](https://github.com/modelcontextprotocol/)

# OpenAPI Integration[¶](https://google.github.io/adk-docs/tools/openapi-tools/#openapi-integration)

python_only

## Integrating REST APIs with OpenAPI[¶](https://google.github.io/adk-docs/tools/openapi-tools/#integrating-rest-apis-with-openapi)

ADK simplifies interacting with external REST APIs by automatically generating callable tools directly from an [OpenAPI Specification (v3.x)](https://swagger.io/specification/). This eliminates the need to manually define individual function tools for each API endpoint.

**Core Benefit**

Use OpenAPIToolset to instantly create agent tools (RestApiTool) from your existing API documentation (OpenAPI spec), enabling agents to seamlessly call your web services.

## Key Components[¶](https://google.github.io/adk-docs/tools/openapi-tools/#key-components)

* **OpenAPIToolset**: This is the primary class you'll use. You initialize it with your OpenAPI specification, and it handles the parsing and generation of tools.
* **RestApiTool**: This class represents a single, callable API operation (like GET /pets/{petId} or POST /pets). OpenAPIToolset creates one RestApiTool instance for each operation defined in your spec.

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

The process involves these main steps when you use OpenAPIToolset:

1. **Initialization & Parsing**:
   * You provide the OpenAPI specification to OpenAPIToolset either as a Python dictionary, a JSON string, or a YAML string.
   * The toolset internally parses the spec, resolving any internal references ($ref) to understand the complete API structure.
2. **Operation Discovery**:
   * It identifies all valid API operations (e.g., GET, POST, PUT, DELETE) defined within the paths object of your specification.
3. **Tool Generation**:
   * For each discovered operation, OpenAPIToolset automatically creates a corresponding RestApiTool instance.
   * **Tool Name**: Derived from the operationId in the spec (converted to snake\_case, max 60 chars). If operationId is missing, a name is generated from the method and path.
   * **Tool Description**: Uses the summary or description from the operation for the LLM.
   * **API Details**: Stores the required HTTP method, path, server base URL, parameters (path, query, header, cookie), and request body schema internally.
4. **RestApiTool Functionality**: Each generated RestApiTool:
   * **Schema Generation**: Dynamically creates a FunctionDeclaration based on the operation's parameters and request body. This schema tells the LLM how to call the tool (what arguments are expected).
   * **Execution**: When called by the LLM, it constructs the correct HTTP request (URL, headers, query params, body) using the arguments provided by the LLM and the details from the OpenAPI spec. It handles authentication (if configured) and executes the API call using the requests library.
   * **Response Handling**: Returns the API response (typically JSON) back to the agent flow.
5. **Authentication**: You can configure global authentication (like API keys or OAuth - see [Authentication](https://google.github.io/adk-docs/tools/authentication/) for details) when initializing OpenAPIToolset. This authentication configuration is automatically applied to all generated RestApiTool instances.

## Usage Workflow[¶](https://google.github.io/adk-docs/tools/openapi-tools/#usage-workflow)

Follow these steps to integrate an OpenAPI spec into your agent:

1. **Obtain Spec**: Get your OpenAPI specification document (e.g., load from a .json or .yaml file, fetch from a URL).
2. **Instantiate Toolset**: Create an OpenAPIToolset instance, passing the spec content and type (spec\_str/spec\_dict, spec\_str\_type). Provide authentication details (auth\_scheme, auth\_credential) if required by the API.

from google.adk.tools.openapi\_tool.openapi\_spec\_parser.openapi\_toolset import OpenAPIToolset

# Example with a JSON string

openapi\_spec\_json = '...' # Your OpenAPI JSON string

toolset = OpenAPIToolset(spec\_str=openapi\_spec\_json, spec\_str\_type="json")

# Example with a dictionary

# openapi\_spec\_dict = {...} # Your OpenAPI spec as a dict

# toolset = OpenAPIToolset(spec\_dict=openapi\_spec\_dict)

1. **Retrieve Tools**: Get the list of generated RestApiTool instances from the toolset.

api\_tools = toolset.get\_tools()

# Or get a specific tool by its generated name (snake\_case operationId)

# specific\_tool = toolset.get\_tool("list\_pets")

1. **Add to Agent**: Include the retrieved tools in your LlmAgent's tools list.

from google.adk.agents import LlmAgent

my\_agent = LlmAgent(

name="api\_interacting\_agent",

model="gemini-2.0-flash", # Or your preferred model

tools=api\_tools, # Pass the list of generated tools

# ... other agent config ...

)

1. **Instruct Agent**: Update your agent's instructions to inform it about the new API capabilities and the names of the tools it can use (e.g., list\_pets, create\_pet). The tool descriptions generated from the spec will also help the LLM.
2. **Run Agent**: Execute your agent using the Runner. When the LLM determines it needs to call one of the APIs, it will generate a function call targeting the appropriate RestApiTool, which will then handle the HTTP request automatically.

## Example[¶](https://google.github.io/adk-docs/tools/openapi-tools/#example)

This example demonstrates generating tools from a simple Pet Store OpenAPI spec (using httpbin.org for mock responses) and interacting with them via an agent.

**Code: Pet Store API**

**openapi\_example.py**

import asyncio

import uuid # For unique session IDs

from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

# --- OpenAPI Tool Imports ---

from google.adk.tools.openapi\_tool.openapi\_spec\_parser.openapi\_toolset import OpenAPIToolset

# --- Constants ---

APP\_NAME\_OPENAPI = "openapi\_petstore\_app"

USER\_ID\_OPENAPI = "user\_openapi\_1"

SESSION\_ID\_OPENAPI = f"session\_openapi\_{uuid.uuid4()}" # Unique session ID

AGENT\_NAME\_OPENAPI = "petstore\_manager\_agent"

GEMINI\_MODEL = "gemini-2.0-flash"

# --- Sample OpenAPI Specification (JSON String) ---

# A basic Pet Store API example using httpbin.org as a mock server

openapi\_spec\_string = """

{

"openapi": "3.0.0",

"info": {

"title": "Simple Pet Store API (Mock)",

"version": "1.0.1",

"description": "An API to manage pets in a store, using httpbin for responses."

},

"servers": [

{

"url": "https://httpbin.org",

"description": "Mock server (httpbin.org)"

}

],

"paths": {

"/get": {

"get": {

"summary": "List all pets (Simulated)",

"operationId": "listPets",

"description": "Simulates returning a list of pets. Uses httpbin's /get endpoint which echoes query parameters.",

"parameters": [

{

"name": "limit",

"in": "query",

"description": "Maximum number of pets to return",

"required": false,

"schema": { "type": "integer", "format": "int32" }

},

{

"name": "status",

"in": "query",

"description": "Filter pets by status",

"required": false,

"schema": { "type": "string", "enum": ["available", "pending", "sold"] }

}

],

"responses": {

"200": {

"description": "A list of pets (echoed query params).",

"content": { "application/json": { "schema": { "type": "object" } } }

}

}

}

},

"/post": {

"post": {

"summary": "Create a pet (Simulated)",

"operationId": "createPet",

"description": "Simulates adding a new pet. Uses httpbin's /post endpoint which echoes the request body.",

"requestBody": {

"description": "Pet object to add",

"required": true,

"content": {

"application/json": {

"schema": {

"type": "object",

"required": ["name"],

"properties": {

"name": {"type": "string", "description": "Name of the pet"},

"tag": {"type": "string", "description": "Optional tag for the pet"}

}

}

}

}

},

"responses": {

"201": {

"description": "Pet created successfully (echoed request body).",

"content": { "application/json": { "schema": { "type": "object" } } }

}

}

}

},

"/get?petId={petId}": {

"get": {

"summary": "Info for a specific pet (Simulated)",

"operationId": "showPetById",

"description": "Simulates returning info for a pet ID. Uses httpbin's /get endpoint.",

"parameters": [

{

"name": "petId",

"in": "path",

"description": "This is actually passed as a query param to httpbin /get",

"required": true,

"schema": { "type": "integer", "format": "int64" }

}

],

"responses": {

"200": {

"description": "Information about the pet (echoed query params)",

"content": { "application/json": { "schema": { "type": "object" } } }

},

"404": { "description": "Pet not found (simulated)" }

}

}

}

}

}

"""

# --- Create OpenAPIToolset ---

generated\_tools\_list = []

try:

# Instantiate the toolset with the spec string

petstore\_toolset = OpenAPIToolset(

spec\_str=openapi\_spec\_string,

spec\_str\_type="json"

# No authentication needed for httpbin.org

)

# Get all tools generated from the spec

generated\_tools\_list = petstore\_toolset.get\_tools()

print(f"Generated {len(generated\_tools\_list)} tools from OpenAPI spec:")

for tool in generated\_tools\_list:

# Tool names are snake\_case versions of operationId

print(f"- Tool Name: '{tool.name}', Description: {tool.description[:60]}...")

except ValueError as ve:

print(f"Validation Error creating OpenAPIToolset: {ve}")

# Handle error appropriately, maybe exit or skip agent creation

except Exception as e:

print(f"Unexpected Error creating OpenAPIToolset: {e}")

# Handle error appropriately

# --- Agent Definition ---

openapi\_agent = LlmAgent(

name=AGENT\_NAME\_OPENAPI,

model=GEMINI\_MODEL,

tools=generated\_tools\_list, # Pass the list of RestApiTool objects

instruction=f"""You are a Pet Store assistant managing pets via an API.

Use the available tools to fulfill user requests.

Available tools: {', '.join([t.name for t in generated\_tools\_list])}.

When creating a pet, confirm the details echoed back by the API.

When listing pets, mention any filters used (like limit or status).

When showing a pet by ID, state the ID you requested.

""",

description="Manages a Pet Store using tools generated from an OpenAPI spec."

)

# --- Session and Runner Setup ---

session\_service\_openapi = InMemorySessionService()

runner\_openapi = Runner(

agent=openapi\_agent, app\_name=APP\_NAME\_OPENAPI, session\_service=session\_service\_openapi

)

session\_openapi = session\_service\_openapi.create\_session(

app\_name=APP\_NAME\_OPENAPI, user\_id=USER\_ID\_OPENAPI, session\_id=SESSION\_ID\_OPENAPI

)

# --- Agent Interaction Function ---

async def call\_openapi\_agent\_async(query):

print("\n--- Running OpenAPI Pet Store Agent ---")

print(f"Query: {query}")

if not generated\_tools\_list:

print("Skipping execution: No tools were generated.")

print("-" \* 30)

return

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

final\_response\_text = "Agent did not provide a final text response."

try:

async for event in runner\_openapi.run\_async(

user\_id=USER\_ID\_OPENAPI, session\_id=SESSION\_ID\_OPENAPI, new\_message=content

):

# Optional: Detailed event logging for debugging

# print(f" DEBUG Event: Author={event.author}, Type={'Final' if event.is\_final\_response() else 'Intermediate'}, Content={str(event.content)[:100]}...")

if event.get\_function\_calls():

call = event.get\_function\_calls()[0]

print(f" Agent Action: Called function '{call.name}' with args {call.args}")

elif event.get\_function\_responses():

response = event.get\_function\_responses()[0]

print(f" Agent Action: Received response for '{response.name}'")

# print(f" Tool Response Snippet: {str(response.response)[:200]}...") # Uncomment for response details

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

# Capture the last final text response

final\_response\_text = event.content.parts[0].text.strip()

print(f"Agent Final Response: {final\_response\_text}")

except Exception as e:

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

import traceback

traceback.print\_exc() # Print full traceback for errors

print("-" \* 30)

# --- Run Examples ---

async def run\_openapi\_example():

# Trigger listPets

await call\_openapi\_agent\_async("Show me the pets available.")

# Trigger createPet

await call\_openapi\_agent\_async("Please add a new dog named 'Dukey'.")

# Trigger showPetById

await call\_openapi\_agent\_async("Get info for pet with ID 123.")

# --- Execute ---

if \_\_name\_\_ == "\_\_main\_\_":

print("Executing OpenAPI example...")

# Use asyncio.run() for top-level execution

try:

asyncio.run(run\_openapi\_example())

except RuntimeError as e:

if "cannot be called from a running event loop" in str(e):

print("Info: Cannot run asyncio.run from a running event loop (e.g., Jupyter/Colab).")

# If in Jupyter/Colab, you might need to run like this:

# await run\_openapi\_example()

else:

raise e

print("OpenAPI example finished.")

Back to top

[Previous](https://google.github.io/adk-docs/tools/mcp-tools/)

[MCP tools](https://google.github.io/adk-docs/tools/mcp-tools/)

[Next](https://google.github.io/adk-docs/tools/authentication/)

[Authentication](https://google.github.io/adk-docs/tools/authentication/)

Copyright Google 2025

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

# Authenticating with Tools[¶](https://google.github.io/adk-docs/tools/authentication/#authenticating-with-tools)

python_only

## Core Concepts[¶](https://google.github.io/adk-docs/tools/authentication/#core-concepts)

Many tools need to access protected resources (like user data in Google Calendar, Salesforce records, etc.) and require authentication. ADK provides a system to handle various authentication methods securely.

The key components involved are:

1. **AuthScheme**: Defines *how* an API expects authentication credentials (e.g., as an API Key in a header, an OAuth 2.0 Bearer token). ADK supports the same types of authentication schemes as OpenAPI 3.0. To know more about what each type of credential is, refer to [OpenAPI doc: Authentication](https://swagger.io/docs/specification/v3_0/authentication/). ADK uses specific classes like APIKey, HTTPBearer, OAuth2, OpenIdConnectWithConfig.
2. **AuthCredential**: Holds the *initial* information needed to *start* the authentication process (e.g., your application's OAuth Client ID/Secret, an API key value). It includes an auth\_type (like API\_KEY, OAUTH2, SERVICE\_ACCOUNT) specifying the credential type.

The general flow involves providing these details when configuring a tool. ADK then attempts to automatically exchange the initial credential for a usable one (like an access token) before the tool makes an API call. For flows requiring user interaction (like OAuth consent), a specific interactive process involving the Agent Client application is triggered.

## Supported Initial Credential Types[¶](https://google.github.io/adk-docs/tools/authentication/#supported-initial-credential-types)

* **API\_KEY:** For simple key/value authentication. Usually requires no exchange.
* **HTTP:** Can represent Basic Auth (not recommended/supported for exchange) or already obtained Bearer tokens. If it's a Bearer token, no exchange is needed.
* **OAUTH2:** For standard OAuth 2.0 flows. Requires configuration (client ID, secret, scopes) and often triggers the interactive flow for user consent.
* **OPEN\_ID\_CONNECT:** For authentication based on OpenID Connect. Similar to OAuth2, often requires configuration and user interaction.
* **SERVICE\_ACCOUNT:** For Google Cloud Service Account credentials (JSON key or Application Default Credentials). Typically exchanged for a Bearer token.

## Configuring Authentication on Tools[¶](https://google.github.io/adk-docs/tools/authentication/#configuring-authentication-on-tools)

You set up authentication when defining your tool:

* **RestApiTool / OpenAPIToolset**: Pass auth\_scheme and auth\_credential during initialization
* **GoogleApiToolSet Tools**: ADK has built-in 1st party tools like Google Calendar, BigQuery etc,. Use the toolset's specific method.
* **APIHubToolset / ApplicationIntegrationToolset**: Pass auth\_scheme and auth\_credentialduring initialization, if the API managed in API Hub / provided by Application Integration requires authentication.

**WARNING**

Storing sensitive credentials like access tokens and especially refresh tokens directly in the session state might pose security risks depending on your session storage backend (SessionService) and overall application security posture.

* **InMemorySessionService:** Suitable for testing and development, but data is lost when the process ends. Less risk as it's transient.
* **Database/Persistent Storage:** **Strongly consider encrypting** the token data before storing it in the database using a robust encryption library (like cryptography) and managing encryption keys securely (e.g., using a key management service).
* **Secure Secret Stores:** For production environments, storing sensitive credentials in a dedicated secret manager (like Google Cloud Secret Manager or HashiCorp Vault) is the **most recommended approach**. Your tool could potentially store only short-lived access tokens or secure references (not the refresh token itself) in the session state, fetching the necessary secrets from the secure store when needed.

## Journey 1: Building Agentic Applications with Authenticated Tools[¶](https://google.github.io/adk-docs/tools/authentication/#journey-1-building-agentic-applications-with-authenticated-tools)

This section focuses on using pre-existing tools (like those from RestApiTool/ OpenAPIToolset, APIHubToolset, GoogleApiToolSet) that require authentication within your agentic application. Your main responsibility is configuring the tools and handling the client-side part of interactive authentication flows (if required by the tool).

### 1. Configuring Tools with Authentication[¶](https://google.github.io/adk-docs/tools/authentication/#1-configuring-tools-with-authentication)

When adding an authenticated tool to your agent, you need to provide its required AuthScheme and your application's initial AuthCredential.

**A. Using OpenAPI-based Toolsets (OpenAPIToolset, APIHubToolset, etc.)**

Pass the scheme and credential during toolset initialization. The toolset applies them to all generated tools. Here are few ways to create tools with authentication in ADK.

**[API Key](https://google.github.io/adk-docs/tools/authentication/#api-key)**

**[OAuth2](https://google.github.io/adk-docs/tools/authentication/#oauth2)**

**[Service Account](https://google.github.io/adk-docs/tools/authentication/#service-account)**

**[OpenID connect](https://google.github.io/adk-docs/tools/authentication/#openid-connect)**

Create a tool requiring an API Key.

from google.adk.tools.openapi\_tool.auth.auth\_helpers import token\_to\_scheme\_credential

from google.adk.tools.apihub\_tool.apihub\_toolset import APIHubToolset

auth\_scheme, auth\_credential = token\_to\_scheme\_credential(

"apikey", "query", "apikey", YOUR\_API\_KEY\_STRING

)

sample\_api\_toolset = APIHubToolset(

name="sample-api-requiring-api-key",

description="A tool using an API protected by API Key",

apihub\_resource\_name="...",

auth\_scheme=auth\_scheme,

auth\_credential=auth\_credential,

)

**B. Using Google API Toolsets (e.g., calendar\_tool\_set)**

These toolsets often have dedicated configuration methods.

Tip: For how to create a Google OAuth Client ID & Secret, see this guide: [Get your Google API Client ID](https://developers.google.com/identity/gsi/web/guides/get-google-api-clientid#get_your_google_api_client_id)

# Example: Configuring Google Calendar Tools

from google.adk.tools.google\_api\_tool import calendar\_tool\_set

client\_id = "YOUR\_GOOGLE\_OAUTH\_CLIENT\_ID.apps.googleusercontent.com"

client\_secret = "YOUR\_GOOGLE\_OAUTH\_CLIENT\_SECRET"

# Use the specific configure method for this toolset type

calendar\_tool\_set.configure\_auth(

client\_id=oauth\_client\_id, client\_secret=oauth\_client\_secret

)

# agent = LlmAgent(..., tools=calendar\_tool\_set.get\_tool('calendar\_tool\_set'))

The sequence diagram of auth request flow (where tools are requesting auth credentials) looks like below:

Authentication

### 2. Handling the Interactive OAuth/OIDC Flow (Client-Side)[¶](https://google.github.io/adk-docs/tools/authentication/#2-handling-the-interactive-oauthoidc-flow-client-side)

If a tool requires user login/consent (typically OAuth 2.0 or OIDC), the ADK framework pauses execution and signals your **Agent Client** application. There are two cases:

* **Agent Client** application runs the agent directly (via runner.run\_async) in the same process. e.g. UI backend, CLI app, or Spark job etc.
* **Agent Client** application interacts with ADK's fastapi server via /run or /run\_sse endpoint. While ADK's fastapi server could be setup on the same server or different server as **Agent Client** application

The second case is a special case of first case, because /run or /run\_sse endpoint also invokes runner.run\_async. The only differences are:

* Whether to call a python function to run the agent (first case) or call a service endpoint to run the agent (second case).
* Whether the result events are in-memory objects (first case) or serialized json string in http response (second case).

Below sections focus on the first case and you should be able to map it to the second case very straightforward. We will also describe some differences to handle for the second case if necessary.

Here's the step-by-step process for your client application:

**Step 1: Run Agent & Detect Auth Request**

* Initiate the agent interaction using runner.run\_async.
* Iterate through the yielded events.
* Look for a specific function call event whose function call has a special name: adk\_request\_credential. This event signals that user interaction is needed. You can use helper functions to identify this event and extract necessary information. (For the second case, the logic is similar. You deserialize the event from the http response).

# runner = Runner(...)

# session = await session\_service.create\_session(...)

# content = types.Content(...) # User's initial query

print("\nRunning agent...")

events\_async = runner.run\_async(

session\_id=session.id, user\_id='user', new\_message=content

)

auth\_request\_function\_call\_id, auth\_config = None, None

async for event in events\_async:

# Use helper to check for the specific auth request event

if (auth\_request\_function\_call := get\_auth\_request\_function\_call(event)):

print("--> Authentication required by agent.")

# Store the ID needed to respond later

if not (auth\_request\_function\_call\_id := auth\_request\_function\_call.id):

raise ValueError(f'Cannot get function call id from function call: {auth\_request\_function\_call}')

# Get the AuthConfig containing the auth\_uri etc.

auth\_config = get\_auth\_config(auth\_request\_function\_call)

break # Stop processing events for now, need user interaction

if not auth\_request\_function\_call\_id:

print("\nAuth not required or agent finished.")

# return # Or handle final response if received

*Helper functions helpers.py:*

from google.adk.events import Event

from google.adk.auth import AuthConfig # Import necessary type

from google.genai import types

def get\_auth\_request\_function\_call(event: Event) -> types.FunctionCall:

# Get the special auth request function call from the event

if not event.content or event.content.parts:

return

for part in event.content.parts:

if (

part

and part.function\_call

and part.function\_call.name == 'adk\_request\_credential'

and event.long\_running\_tool\_ids

and part.function\_call.id in event.long\_running\_tool\_ids

):

return part.function\_call

def get\_auth\_config(auth\_request\_function\_call: types.FunctionCall) -> AuthConfig:

# Extracts the AuthConfig object from the arguments of the auth request function call

if not auth\_request\_function\_call.args or not (auth\_config := auth\_request\_function\_call.args.get('auth\_config')):

raise ValueError(f'Cannot get auth config from function call: {auth\_request\_function\_call}')

if not isinstance(auth\_config, AuthConfig):

raise ValueError(f'Cannot get auth config {auth\_config} is not an instance of AuthConfig.')

return auth\_config

**Step 2: Redirect User for Authorization**

* Get the authorization URL (auth\_uri) from the auth\_config extracted in the previous step.
* **Crucially, append your application's** redirect\_uri as a query parameter to this auth\_uri. This redirect\_uri must be pre-registered with your OAuth provider (e.g., [Google Cloud Console](https://developers.google.com/identity/protocols/oauth2/web-server#creatingcred), [Okta admin panel](https://developer.okta.com/docs/guides/sign-into-web-app-redirect/spring-boot/main/#create-an-app-integration-in-the-admin-console)).
* Direct the user to this complete URL (e.g., open it in their browser).

# (Continuing after detecting auth needed)

if auth\_request\_function\_call\_id and auth\_config:

# Get the base authorization URL from the AuthConfig

base\_auth\_uri = auth\_config.exchanged\_auth\_credential.oauth2.auth\_uri

if base\_auth\_uri:

redirect\_uri = 'http://localhost:8000/callback' # MUST match your OAuth client app config

# Append redirect\_uri (use urlencode in production)

auth\_request\_uri = base\_auth\_uri + f'&redirect\_uri={redirect\_uri}'

# Now you need to redirect your end user to this auth\_request\_uri or ask them to open this auth\_request\_uri in their browser

# This auth\_request\_uri should be served by the corresponding auth provider and the end user should login and authorize your applicaiton to access their data

# And then the auth provider will redirect the end user to the redirect\_uri you provided

# Next step: Get this callback URL from the user (or your web server handler)

else:

print("ERROR: Auth URI not found in auth\_config.")

# Handle error

**Step 3. Handle the Redirect Callback (Client):**

* Your application must have a mechanism (e.g., a web server route at the redirect\_uri) to receive the user after they authorize the application with the provider.
* The provider redirects the user to your redirect\_uri and appends an authorization\_code (and potentially state, scope) as query parameters to the URL.
* Capture the **full callback URL** from this incoming request.
* (This step happens outside the main agent execution loop, in your web server or equivalent callback handler.)

**Step 4. Send Authentication Result Back to ADK (Client):**

* Once you have the full callback URL (containing the authorization code), retrieve the auth\_request\_function\_call\_id and the auth\_config object saved in Client Step 1.
* Set the captured callback URL into the exchanged\_auth\_credential.oauth2.auth\_response\_uri field. Also ensure exchanged\_auth\_credential.oauth2.redirect\_uri contains the redirect URI you used.
* Create a types.Content object containing a types.Part with a types.FunctionResponse.
  + Set name to "adk\_request\_credential". (Note: This is a special name for ADK to proceed with authentication. Do not use other names.)
  + Set id to the auth\_request\_function\_call\_id you saved.
  + Set response to the *serialized* (e.g., .model\_dump()) updated AuthConfig object.
* Call runner.run\_async **again** for the same session, passing this FunctionResponse content as the new\_message.

# (Continuing after user interaction)

# Simulate getting the callback URL (e.g., from user paste or web handler)

auth\_response\_uri = await get\_user\_input(

f'Paste the full callback URL here:\n> '

)

auth\_response\_uri = auth\_response\_uri.strip() # Clean input

if not auth\_response\_uri:

print("Callback URL not provided. Aborting.")

return

# Update the received AuthConfig with the callback details

auth\_config.exchanged\_auth\_credential.oauth2.auth\_response\_uri = auth\_response\_uri

# Also include the redirect\_uri used, as the token exchange might need it

auth\_config.exchanged\_auth\_credential.oauth2.redirect\_uri = redirect\_uri

# Construct the FunctionResponse Content object

auth\_content = types.Content(

role='user', # Role can be 'user' when sending a FunctionResponse

parts=[

types.Part(

function\_response=types.FunctionResponse(

id=auth\_request\_function\_call\_id, # Link to the original request

name='adk\_request\_credential', # Special framework function name

response=auth\_config.model\_dump() # Send back the \*updated\* AuthConfig

)

)

],

)

# --- Resume Execution ---

print("\nSubmitting authentication details back to the agent...")

events\_async\_after\_auth = runner.run\_async(

session\_id=session.id,

user\_id='user',

new\_message=auth\_content, # Send the FunctionResponse back

)

# --- Process Final Agent Output ---

print("\n--- Agent Response after Authentication ---")

async for event in events\_async\_after\_auth:

# Process events normally, expecting the tool call to succeed now

print(event) # Print the full event for inspection

**Step 5: ADK Handles Token Exchange & Tool Retry and gets Tool result**

* ADK receives the FunctionResponse for adk\_request\_credential.
* It uses the information in the updated AuthConfig (including the callback URL containing the code) to perform the OAuth **token exchange** with the provider's token endpoint, obtaining the access token (and possibly refresh token).
* ADK internally makes these tokens available by setting them in the session state).
* ADK **automatically retries** the original tool call (the one that initially failed due to missing auth).
* This time, the tool finds the valid tokens (via tool\_context.get\_auth\_response()) and successfully executes the authenticated API call.
* The agent receives the actual result from the tool and generates its final response to the user.

The sequence diagram of auth response flow (where Agent Client send back the auth response and ADK retries tool calling) looks like below:

Authentication

## Journey 2: Building Custom Tools (FunctionTool) Requiring Authentication[¶](https://google.github.io/adk-docs/tools/authentication/#journey-2-building-custom-tools-functiontool-requiring-authentication)

This section focuses on implementing the authentication logic *inside* your custom Python function when creating a new ADK Tool. We will implement a FunctionTool as an example.

### Prerequisites[¶](https://google.github.io/adk-docs/tools/authentication/#prerequisites)

Your function signature *must* include [tool\_context: ToolContext](https://google.github.io/adk-docs/tools/#tool-context). ADK automatically injects this object, providing access to state and auth mechanisms.

from google.adk.tools import FunctionTool, ToolContext

from typing import Dict

def my\_authenticated\_tool\_function(param1: str, ..., tool\_context: ToolContext) -> dict:

# ... your logic ...

pass

my\_tool = FunctionTool(func=my\_authenticated\_tool\_function)

### Authentication Logic within the Tool Function[¶](https://google.github.io/adk-docs/tools/authentication/#authentication-logic-within-the-tool-function)

Implement the following steps inside your function:

**Step 1: Check for Cached & Valid Credentials:**

Inside your tool function, first check if valid credentials (e.g., access/refresh tokens) are already stored from a previous run in this session. Credentials for the current sessions should be stored in tool\_context.invocation\_context.session.state (a dictionary of state) Check existence of existing credentials by checking tool\_context.invocation\_context.session.state.get(credential\_name, None).

# Inside your tool function

TOKEN\_CACHE\_KEY = "my\_tool\_tokens" # Choose a unique key

SCOPES = ["scope1", "scope2"] # Define required scopes

creds = None

cached\_token\_info = tool\_context.state.get(TOKEN\_CACHE\_KEY)

if cached\_token\_info:

try:

creds = Credentials.from\_authorized\_user\_info(cached\_token\_info, SCOPES)

if not creds.valid and creds.expired and creds.refresh\_token:

creds.refresh(Request())

tool\_context.state[TOKEN\_CACHE\_KEY] = json.loads(creds.to\_json()) # Update cache

elif not creds.valid:

creds = None # Invalid, needs re-auth

tool\_context.state[TOKEN\_CACHE\_KEY] = None

except Exception as e:

print(f"Error loading/refreshing cached creds: {e}")

creds = None

tool\_context.state[TOKEN\_CACHE\_KEY] = None

if creds and creds.valid:

# Skip to Step 5: Make Authenticated API Call

pass

else:

# Proceed to Step 2...

pass

**Step 2: Check for Auth Response from Client**

* If Step 1 didn't yield valid credentials, check if the client just completed the interactive flow by calling exchanged\_credential = tool\_context.get\_auth\_response().
* This returns the updated exchanged\_credential object sent back by the client (containing the callback URL in auth\_response\_uri).

# Use auth\_scheme and auth\_credential configured in the tool.

# exchanged\_credential: AuthCredential | None

exchanged\_credential = tool\_context.get\_auth\_response(AuthConfig(

auth\_scheme=auth\_scheme,

raw\_auth\_credential=auth\_credential,

))

# If exchanged\_credential is not None, then there is already an exchanged credetial from the auth response.

if exchanged\_credential:

# ADK exchanged the access token already for us

access\_token = auth\_response.oauth2.access\_token

refresh\_token = auth\_response.oauth2.refresh\_token

creds = Credentials(

token=access\_token,

refresh\_token=refresh\_token,

token\_uri=auth\_scheme.flows.authorizationCode.tokenUrl,

client\_id=oauth\_client\_id,

client\_secret=oauth\_client\_secret,

scopes=list(auth\_scheme.flows.authorizationCode.scopes.keys()),

)

# Cache the token in session state and call the API, skip to step 5

**Step 3: Initiate Authentication Request**

If no valid credentials (Step 1.) and no auth response (Step 2.) are found, the tool needs to start the OAuth flow. Define the AuthScheme and initial AuthCredential and call tool\_context.request\_credential(). Return a response indicating authorization is needed.

# Use auth\_scheme and auth\_credential configured in the tool.

tool\_context.request\_credential(AuthConfig(

auth\_scheme=auth\_scheme,

raw\_auth\_credential=auth\_credential,

))

return {'pending': true, 'message': 'Awaiting user authentication.'}

# By setting request\_credential, ADK detects a pending authentication event. It pauses execution and ask end user to login.

**Step 4: Exchange Authorization Code for Tokens**

ADK automatically generates oauth authorization URL and presents it to your Agent Client application. your Agent Client application should follow the same way described in Journey 1 to redirect the user to the authorization URL (with redirect\_uri appended). Once a user completes the login flow following the authorization URL and ADK extracts the authentication callback url from Agent Client applications, automatically parses the auth code, and generates auth token. At the next Tool call, tool\_context.get\_auth\_response in step 2 will contain a valid credential to use in subsequent API calls.

**Step 5: Cache Obtained Credentials**

After successfully obtaining the token from ADK (Step 2) or if the token is still valid (Step 1), **immediately store** the new Credentials object in tool\_context.state (serialized, e.g., as JSON) using your cache key.

# Inside your tool function, after obtaining 'creds' (either refreshed or newly exchanged)

# Cache the new/refreshed tokens

tool\_context.state[TOKEN\_CACHE\_KEY] = json.loads(creds.to\_json())

print(f"DEBUG: Cached/updated tokens under key: {TOKEN\_CACHE\_KEY}")

# Proceed to Step 6 (Make API Call)

**Step 6: Make Authenticated API Call**

* Once you have a valid Credentials object (creds from Step 1 or Step 4), use it to make the actual call to the protected API using the appropriate client library (e.g., googleapiclient, requests). Pass the credentials=creds argument.
* Include error handling, especially for HttpError 401/403, which might mean the token expired or was revoked between calls. If you get such an error, consider clearing the cached token (tool\_context.state.pop(...)) and potentially returning the auth\_required status again to force re-authentication.

# Inside your tool function, using the valid 'creds' object

# Ensure creds is valid before proceeding

if not creds or not creds.valid:

return {"status": "error", "error\_message": "Cannot proceed without valid credentials."}

try:

service = build("calendar", "v3", credentials=creds) # Example

api\_result = service.events().list(...).execute()

# Proceed to Step 7

except Exception as e:

# Handle API errors (e.g., check for 401/403, maybe clear cache and re-request auth)

print(f"ERROR: API call failed: {e}")

return {"status": "error", "error\_message": f"API call failed: {e}"}

**Step 7: Return Tool Result**

* After a successful API call, process the result into a dictionary format that is useful for the LLM.
* **Crucially, include a** along with the data.

# Inside your tool function, after successful API call

processed\_result = [...] # Process api\_result for the LLM

return {"status": "success", "data": processed\_result}

**Full Code**

**[Tools and Agent](https://google.github.io/adk-docs/tools/authentication/#tools-and-agent)**

**[Agent CLI](https://google.github.io/adk-docs/tools/authentication/#agent-cli)**

**[Helper](https://google.github.io/adk-docs/tools/authentication/#helper)**

**[Spec](https://google.github.io/adk-docs/tools/authentication/#spec)**

**tools\_and\_agent.py**

import asyncio

from dotenv import load\_dotenv

from google.adk.artifacts.in\_memory\_artifact\_service import InMemoryArtifactService

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

from .helpers import is\_pending\_auth\_event, get\_function\_call\_id, get\_function\_call\_auth\_config, get\_user\_input

from .tools\_and\_agent import root\_agent

load\_dotenv()

agent = root\_agent

async def async\_main():

"""

Main asynchronous function orchestrating the agent interaction and authentication flow.

"""

# --- Step 1: Service Initialization ---

# Use in-memory services for session and artifact storage (suitable for demos/testing).

session\_service = InMemorySessionService()

artifacts\_service = InMemoryArtifactService()

# Create a new user session to maintain conversation state.

session = session\_service.create\_session(

state={}, # Optional state dictionary for session-specific data

app\_name='my\_app', # Application identifier

user\_id='user' # User identifier

)

# --- Step 2: Initial User Query ---

# Define the user's initial request.

query = 'Show me my user info'

print(f"user: {query}")

# Format the query into the Content structure expected by the ADK Runner.

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

# Initialize the ADK Runner

runner = Runner(

app\_name='my\_app',

agent=agent,

artifact\_service=artifacts\_service,

session\_service=session\_service,

)

# --- Step 3: Send Query and Handle Potential Auth Request ---

print("\nRunning agent with initial query...")

events\_async = runner.run\_async(

session\_id=session.id, user\_id='user', new\_message=content

)

# Variables to store details if an authentication request occurs.

auth\_request\_event\_id, auth\_config = None, None

# Iterate through the events generated by the first run.

async for event in events\_async:

# Check if this event is the specific 'adk\_request\_credential' function call.

if is\_pending\_auth\_event(event):

print("--> Authentication required by agent.")

auth\_request\_event\_id = get\_function\_call\_id(event)

auth\_config = get\_function\_call\_auth\_config(event)

# Once the auth request is found and processed, exit this loop.

# We need to pause execution here to get user input for authentication.

break

# If no authentication request was detected after processing all events, exit.

if not auth\_request\_event\_id or not auth\_config:

print("\nAuthentication not required for this query or processing finished.")

return # Exit the main function

# --- Step 4: Manual Authentication Step (Simulated OAuth 2.0 Flow) ---

# This section simulates the user interaction part of an OAuth 2.0 flow.

# In a real web application, this would involve browser redirects.

# Define the Redirect URI. This \*must\* match one of the URIs registered

# with the OAuth provider for your application. The provider sends the user

# back here after they approve the request.

redirect\_uri = 'http://localhost:8000/dev-ui' # Example for local development

# Construct the Authorization URL that the user must visit.

# This typically includes the provider's authorization endpoint URL,

# client ID, requested scopes, response type (e.g., 'code'), and the redirect URI.

# Here, we retrieve the base authorization URI from the AuthConfig provided by ADK

# and append the redirect\_uri.

# NOTE: A robust implementation would use urlencode and potentially add state, scope, etc.

auth\_request\_uri = (

auth\_config.exchanged\_auth\_credential.oauth2.auth\_uri

+ f'&redirect\_uri={redirect\_uri}' # Simple concatenation; ensure correct query param format

)

print("\n--- User Action Required ---")

# Prompt the user to visit the authorization URL, log in, grant permissions,

# and then paste the \*full\* URL they are redirected back to (which contains the auth code).

auth\_response\_uri = await get\_user\_input(

f'1. Please open this URL in your browser to log in:\n {auth\_request\_uri}\n\n'

f'2. After successful login and authorization, your browser will be redirected.\n'

f' Copy the \*entire\* URL from the browser\'s address bar.\n\n'

f'3. Paste the copied URL here and press Enter:\n\n> '

)

# --- Step 5: Prepare Authentication Response for the Agent ---

# Update the AuthConfig object with the information gathered from the user.

# The ADK framework needs the full response URI (containing the code)

# and the original redirect URI to complete the OAuth token exchange process internally.

auth\_config.exchanged\_auth\_credential.oauth2.auth\_response\_uri = auth\_response\_uri

auth\_config.exchanged\_auth\_credential.oauth2.redirect\_uri = redirect\_uri

# Construct a FunctionResponse Content object to send back to the agent/runner.

# This response explicitly targets the 'adk\_request\_credential' function call

# identified earlier by its ID.

auth\_content = types.Content(

role='user',

parts=[

types.Part(

function\_response=types.FunctionResponse(

# Crucially, link this response to the original request using the saved ID.

id=auth\_request\_event\_id,

# The special name of the function call we are responding to.

name='adk\_request\_credential',

# The payload containing all necessary authentication details.

response=auth\_config.model\_dump(),

)

)

],

)

# --- Step 6: Resume Execution with Authentication ---

print("\nSubmitting authentication details back to the agent...")

# Run the agent again, this time providing the `auth\_content` (FunctionResponse).

# The ADK Runner intercepts this, processes the 'adk\_request\_credential' response

# (performs token exchange, stores credentials), and then allows the agent

# to retry the original tool call that required authentication, now succeeding with

# a valid access token embedded.

events\_async = runner.run\_async(

session\_id=session.id,

user\_id='user',

new\_message=auth\_content, # Provide the prepared auth response

)

# Process and print the final events from the agent after authentication is complete.

# This stream now contain the actual result from the tool (e.g., the user info).

print("\n--- Agent Response after Authentication ---")

async for event in events\_async:

print(event)

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(async\_main())

Back to top

[Previous](https://google.github.io/adk-docs/tools/openapi-tools/)

[OpenAPI tools](https://google.github.io/adk-docs/tools/openapi-tools/)

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

[Agent Runtime](https://google.github.io/adk-docs/runtime/)

Copyright Google 2025

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

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

## What are Context[¶](https://google.github.io/adk-docs/context/#what-are-context)

In the Agent Development Kit (ADK), "context" refers to the crucial bundle of information available to your agent and its tools during specific operations. Think of it as the necessary background knowledge and resources needed to handle a current task or conversation turn effectively.

Agents often need more than just the latest user message to perform well. Context is essential because it enables:

1. **Maintaining State:** Remembering details across multiple steps in a conversation (e.g., user preferences, previous calculations, items in a shopping cart). This is primarily managed through **session state**.
2. **Passing Data:** Sharing information discovered or generated in one step (like an LLM call or a tool execution) with subsequent steps. Session state is key here too.
3. **Accessing Services:** Interacting with framework capabilities like:
   * **Artifact Storage:** Saving or loading files or data blobs (like PDFs, images, configuration files) associated with the session.
   * **Memory:** Searching for relevant information from past interactions or external knowledge sources connected to the user.
   * **Authentication:** Requesting and retrieving credentials needed by tools to access external APIs securely.
4. **Identity and Tracking:** Knowing which agent is currently running (agent.name) and uniquely identifying the current request-response cycle (invocation\_id) for logging and debugging.
5. **Tool-Specific Actions:** Enabling specialized operations within tools, such as requesting authentication or searching memory, which require access to the current interaction's details.

The central piece holding all this information together for a single, complete user-request-to-final-response cycle (an **invocation**) is the InvocationContext. However, you typically won't create or manage this object directly. The ADK framework creates it when an invocation starts (e.g., via runner.run\_async) and passes the relevant contextual information implicitly to your agent code, callbacks, and tools.

**[Python](https://google.github.io/adk-docs/context/#python)**

**[Java](https://google.github.io/adk-docs/context/#java)**

# Conceptual Pseudocode: How the framework provides context (Internal Logic)

# runner = Runner(agent=my\_root\_agent, session\_service=..., artifact\_service=...)

# user\_message = types.Content(...)

# session = session\_service.get\_session(...) # Or create new

# --- Inside runner.run\_async(...) ---

# 1. Framework creates the main context for this specific run

# invocation\_context = InvocationContext(

# invocation\_id="unique-id-for-this-run",

# session=session,

# user\_content=user\_message,

# agent=my\_root\_agent, # The starting agent

# session\_service=session\_service,

# artifact\_service=artifact\_service,

# memory\_service=memory\_service,

# # ... other necessary fields ...

# )

#

# 2. Framework calls the agent's run method, passing the context implicitly

# (The agent's method signature will receive it, e.g., runAsyncImpl(InvocationContext invocationContext))

# await my\_root\_agent.run\_async(invocation\_context)

# --- End Internal Logic ---

#

# As a developer, you work with the context objects provided in method arguments.

## The Different types of Context[¶](https://google.github.io/adk-docs/context/#the-different-types-of-context)

While InvocationContext acts as the comprehensive internal container, ADK provides specialized context objects tailored to specific situations. This ensures you have the right tools and permissions for the task at hand without needing to handle the full complexity of the internal context everywhere. Here are the different "flavors" you'll encounter:

1. **InvocationContext**
   * **Where Used:** Received as the ctx argument directly within an agent's core implementation methods (\_run\_async\_impl, \_run\_live\_impl).
   * **Purpose:** Provides access to the *entire* state of the current invocation. This is the most comprehensive context object.
   * **Key Contents:** Direct access to session (including state and events), the current agent instance, invocation\_id, initial user\_content, references to configured services (artifact\_service, memory\_service, session\_service), and fields related to live/streaming modes.
   * **Use Case:** Primarily used when the agent's core logic needs direct access to the overall session or services, though often state and artifact interactions are delegated to callbacks/tools which use their own contexts. Also used to control the invocation itself (e.g., setting ctx.end\_invocation = True).
2. **[Python](https://google.github.io/adk-docs/context/#python_1)**
3. **[Java](https://google.github.io/adk-docs/context/#java_1)**

# Pseudocode: Agent implementation receiving InvocationContext

from google.adk.agents import BaseAgent, InvocationContext

from google.adk.events import Event

from typing import AsyncGenerator

class MyAgent(BaseAgent):

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

# Direct access example

agent\_name = ctx.agent.name

session\_id = ctx.session.id

print(f"Agent {agent\_name} running in session {session\_id} for invocation {ctx.invocation\_id}")

# ... agent logic using ctx ...

yield # ... event ...

1. **ReadonlyContext**
   * **Where Used:** Provided in scenarios where only read access to basic information is needed and mutation is disallowed (e.g., InstructionProvider functions). It's also the base class for other contexts.
   * **Purpose:** Offers a safe, read-only view of fundamental contextual details.
   * **Key Contents:** invocation\_id, agent\_name, and a read-only *view* of the current state.
2. **[Python](https://google.github.io/adk-docs/context/#python_2)**
3. **[Java](https://google.github.io/adk-docs/context/#java_2)**

# Pseudocode: Instruction provider receiving ReadonlyContext

from google.adk.agents import ReadonlyContext

def my\_instruction\_provider(context: ReadonlyContext) -> str:

# Read-only access example

user\_tier = context.state().get("user\_tier", "standard") # Can read state

# context.state['new\_key'] = 'value' # This would typically cause an error or be ineffective

return f"Process the request for a {user\_tier} user."

1. **CallbackContext**
   * **Where Used:** Passed as callback\_context to agent lifecycle callbacks (before\_agent\_callback, after\_agent\_callback) and model interaction callbacks (before\_model\_callback, after\_model\_callback).
   * **Purpose:** Facilitates inspecting and modifying state, interacting with artifacts, and accessing invocation details *specifically within callbacks*.
   * **Key Capabilities (Adds to ReadonlyContext):**
     + **Mutable state Property:** Allows reading *and writing* to session state. Changes made here (callback\_context.state['key'] = value) are tracked and associated with the event generated by the framework after the callback.
     + **Artifact Methods:** load\_artifact(filename) and save\_artifact(filename, part) methods for interacting with the configured artifact\_service.
     + Direct user\_content access.
2. **[Python](https://google.github.io/adk-docs/context/#python_3)**
3. **[Java](https://google.github.io/adk-docs/context/#java_3)**

# Pseudocode: Callback receiving CallbackContext

from google.adk.agents import CallbackContext

from google.adk.models import LlmRequest

from google.genai import types

from typing import Optional

def my\_before\_model\_cb(callback\_context: CallbackContext, request: LlmRequest) -> Optional[types.Content]:

# Read/Write state example

call\_count = callback\_context.state.get("model\_calls", 0)

callback\_context.state["model\_calls"] = call\_count + 1 # Modify state

# Optionally load an artifact

# config\_part = callback\_context.load\_artifact("model\_config.json")

print(f"Preparing model call #{call\_count + 1} for invocation {callback\_context.invocation\_id}")

return None # Allow model call to proceed

1. **ToolContext**
   * **Where Used:** Passed as tool\_context to the functions backing FunctionTools and to tool execution callbacks (before\_tool\_callback, after\_tool\_callback).
   * **Purpose:** Provides everything CallbackContext does, plus specialized methods essential for tool execution, like handling authentication, searching memory, and listing artifacts.
   * **Key Capabilities (Adds to CallbackContext):**
     + **Authentication Methods:** request\_credential(auth\_config) to trigger an auth flow, and get\_auth\_response(auth\_config) to retrieve credentials provided by the user/system.
     + **Artifact Listing:** list\_artifacts() to discover available artifacts in the session.
     + **Memory Search:** search\_memory(query) to query the configured memory\_service.
     + **function\_call\_id Property:** Identifies the specific function call from the LLM that triggered this tool execution, crucial for linking authentication requests or responses back correctly.
     + **actions Property:** Direct access to the EventActions object for this step, allowing the tool to signal state changes, auth requests, etc.
2. **[Python](https://google.github.io/adk-docs/context/#python_4)**
3. **[Java](https://google.github.io/adk-docs/context/#java_4)**

# Pseudocode: Tool function receiving ToolContext

from google.adk.tools import ToolContext

from typing import Dict, Any

# Assume this function is wrapped by a FunctionTool

def search\_external\_api(query: str, tool\_context: ToolContext) -> Dict[str, Any]:

api\_key = tool\_context.state.get("api\_key")

if not api\_key:

# Define required auth config

# auth\_config = AuthConfig(...)

# tool\_context.request\_credential(auth\_config) # Request credentials

# Use the 'actions' property to signal the auth request has been made

# tool\_context.actions.requested\_auth\_configs[tool\_context.function\_call\_id] = auth\_config

return {"status": "Auth Required"}

# Use the API key...

print(f"Tool executing for query '{query}' using API key. Invocation: {tool\_context.invocation\_id}")

# Optionally search memory or list artifacts

# relevant\_docs = tool\_context.search\_memory(f"info related to {query}")

# available\_files = tool\_context.list\_artifacts()

return {"result": f"Data for {query} fetched."}

Understanding these different context objects and when to use them is key to effectively managing state, accessing services, and controlling the flow of your ADK application. The next section will detail common tasks you can perform using these contexts.

## Common Tasks Using Context[¶](https://google.github.io/adk-docs/context/#common-tasks-using-context)

Now that you understand the different context objects, let's focus on how to use them for common tasks when building your agents and tools.

### Accessing Information[¶](https://google.github.io/adk-docs/context/#accessing-information)

You'll frequently need to read information stored within the context.

* **Reading Session State:** Access data saved in previous steps or user/app-level settings. Use dictionary-like access on the state property.
* **[Python](https://google.github.io/adk-docs/context/#python_5)**
* **[Java](https://google.github.io/adk-docs/context/#java_5)**

# Pseudocode: In a Tool function

from google.adk.tools import ToolContext

def my\_tool(tool\_context: ToolContext, \*\*kwargs):

user\_pref = tool\_context.state.get("user\_display\_preference", "default\_mode")

api\_endpoint = tool\_context.state.get("app:api\_endpoint") # Read app-level state

if user\_pref == "dark\_mode":

# ... apply dark mode logic ...

pass

print(f"Using API endpoint: {api\_endpoint}")

# ... rest of tool logic ...

# Pseudocode: In a Callback function

from google.adk.agents import CallbackContext

def my\_callback(callback\_context: CallbackContext, \*\*kwargs):

last\_tool\_result = callback\_context.state.get("temp:last\_api\_result") # Read temporary state

if last\_tool\_result:

print(f"Found temporary result from last tool: {last\_tool\_result}")

# ... callback logic ...

* **Getting Current Identifiers:** Useful for logging or custom logic based on the current operation.
* **[Python](https://google.github.io/adk-docs/context/#python_6)**
* **[Java](https://google.github.io/adk-docs/context/#java_6)**

# Pseudocode: In any context (ToolContext shown)

from google.adk.tools import ToolContext

def log\_tool\_usage(tool\_context: ToolContext, \*\*kwargs):

agent\_name = tool\_context.agent\_nameSystem.out.println("Found temporary result from last tool: " + lastToolResult);

inv\_id = tool\_context.invocation\_id

func\_call\_id = getattr(tool\_context, 'function\_call\_id', 'N/A') # Specific to ToolContext

print(f"Log: Invocation={inv\_id}, Agent={agent\_name}, FunctionCallID={func\_call\_id} - Tool Executed.")

* **Accessing the Initial User Input:** Refer back to the message that started the current invocation.
* **[Python](https://google.github.io/adk-docs/context/#python_7)**
* **[Java](https://google.github.io/adk-docs/context/#java_7)**

# Pseudocode: In a Callback

from google.adk.agents import CallbackContext

def check\_initial\_intent(callback\_context: CallbackContext, \*\*kwargs):

initial\_text = "N/A"

if callback\_context.user\_content and callback\_context.user\_content.parts:

initial\_text = callback\_context.user\_content.parts[0].text or "Non-text input"

print(f"This invocation started with user input: '{initial\_text}'")

# Pseudocode: In an Agent's \_run\_async\_impl

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

# if ctx.user\_content and ctx.user\_content.parts:

# initial\_text = ctx.user\_content.parts[0].text

# print(f"Agent logic remembering initial query: {initial\_text}")

# ...

### Managing Session State[¶](https://google.github.io/adk-docs/context/#managing-session-state)

State is crucial for memory and data flow. When you modify state using CallbackContext or ToolContext, the changes are automatically tracked and persisted by the framework.

* **How it Works:** Writing to callback\_context.state['my\_key'] = my\_value or tool\_context.state['my\_key'] = my\_value adds this change to the EventActions.state\_delta associated with the current step's event. The SessionService then applies these deltas when persisting the event.
* **Passing Data Between Tools:**
* **[Python](https://google.github.io/adk-docs/context/#python_8)**
* **[Java](https://google.github.io/adk-docs/context/#java_8)**

# Pseudocode: Tool 1 - Fetches user ID

from google.adk.tools import ToolContext

import uuid

def get\_user\_profile(tool\_context: ToolContext) -> dict:

user\_id = str(uuid.uuid4()) # Simulate fetching ID

# Save the ID to state for the next tool

tool\_context.state["temp:current\_user\_id"] = user\_id

return {"profile\_status": "ID generated"}

# Pseudocode: Tool 2 - Uses user ID from state

def get\_user\_orders(tool\_context: ToolContext) -> dict:

user\_id = tool\_context.state.get("temp:current\_user\_id")

if not user\_id:

return {"error": "User ID not found in state"}

print(f"Fetching orders for user ID: {user\_id}")

# ... logic to fetch orders using user\_id ...

return {"orders": ["order123", "order456"]}

* **Updating User Preferences:**
* **[Python](https://google.github.io/adk-docs/context/#python_9)**
* **[Java](https://google.github.io/adk-docs/context/#java_9)**

# Pseudocode: Tool or Callback identifies a preference

from google.adk.tools import ToolContext # Or CallbackContext

def set\_user\_preference(tool\_context: ToolContext, preference: str, value: str) -> dict:

# Use 'user:' prefix for user-level state (if using a persistent SessionService)

state\_key = f"user:{preference}"

tool\_context.state[state\_key] = value

print(f"Set user preference '{preference}' to '{value}'")

return {"status": "Preference updated"}

* **State Prefixes:** While basic state is session-specific, prefixes like app: and user: can be used with persistent SessionService implementations (like DatabaseSessionService or VertexAiSessionService) to indicate broader scope (app-wide or user-wide across sessions). temp: can denote data only relevant within the current invocation.

### Working with Artifacts[¶](https://google.github.io/adk-docs/context/#working-with-artifacts)

Use artifacts to handle files or large data blobs associated with the session. Common use case: processing uploaded documents.

* **Document Summarizer Example Flow:**
  1. **Ingest Reference (e.g., in a Setup Tool or Callback):** Save the *path or URI* of the document, not the entire content, as an artifact.
  2. **[Python](https://google.github.io/adk-docs/context/#python_10)**
  3. **[Java](https://google.github.io/adk-docs/context/#java_10)**

# Pseudocode: In a callback or initial tool

from google.adk.agents import CallbackContext # Or ToolContext

from google.genai import types

def save\_document\_reference(context: CallbackContext, file\_path: str) -> None:

# Assume file\_path is something like "gs://my-bucket/docs/report.pdf" or "/local/path/to/report.pdf"

try:

# Create a Part containing the path/URI text

artifact\_part = types.Part(text=file\_path)

version = context.save\_artifact("document\_to\_summarize.txt", artifact\_part)

print(f"Saved document reference '{file\_path}' as artifact version {version}")

# Store the filename in state if needed by other tools

context.state["temp:doc\_artifact\_name"] = "document\_to\_summarize.txt"

except ValueError as e:

print(f"Error saving artifact: {e}") # E.g., Artifact service not configured

except Exception as e:

print(f"Unexpected error saving artifact reference: {e}")

# Example usage:

# save\_document\_reference(callback\_context, "gs://my-bucket/docs/report.pdf")

* 1. **Summarizer Tool:** Load the artifact to get the path/URI, read the actual document content using appropriate libraries, summarize, and return the result.
  2. **[Python](https://google.github.io/adk-docs/context/#python_11)**
  3. **[Java](https://google.github.io/adk-docs/context/#java_11)**

# Pseudocode: In the Summarizer tool function

from google.adk.tools import ToolContext

from google.genai import types

# Assume libraries like google.cloud.storage or built-in open are available

# Assume a 'summarize\_text' function exists

# from my\_summarizer\_lib import summarize\_text

def summarize\_document\_tool(tool\_context: ToolContext) -> dict:

artifact\_name = tool\_context.state.get("temp:doc\_artifact\_name")

if not artifact\_name:

return {"error": "Document artifact name not found in state."}

try:

# 1. Load the artifact part containing the path/URI

artifact\_part = tool\_context.load\_artifact(artifact\_name)

if not artifact\_part or not artifact\_part.text:

return {"error": f"Could not load artifact or artifact has no text path: {artifact\_name}"}

file\_path = artifact\_part.text

print(f"Loaded document reference: {file\_path}")

# 2. Read the actual document content (outside ADK context)

document\_content = ""

if file\_path.startswith("gs://"):

# Example: Use GCS client library to download/read

# from google.cloud import storage

# client = storage.Client()

# blob = storage.Blob.from\_string(file\_path, client=client)

# document\_content = blob.download\_as\_text() # Or bytes depending on format

pass # Replace with actual GCS reading logic

elif file\_path.startswith("/"):

# Example: Use local file system

with open(file\_path, 'r', encoding='utf-8') as f:

document\_content = f.read()

else:

return {"error": f"Unsupported file path scheme: {file\_path}"}

# 3. Summarize the content

if not document\_content:

return {"error": "Failed to read document content."}

# summary = summarize\_text(document\_content) # Call your summarization logic

summary = f"Summary of content from {file\_path}" # Placeholder

return {"summary": summary}

except ValueError as e:

return {"error": f"Artifact service error: {e}"}

except FileNotFoundError:

return {"error": f"Local file not found: {file\_path}"}

# except Exception as e: # Catch specific exceptions for GCS etc.

# return {"error": f"Error reading document {file\_path}: {e}"}

* **Listing Artifacts:** Discover what files are available.
* **[Python](https://google.github.io/adk-docs/context/#python_12)**
* **[Java](https://google.github.io/adk-docs/context/#java_12)**

# Pseudocode: In a tool function

from google.adk.tools import ToolContext

def check\_available\_docs(tool\_context: ToolContext) -> dict:

try:

artifact\_keys = tool\_context.list\_artifacts()

print(f"Available artifacts: {artifact\_keys}")

return {"available\_docs": artifact\_keys}

except ValueError as e:

return {"error": f"Artifact service error: {e}"}

### Handling Tool Authentication[¶](https://google.github.io/adk-docs/context/#handling-tool-authentication)

python_only

Securely manage API keys or other credentials needed by tools.

# Pseudocode: Tool requiring auth

from google.adk.tools import ToolContext

from google.adk.auth import AuthConfig # Assume appropriate AuthConfig is defined

# Define your required auth configuration (e.g., OAuth, API Key)

MY\_API\_AUTH\_CONFIG = AuthConfig(...)

AUTH\_STATE\_KEY = "user:my\_api\_credential" # Key to store retrieved credential

def call\_secure\_api(tool\_context: ToolContext, request\_data: str) -> dict:

# 1. Check if credential already exists in state

credential = tool\_context.state.get(AUTH\_STATE\_KEY)

if not credential:

# 2. If not, request it

print("Credential not found, requesting...")

try:

tool\_context.request\_credential(MY\_API\_AUTH\_CONFIG)

# The framework handles yielding the event. The tool execution stops here for this turn.

return {"status": "Authentication required. Please provide credentials."}

except ValueError as e:

return {"error": f"Auth error: {e}"} # e.g., function\_call\_id missing

except Exception as e:

return {"error": f"Failed to request credential: {e}"}

# 3. If credential exists (might be from a previous turn after request)

# or if this is a subsequent call after auth flow completed externally

try:

# Optionally, re-validate/retrieve if needed, or use directly

# This might retrieve the credential if the external flow just completed

auth\_credential\_obj = tool\_context.get\_auth\_response(MY\_API\_AUTH\_CONFIG)

api\_key = auth\_credential\_obj.api\_key # Or access\_token, etc.

# Store it back in state for future calls within the session

tool\_context.state[AUTH\_STATE\_KEY] = auth\_credential\_obj.model\_dump() # Persist retrieved credential

print(f"Using retrieved credential to call API with data: {request\_data}")

# ... Make the actual API call using api\_key ...

api\_result = f"API result for {request\_data}"

return {"result": api\_result}

except Exception as e:

# Handle errors retrieving/using the credential

print(f"Error using credential: {e}")

# Maybe clear the state key if credential is invalid?

# tool\_context.state[AUTH\_STATE\_KEY] = None

return {"error": "Failed to use credential"}

*Remember: request\_credential pauses the tool and signals the need for authentication. The user/system provides credentials, and on a subsequent call, get\_auth\_response (or checking state again) allows the tool to proceed.* The tool\_context.function\_call\_id is used implicitly by the framework to link the request and response.

### Leveraging Memory[¶](https://google.github.io/adk-docs/context/#leveraging-memory)

python_only

Access relevant information from the past or external sources.

# Pseudocode: Tool using memory search

from google.adk.tools import ToolContext

def find\_related\_info(tool\_context: ToolContext, topic: str) -> dict:

try:

search\_results = tool\_context.search\_memory(f"Information about {topic}")

if search\_results.results:

print(f"Found {len(search\_results.results)} memory results for '{topic}'")

# Process search\_results.results (which are SearchMemoryResponseEntry)

top\_result\_text = search\_results.results[0].text

return {"memory\_snippet": top\_result\_text}

else:

return {"message": "No relevant memories found."}

except ValueError as e:

return {"error": f"Memory service error: {e}"} # e.g., Service not configured

except Exception as e:

return {"error": f"Unexpected error searching memory: {e}"}

### Advanced: Direct InvocationContext Usage[¶](https://google.github.io/adk-docs/context/#advanced-direct-invocationcontext-usage)

python_only

While most interactions happen via CallbackContext or ToolContext, sometimes the agent's core logic (\_run\_async\_impl/\_run\_live\_impl) needs direct access.

# Pseudocode: Inside agent's \_run\_async\_impl

from google.adk.agents import InvocationContext, BaseAgent

from google.adk.events import Event

from typing import AsyncGenerator

class MyControllingAgent(BaseAgent):

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

# Example: Check if a specific service is available

if not ctx.memory\_service:

print("Memory service is not available for this invocation.")

# Potentially change agent behavior

# Example: Early termination based on some condition

if ctx.session.state.get("critical\_error\_flag"):

print("Critical error detected, ending invocation.")

ctx.end\_invocation = True # Signal framework to stop processing

yield Event(author=self.name, invocation\_id=ctx.invocation\_id, content="Stopping due to critical error.")

return # Stop this agent's execution

# ... Normal agent processing ...

yield # ... event ...

Setting ctx.end\_invocation = True is a way to gracefully stop the entire request-response cycle from within the agent or its callbacks/tools (via their respective context objects which also have access to modify the underlying InvocationContext's flag).

## Key Takeaways & Best Practices[¶](https://google.github.io/adk-docs/context/#key-takeaways-best-practices)

* **Use the Right Context:** Always use the most specific context object provided (ToolContext in tools/tool-callbacks, CallbackContext in agent/model-callbacks, ReadonlyContext where applicable). Use the full InvocationContext (ctx) directly in \_run\_async\_impl / \_run\_live\_impl only when necessary.
* **State for Data Flow:** context.state is the primary way to share data, remember preferences, and manage conversational memory *within* an invocation. Use prefixes (app:, user:, temp:) thoughtfully when using persistent storage.
* **Artifacts for Files:** Use context.save\_artifact and context.load\_artifact for managing file references (like paths or URIs) or larger data blobs. Store references, load content on demand.
* **Tracked Changes:** Modifications to state or artifacts made via context methods are automatically linked to the current step's EventActions and handled by the SessionService.
* **Start Simple:** Focus on state and basic artifact usage first. Explore authentication, memory, and advanced InvocationContext fields (like those for live streaming) as your needs become more complex.

By understanding and effectively using these context objects, you can build more sophisticated, stateful, and capable agents with ADK.