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

🚀 Agent Tools

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

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

🤔 Why do Agents need Tools?

The Problem

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

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

In this notebook, you'll:

- ✅ Turn your Python functions into Agent tools
- ✅ Build an Agent and use it **as a tool** in another agent
- ✅ **Build your first multi-tool agent**

- ✅ Explore the different tool types in ADK

‼️ Please Read

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

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

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

For help: Ask questions on the Kaggle Discord server.

📖 Get started with Kaggle Notebooks

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

Here's how to get started:

1. Verify Your Account (Required)

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

You can do this in your Kaggle settings.

2. Make Your Own Copy

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

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

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

3. Run Code Cells

Once you have your copy, you can run code.

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

```
%pip install
```

Run the cells in order from top to bottom.

4. If You Get Stuck

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

For help: Ask questions on the Kaggle Discord server.

⚙️ Section 1: Setup

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

1.1: Install dependencies

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

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

```
pip install google-adk
```

1.2: Configure your Gemini API Key

This notebook uses the Gemini API, which requires an API key.

1. Get your API key

If you don't have one already, create an API key in Google AI Studio.

2. Add the key to Kaggle Secrets

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

1. In the top menu bar of the notebook editor, select `Add-ons` then `Secrets`.
2. Create a new secret with the label `GOOGLE_API_KEY`.

3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to `GOOGLE_API_KEY` is selected so that the secret is attached to the notebook.

3. Authenticate in the notebook

Run the cell below to access the `GOOGLE_API_KEY` you just saved and set it as an environment variable for the notebook to use:

```
import os
from kaggle_secrets import UserSecretsClient

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

1.3: Import ADK components

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

```
from google.genai import types

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

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

1.4: Helper functions

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

```
def show_python_code_and_result(response):
    for i in range(len(response)):
```

```
        # Check if the response contains a valid function call result from the code executor
        if (
            (response[i].content.parts)
            and (response[i].content.parts[0])
            and (response[i].content.parts[0].function_response)
            and (response[i].content.parts[0].function_response.response)
        ):
            response_code = response[i].content.parts[0].function_response.response
            if "result" in response_code and response_code["result"] != "```":
                if "tool_code" in response_code["result"]:
                    print(
                        "Generated Python Code >> ",
                        response_code["result"].replace("tool_code", ""),
                    )
                else:
                    print("Generated Python Response >> ", response_code["result"])


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

1.5: Configure Retry Options

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

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

🤖 Section 2: What are Custom Tools?

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

When to use Custom Tools?

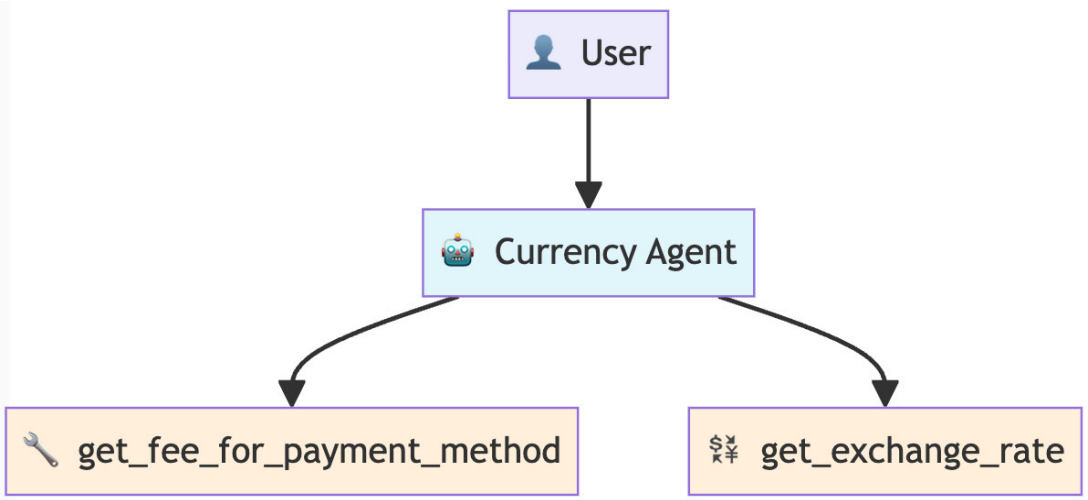
Built-in tools like Google Search are powerful, but **every business has unique requirements** that generic tools can't handle. Custom tools let you implement your specific business logic, connect to your systems, and solve domain-specific problems. ADK provides multiple custom tool types to handle these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

2.1: Building Custom Function Tools

Example: Currency Converter Agent

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

1. **Fee Lookup Tool** - Finds transaction fees for the conversion (mock)
2. **Exchange Rate Tool** - Gets currency conversion rates (mock)
3. **Calculation Step** - Calculates the total conversion cost including the fees



🤔 2.2: How to define a Tool?

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

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

🏆 ADK Best Practices in Action

Notice how our tools follow ADK best practices:

1. Dictionary Returns: Tools return `{"status": "success", "data": ...}` or `{"status": "error", "error_message": ...}`
2. Clear Docstrings: LLMs use docstrings to understand when and how to use tools

3. **Type Hints**: Enable ADK to generate proper schemas (`str`, `dict`, etc.)
4. **Error Handling**: Structured error responses help LLMs handle failures gracefully

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

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

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

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

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

        Returns:
            Dictionary with status and fee information.
            Success: {"status": "success", "fee_percentage": 0.02}
            Error: {"status": "error", "error_message": "Payment method not found"}
        """
        # This simulates looking up a company's internal fee structure.
        fee_database = {
            "platinum credit card": 0.02,  # 2%
            "gold debit card": 0.035,  # 3.5%
            "bank transfer": 0.01,  # 1%
        }

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


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

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

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

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

    Returns:
        Dictionary with status and rate information.
        Success: {"status": "success", "rate": 0.93}
        Error: {"status": "error", "error_message": "Unsupported currency pair"}
    """

    # Static data simulating a live exchange rate API
    # In production, this would call something like: requests.get("api.exchangerates.com")
    rate_database = {
        "usd": {
            "eur": 0.93,  # Euro
            "jpy": 157.50,  # Japanese Yen
            "inr": 83.58,  # Indian Rupee
        }
    }

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

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


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

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

Key Points:

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

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

    For currency conversion requests:
    1. Use `get_fee_for_payment_method()` to find transaction fees
    2. Use `get_exchange_rate()` to get currency conversion rates
    3. Check the "status" field in each tool's response for errors
    4. Calculate the final amount after fees based on the output from `get_fee_for_payment_method` and `get_exchange_rate` methods and
    5. First, state the final converted amount.
        Then, explain how you got that result by showing the intermediate amounts. Your explanation must include: the fee percentage a
        value in the original currency, the amount remaining after the fee, and the exchange rate used for the final conversion.

    If any tool returns status "error", explain the issue to the user clearly.
    """,
    tools=[get_fee_for_payment_method, get_exchange_rate],
)

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

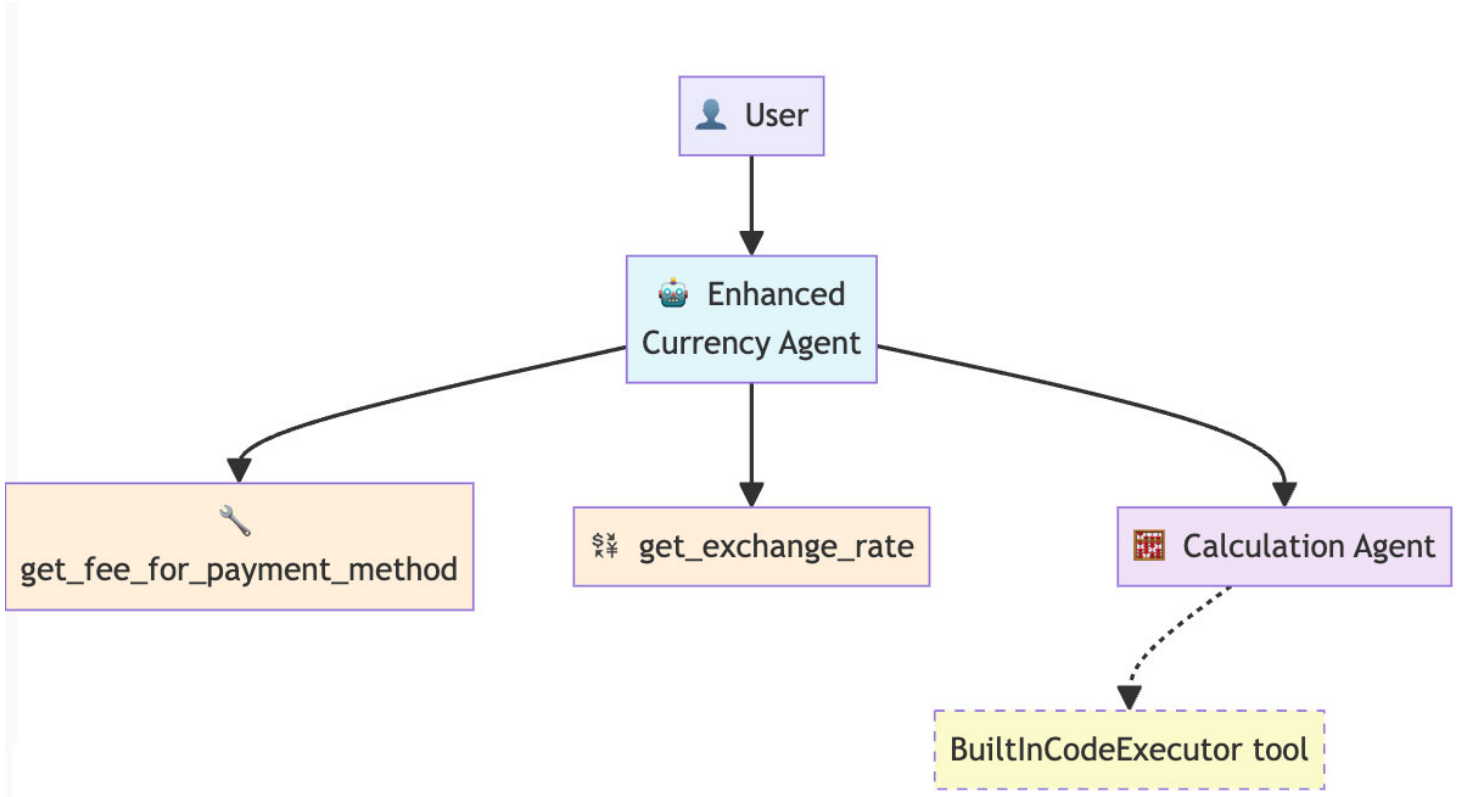
```
# Test the currency agent
currency_runner = InMemoryRunner(agent=currency_agent)
_ = await currency_runner.run_debug(
    "I want to convert 500 US Dollars to Euros using my Platinum Credit Card. How much will I receive?"
)
```

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

Section 3: Improving Agent Reliability with Code

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

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



3.1 Built-in Code Executor

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

Let's create a `calculation_agent` which takes in a Python code and uses the `BuiltInCodeExecutor` to run it.

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

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

         **RULES:**
        1.  Your output MUST be ONLY a Python code block.
        2.  Do NOT write any text before or after the code block.
        3.  The Python code MUST calculate the result.
        4.  The Python code MUST print the final result to stdout.
```

```
    5.  You are PROHIBITED from performing the calculation yourself. Your only job is to generate the code that will perform the calcu

    Failure to follow these rules will result in an error.
       """,
    code_executor=BuiltInCodeExecutor(),  # Use the built-in Code Executor Tool. This gives the agent code execution capabilities
)
```

3.2: Update the Agent's instruction and toolset

We'll do two key actions:

1. **Update the `currency_agent`'s instructions to generate Python code**

- Original: "*Calculate the final amount after fees*" (vague math instructions)
- Enhanced: "*Generate a Python code to calculate the final amount .. and use the `calculation_agent` to run the code and compute final amount*"

2. **Add the `calculation_agent` to the toolset**

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

- Add `AgentTool(agent=calculation_agent)` to the tools list
- The specialist agent appears as a callable tool to the root agent

Let's see this in action:

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

  For any currency conversion request:

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

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

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

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

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

Excellent! Notice what happened:

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

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

```
show_python_code_and_result(response)
```

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

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

Agent Tools (what we're using):

- Agent A calls Agent B as a tool
- Agent B's response goes **back to Agent A**

- Agent A stays in control and continues the conversation
- **Use case**: Delegation for specific tasks (like calculations)

Sub-Agents (different pattern):

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

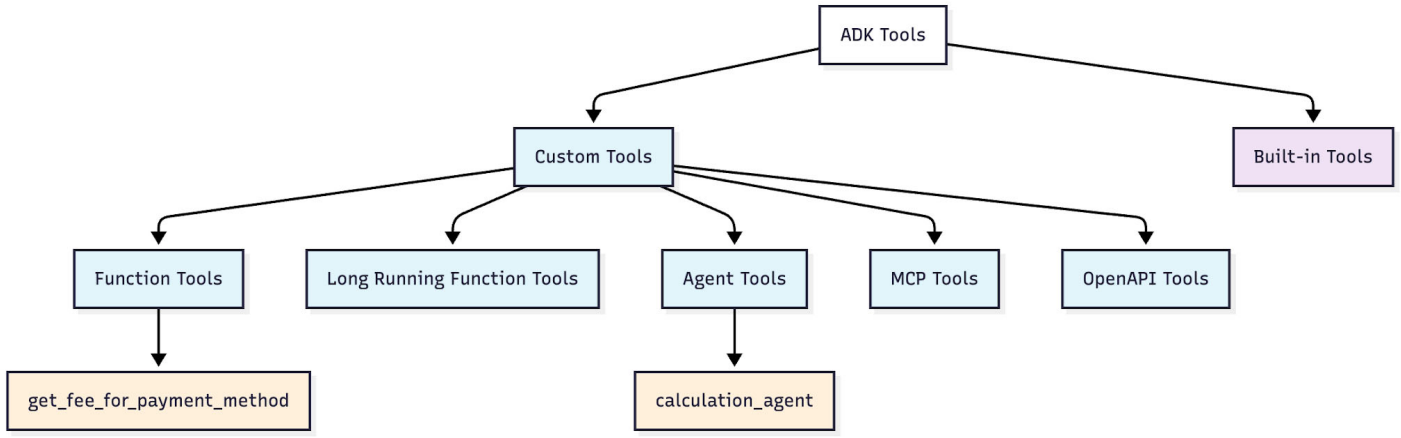
In our currency example: We want the currency agent to get calculation results and continue working with them, so we use **Agent Tools**, not sub-agents.

🧰 Section 4: Complete Guide to ADK Tool Types

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

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

1. Custom Tools



What: Tools you build yourself for specific needs

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

Function Tools ✅ (You've used these!)

- **What**: Python functions converted to agent tools

- **Examples**: `get_fee_for_payment_method`, `get_exchange_rate`
- **Advantage**: Turn any Python function into an agent tool instantly

Long Running Function Tools

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

Agent Tools ✅ (You've used these!)

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

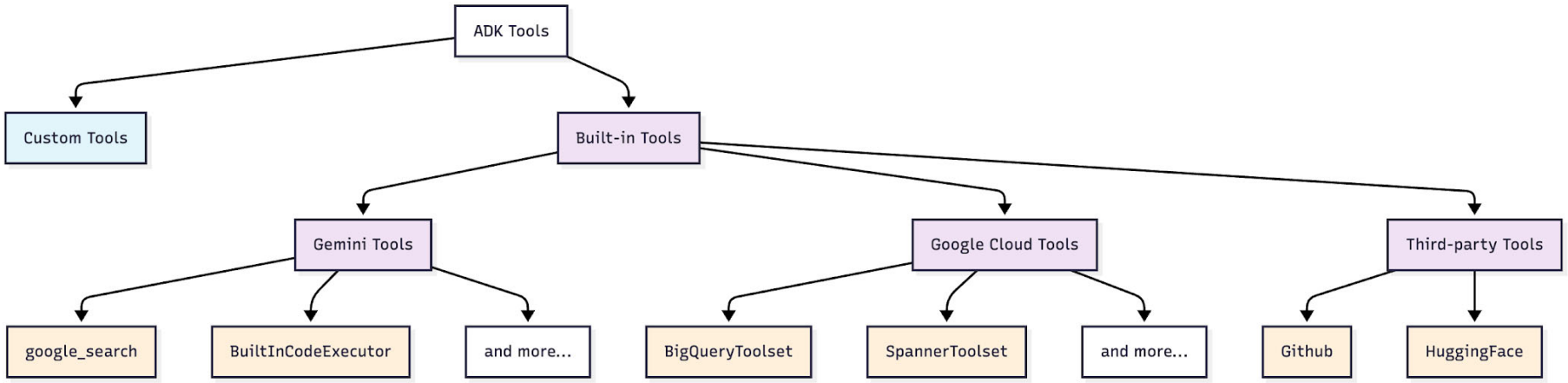
MCP Tools

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

OpenAPI Tools

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

2. Built-in Tools



What: Pre-built tools provided by ADK

Advantage: No development time — use immediately with zero setup

Gemini Tools ✅ (You've used these!)

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

Google Cloud Tools [needs Google Cloud access]

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

Third-party Tools

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

✅ Congratulations!

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

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

ℹ️ **Note: No submission required!**

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

📚 Learn More

Refer to the following documentation to learn more:

- ADK Documentation
- ADK Tools Documentation

- ADK Custom Tools Guide
- ADK Function Tools
- ADK Plugins Overview

🎯 Next Steps

You've built the foundation of agent tool mastery.

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

Authors

Laxmi Harikumar