Lesson 3 - Introduction to Google's ADK - Part II

In this lesson, you will familiarize yourself with Google's Agent Development Kit (ADK) that you will use in the next lessons to build your multi-agent system.

You'll learn:

- how to create and run an agent using ADK (Part I)
- how to create a team of Agents consisting of a root agent and 2 sub-agents (Part II)
- how the team of agents can access a sharable context (Part II)

For each agent, you'll define a tool that allows the agent to interact with the Neo4j database we setup for this course.

💻 **To access the helper.py and neo4j_for_adk.py files:** 1) click on the *"File"* option on the top menu of the notebook and then 2) click on *"Open"*.

3.1. Setup

```
# Import necessary libraries
import os
from google.adk.agents import Agent
from google.adk.models.lite_llm import LiteLlm # For OpenAI support
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner
from google.genai import types # For creating message Content/Parts
from typing import Optional, Dict, Any

# Convenience libraries for working with Neo4j inside of Google ADK
from neo4j_for_adk import graphdb

import warnings
# Ignore all warnings
warnings.filterwarnings("ignore")

import logging
logging.basicConfig(level=logging.CRITICAL)

print("Libraries imported.")
# --- Define Model Constants for easier use ---
MODEL_GPT = "openai/gpt-4o"

llm = LiteLlm(model=MODEL_GPT)

# Test LLM with a direct call
print(llm.llm_client.completion(model=llm.model,
```

```
                                messages=[{"role": "user", "content": "Are you
ready?"}],
                                tools=[]))

print("\nOpenAI is ready for use.")
```

3.2. Set up AgentCaller

Remember that we defined `AgentCaller` in part I as a simple wrapper for interacting with an ADK agent. Additionally we defined a factory function `make_agent_caller` to make instances of `AgentCaller`. Let's import the factory function from the `helper` script.

```
from helper import make_agent_caller
```

3.3. A Simple Multi-Agent Team - Delegation for Greetings & Farewells

Using multiple agents is a common pattern in real-world applications. It allows for better modularity, specialization, and scalability.

You will:

1. Define another simple tool that will handle farewells (`say_goodbye`).
2. Create two new specialized sub-agents: `greeting_agent` and `farewell_agent`.
3. Create a new top-level agent (`cypher_agent_team`) to act as the **root agent**.
4. Configure the root agent with its sub-agents, enabling **automatic delegation**.
5. Test the delegation flow by sending different types of requests to the root agent.

3.3.1 Define Tools for Sub-Agents

```
# Define the hello tool
def say_hello(person_name: str) -> dict:
    """Formats a welcome message to a named person.

    Args:
        person_name (str): the name of the person saying hello

    Returns:
        dict: A dictionary containing the results of the query.
              Includes a 'status' key ('success' or 'error').
              If 'success', includes a 'query_result' key with an array of result
rows.
              If 'error', includes an 'error_message' key.
    """
    return graphdb.send_query("RETURN 'Hello to you, ' + $person_name AS reply",
    {
        "person_name": person_name
    })
# Define the new goodbye tool
```

```
def say_goodbye() -> dict:
    """Provides a simple farewell message to conclude the conversation."""
    return graphdb.send_query("RETURN 'Goodbye from Cypher!' as farewell")
```

3.3.2. Define the Sub-Agents (Greeting & Farewell)

Now, create the `Agent` instances for our specialists. Notice their highly focused `instruction` and, critically, their clear `description`. The `description` is the primary information the *root agent* uses to decide *when* to delegate to these sub-agents.

Best Practice:

- Sub-agent `description` fields should accurately and concisely summarize their specific capability. This is crucial for effective automatic delegation.
- Sub-agent `instruction` fields should be tailored to their limited scope, telling them exactly what to do and *what not* to do (e.g., "Your *only* task is...").

```
# --- Greeting Agent ---
greeting_subagent = Agent(
    model=llm,
    name="greeting_subagent_v1",
    instruction="You are the Greeting Agent. Your ONLY task is to provide a
friendly greeting to the user. "
                "Use the 'say_hello' tool to generate the greeting. "
                "If the user provides their name, make sure to pass it to the
tool. "
                "Do not engage in any other conversation or tasks.",
    description="Handles simple greetings and hellos using the 'say_hello'
tool.", # Crucial for delegation
    tools=[say_hello],
)
print(f"✅ Agent '{greeting_subagent.name}' created.")

# --- Farewell Agent ---
farewell_subagent = Agent(
    # Can use the same or a different model
    model=llm, # Sticking with GPT for this example
    name="farewell_subagent_v1",
    instruction="You are the Farewell Agent. Your ONLY task is to provide a
polite goodbye message. "
                "Use the 'say_goodbye' tool when the user indicates they are
leaving or ending the conversation "
                "(e.g., using words like 'bye', 'goodbye', 'thanks bye', 'see
you'). "
                "Do not perform any other actions.",
```

```
    description="Handles simple farewells and goodbyes using the 'say_goodbye'
tool.", # Crucial for delegation
    tools=[say_goodbye],
)
print(f"✅ Agent '{farewell_subagent.name}' created.")
```

3.3.3. Define the Root Agent with Sub-Agents

The `root_agent` can now manage sub-agents by including them in the `sub_agents` parameter and updating its instructions to explain when to delegate tasks. With this setup, ADK enables automatic delegation: if a user query is better suited to a sub-agent based on its description, the root agent will automatically hand off control to that sub-agent. For effective delegation, clearly specify in the root agent's instructions which sub-agents exist and when to delegate to them.

```
root_agent = Agent(
    name="friendly_agent_team_v1", # Give it a new version name
    model=llm,
    description="The main coordinator agent. Delegates greetings/farewells to
specialists.",
    instruction="""You are the main Agent coordinating a team. Your primary
responsibility is to be friendly.
                You have specialized sub-agents:
                1. 'greeting_agent': Handles simple greetings like 'Hi', 'Hello'.
Delegate to it for these.
                2. 'farewell_agent': Handles simple farewells like 'Bye', 'See
you'. Delegate to it for these.

                Analyze the user's query. If it's a greeting, delegate to
'greeting_agent'.
                If it's a farewell, delegate to 'farewell_agent'.

                For anything else, respond appropriately or state you cannot
handle it.
                """,
    tools=[], # No tools for the root agent
    # Key change: Link the sub-agents here!
    sub_agents=[greeting_subagent, farewell_subagent]
)


print(f"✅ Root Agent '{root_agent.name}' created with sub-agents: {[sa.name for
sa in root_agent.sub_agents]}")
```

3.3.4. Interact with the Agent Team

Now that you've defined our root agent with its specialized sub-agents, let's test the delegation mechanism.

```
from helper import make_agent_caller

root_agent_caller = await make_agent_caller(root_agent)

async def run_team_conversation():
   await root_agent_caller.call("Hello I'm ABK", True)

   await root_agent_caller.call("Thanks, bye!", True)

# Execute the conversation using await
await run_team_conversation()
```

You've now structured your application with multiple collaborating agents. This modular design is fundamental for building more complex and capable agent systems. In the next step, you'll give our agents the ability to remember information across turns using session state.

3.4. Adding Memory and Personalization with Session State

Optional Reading

Currently, agents can delegate tasks but cannot remember past interactions. To enable memory for context-aware behavior, ADK uses **Session State**—a Python dictionary linked to each user session.

> `session.state` is tied to a specific user session identified by `APP_NAME`, `USER_ID`, `SESSION_ID`. It persists information *across multiple conversational turns* within that session.

Session State lets agents and tools store and retrieve information across multiple turns. Tools access this state through the ToolContext object, while agents can automatically save their final responses using the output_key setting.

1. **`ToolContext`** Tools can accept a `ToolContext` object giving access to the session state via `tool_context.state` allowing tools to save information *during* execution.
2. **`output_key`** Agents can have `output_key="your_key"` allowing ADK to save the agent's final textual response into `session.state["your_key"]`.

These constructs allow agents to remember past details, adapt their actions, and personalize responses during a session.

3.4.1. Create State-Aware hello/goodbye Tools

You will create a new version of the hello/goodbye tools. The key feature is accepting `tool_context: ToolContext` which allows them to access `tool_context.state`.

They will write to or read from the `user_name` state variable.

- **Key Concept:** `ToolContext` This object is the bridge allowing your tool logic to interact with the session's context, including reading and writing state variables. ADK injects it automatically if defined as the last parameter of your tool function.
- **Best Practice:** When reading from state, use `dictionary.get('key', default_value)` to handle cases where the key might not exist yet, ensuring your tool doesn't crash.

```
from google.adk.tools.tool_context import ToolContext

def say_hello_stateful(user_name:str, tool_context:ToolContext):
    """Says hello to the user, recording their name into state.

    Args:
        user_name (str): The name of the user.
    """
    tool_context.state["user_name"] = user_name
    print("\ntool_context.state['user_name']:", tool_context.state["user_name"])
    return graphdb.send_query(
        f"RETURN 'Hello to you, ' + $user_name + '.' AS reply",
    {
        "user_name": user_name
    })
def say_goodbye_stateful(tool_context: ToolContext) -> dict:
    """Says goodbye to the user, reading their name from state."""
    user_name = tool_context.state.get("user_name", "stranger")
    print("\ntool_context.state['user_name']:", user_name)
    return graphdb.send_query("RETURN 'Goodbye, ' + $user_name + ', nice to chat
with you!' AS reply",
    {
        "user_name": user_name
    })


print("✅ State-aware 'say_hello_stateful' and 'say_goodbye_stateful' tools
defined.")
```

3.4.2. Redefine Sub-Agents and Update Root Agent

To ensure this step is self-contained and builds correctly, you first redefine the `greeting_agent` and `farewell_agent` exactly as they were in Step 3.3. Then, you define the new root agent (`root_agent_stateful`):

- It uses the new `say_hello_stateful` and `say_goodbye_stateful` tools.
- It includes the greeting and farewell sub-agents for delegation.

```
# define a stateful greeting agent. the only difference is that this agent will
use the stateful say_hello_stateful tool
greeting_agent_stateful = Agent(
   model=llm,
   name="greeting_agent_stateful_v1",
   instruction="You are the Greeting Agent. Your ONLY task is to provide a
friendly greeting using the 'say_hello' tool. Do nothing else.",
   description="Handles simple greetings and hellos using the
'say_hello_stateful' tool.",
   tools=[say_hello_stateful],
)
print(f"✅ Agent '{greeting_agent_stateful.name}' redefined.")

farewell_agent_stateful = Agent(
   model=llm,
   name="farewell_agent_stateful_v1",
   instruction="You are the Farewell Agent. Your ONLY task is to provide a
polite goodbye message using the 'say_goodbye_stateful' tool. Do not perform any
other actions.",
   description="Handles simple farewells and goodbyes using the
'say_goodbye_stateful' tool.",
   tools=[say_goodbye_stateful],
)
print(f"✅ Agent '{farewell_agent_stateful.name}' redefined.")
root_agent_stateful = Agent(
   name="friendly_team_stateful", # New version name
   model=llm,
   description="The main coordinator agent. Delegates greetings/farewells to
specialists.",
   instruction="""You are the main Agent coordinating a team. Your primary
responsibility is to be friendly.

               You have specialized sub-agents:
               1. 'greeting_agent_stateful': Handles simple greetings like 'Hi',
'Hello'. Delegate to it for these.
               2. 'farewell_agent_stateful': Handles simple farewells like
'Bye', 'See you'. Delegate to it for these.

               Analyze the user's query. If it's a greeting, delegate to
'greeting_agent_stateful'. If it's a farewell, delegate to
'farewell_agent_stateful'.

               For anything else, respond appropriately or state you cannot
handle it.
               """,
```

```
        tools=[], # Still no tools for root
        sub_agents=[greeting_agent_stateful, farewell_agent_stateful], # Include
sub-agents
    )

print(f"✅ Root Agent '{root_agent_stateful.name}' created using agents with
stateful tools.")
```

3.4.3. Interact and Test State Flow

Now, you can initialize a new `AgentCaller`. This time, you'll provide an initial state.

Then you can execute a conversation designed to test the state interactions using the `root_agent_stateful` root agent.

```
root_stateful_caller = await make_agent_caller(root_agent_stateful)

session = await root_stateful_caller.get_session()

print(f"Initial State: {session.state}")
```

Now, you can define a conversation, run it, then examine the final session state.

```
async def run_stateful_conversation():
    await root_stateful_caller.call("Hello, I'm ABK!")

    await root_stateful_caller.call("Thanks, bye!")

# Execute the conversation using await in an async context (like Colab/Jupyter)
await run_stateful_conversation()

session = await root_stateful_caller.get_session()

print(f"\nFinal State: {session.state}")
```

3.5. Finally, An Interactive Conversation

Now, let's make this interactive so you can ask your own questions! Run the cell below. It will prompt you to enter your queries directly.

Note: In the video, we re-run the first cell in 3.4.3 to start with a fresh state (no stored name). You might get different results from the video.

```
async def run_interactive_conversation():
    while True:
        user_query = input("Ask me something (or type 'exit' to quit): ")
        if user_query.lower() == 'exit':
            break
        response = await root_stateful_caller.call(user_query)
        print(f"Response: {response}")
```

```
# Execute the interactive conversation
await run_interactive_conversation()
```

In the previous lesson, you learned how to create an agent using Google ADK. Now we're going to create multiple agents. We're going to have a root agent and the two sub-agents. We'll build on what we did previously where we had the say hello agent. We'll introduce a new agent that can say goodbye. Then we'll have another agent that puts them together in one multi-agent system. As usual, we'll go ahead and start with importing the libraries that we need. And then of course, we'll go ahead and set up the LLM we're going to use. Again, this is Open AI. We'll define the LLM and we'll also go ahead and do a quick sanity check by passing it a message, make sure that it's ready. Always fun to see what Open AI might say on any given day. Usually with friendly messages, it's pretty good about saying friendly things back. Next, you can set up the agent caller. We're going to use the same class that we used before in lesson three, part one. Rather than defining all that again, we're just going to import what we need. If you remember from part one, we had the helper function, the make agent caller, that's going to create the agent caller class with all that it needs to run. Okay, with everything ready, we can now begin to create the multi-agent system. You're going to create multiple specialized agents, each designed for a specific capability. You'll create one agent in charge of handling greetings, that's the say hello agent we had before. The new agent is going to take charge of saying goodbye. To combine those, you're going to have what's called a root agent, and you'll hear terms like orchestrator or coordinator, top-level agent. It all really means having an agent that's going to wrap up two other agents and then manage their execution. That root agent is going to receive the initial user request, the hello or whatever the user decides to say, and then decide how to process that either itself or by what's called delegating, handing off to one of the sub-agents. So the root agent, this top-level orchestrator, is going to be delegating both the hello and the bye, hopefully to the sub-agents that specialize in those tasks. Okay, let's start to assemble our multi-agent system. Again, we're going to have the say hello agent that we had before, and then add a say goodbye agent, and then we'll combine them together. First, we'll define the tools for our sub-agents. The first tool is the same say hello tool that we had from the previous lesson. This is the hello world function, takes in a person's name, creates a hello world message as a response. To that, we're going to include a say goodbye function. Now, this tool is going to be even simpler than the say hello because it doesn't accept any arguments whatsoever. It has a constant reply, "Goodbye from Cypher," no matter what you want to do, this is what you're going to get as a response saying goodbye to the user. With those two tools defined, we can now define some agents that can use those tools. Now we're going to have a special agent for greeting and a special agent for farewell, but you'll see they're pretty straightforward. Now, as we're defining these sub-agents, I really want to emphasize this fact that is here and how important it is. You have really good descriptions of each of these agents and also the instructions you're handing to them. You're going to spend most of your time, once you set up any kind of multi-agent system, most of your time is going to be spent actually optimizing these instructions, this classic prompt engineering, and also this description. To repeat what I said in the previous lesson, the description is what allows other agents to know what does this agent do and when should I use this agent? That's called delegation. And the instructions are for the agent itself to understand what is its purpose, what is it trying to do, and what tools does it have available and when to use them. So this is our greeting sub-agent. It's the say hello agent that has access to only the say hello tool. And then we'll add our farewell agent. The farewell agent is pretty straightforward. It's similar to the hello agent. You can look through the

description of what the instructions are. And these instructions are actually useful to take a closer look at. It seems obvious, okay, your job as the farewell agent is to respond whenever a user says goodbye in some kind of way. Well, people say goodbye in lots of different ways. So this is a very small example of how to do some few-shot learning within the agent instructions. So here we have in parentheses some examples like when a user uses words like bye, goodbye, thanks bye, or see you, those are all ways of saying goodbye. Now, most LLMs don't need this extra bit of examples, but this is a good practice to have overall to think, how can I help the LLM understand what this agent's purpose is and actually when to execute and what to do in its role. And of course, we're going to hand it the say goodbye tool that we defined earlier. Great. We now have two sub-agents ready to go. Now we need to combine them. We're going to do that by defining the root agent, which is going to understand both of these sub-agents and when to delegate a task to them. It's similar to doing tool calling, but the agent knows that it's calling another agent so the entire conversation history gets passed along and actually control of the conversation gets passed along to the sub-agent. So it's a little bit different than tool call. The same idea is the current workload is going to go from the agent who's in charge, here the root agent, to one of the sub-agents. So this is similar to tool calling, but let's take a careful look at how agent delegation works here. We know that the agents themselves are described with what their role is, like the say hello agent or the say goodbye agent both have some understanding about what it is they're supposed to do, and their description is described to other agents what it is their job is. We're going to double down on that in the instructions that we give to the root agent. The root agent is going to be told, your job is to coordinate a team of sub-agents. So you can see that in the instructions here, and that its primary goal is to be friendly. In order to be friendly, it has actually two specialized sub-agents. So we describe what those are so that the agent understands when it actually should use the sub-agents. This is like describing when you should use particular tools. The greeting agent should be used for handling simple greetings like hello or hi, and of course the farewell agent should be used for saying goodbye, whether that's bye, see you, or anything else. And we're explicit here about saying when those messages are received from the user, go ahead and delegate to the sub-agents to actually respond to the user rather than responding directly as the coordinator. So if this works well, the coordinator shouldn't do anything other than answer general questions, and when you say hello or goodbye, it should delegate to the sub-agents for actually executing or responding or generating a response to the user. This top-level coordinator doesn't have any tools to use whatsoever. It can only chat or delegate to the sub-agents. We're also going to hand it the list of sub-agents here in the sub-agents key. So the sub-agents are a greeting sub-agent and a farewell sub-agent. Now that we have a team of agents all put together into a multi-agent system, let's go ahead and interact with them. This is similar to what we did in part one of lesson three where we had an asynchronous function that managed a conversation. So here the conversation is going to be just what we did before, "Hello, I'm ADK," and then later, rather than saying I'm excited, this transcript is going to say, "Thanks, bye." This is going to be the second user message. We've also added this "true" here. So if you look at the call to this agent caller, the second argument that's passed in is whether or not to be verbose in responses. We're going to turn on verbosity here so you can see all of what's going on behind the scenes, so you can understand both the user message coming in, see the delegation, see the tool calls, and also see the responses. It's a lot to walk through, but it's worth walking through at least once to get familiar with what you can see when you're debugging an agent interaction. All right. Let's start this up. This is going to be enough to get the user's message. Okay. And you can see that the initial event comes from the friendly agent team. So this is the top-level coordinator. So the team coordinator that's going to call the sub-agents. And you can see the first

action that it takes is that it wants to transfer to an agent. So transfer or delegating to a sub-agent. And it's going to transfer it to the greeting sub-agent. So that's perfect. In response to "Hello, I'm ADK," the top-level coordinator realized, I've got an agent who's really good at saying hello. Let me transfer control over to that agent. So the transfer response doesn't have any value whatsoever, it's just a none. But what you should see next is that the greeting sub-agent takes control. So here's another event, but now from the greeting sub-agent. And it's going to look at the transcript and see the user's message and realize that, okay, it's now my job to actually do something about this. And the greeting sub-agent is going to respond by making a tool call. You can see this function call here, and the name of the function that it wants to call is called say hello, and it's going to pass in some arguments where the one argument that's passing in is the person's name is ADK. So the agent has realized that from the statement here, of course, "Hello, I'm ADK," is able to extract that ADK must be the person's name. And so it's going to make a function call to the tool passing in that name. And then you can receive the response. Here's the function response, and the response is, "Hello to you, ADK." Perfect. Now, the greeting sub-agent is done. It goes ahead, and you can see that the final message is true. So that's the flag that says I'm done processing this message. We can go and terminate this particular event loop. And the final agent response ends up being the response that came back from the tool, "Hello to you, ADK." We send in the second query event or message from the user, and the user says, "Thanks, bye." And again, the top-level agent is going to go ahead. Oh, actually, no, it's still in control of the greeting sub-agent. So the greeting sub-agent itself is going to realize, okay, this is not something I should use. Please transfer this to the farewell sub-agent. So it's aware of the other agents that are available. Transfers to the farewell sub-agent. And that is the end of that function response. Here is the new message from the farewell sub-agent itself. It again is going to look at the transcript now that it's been given control of the current conversation. The first thing it's going to do is realize, okay, the user said goodbye. Let me go ahead and call the say goodbye tool to figure out how to respond to them. So here's the function call to say goodbye. And of course, it takes no arguments as expected. The response to it is the constant that you saw earlier in the tool definition. It's going to get a farewell that is "Goodbye from Cypher." And that again is the final response for this event loop. So that is going to be the final agent response, "Goodbye from Cypher." Okay, that was a lot to walk through. It's worth taking the time to do that for anything else we do in these notebooks. You can really see all the interactions about tool calls, agent delegation, and also the responses and the changes in state overall. So it's really useful to do. We're not going to do that for every notebook. If you'd like to, on your own time, please do take the time to really understand what's happening behind the scenes. Because what's happening behind the scenes is going to affect, of course, how you do the overall agent definitions and also the agent orchestration. You're going to take one more step in this multi-agent system. We have an execution environment for agents. We've got a team of multiple agents that can work together. They can delegate tasks to each other. And each of those agents also has tools that they can use. The last important component for all agent systems is having memory. Memory is just internal state happening for a particular session or across any agents that are involved in that session, and of course the user themselves. What is session state really? Session state within Google ADK, by default, is really just a dictionary that's available that you can update keys within that dictionary. And when you update values of those keys, Google ADK tracks those changes. It tracks deltas to the state, basically, and updates the overall session state to keep it consistent across all agents, whether they're running in parallel or running sequentially, because it's an asynchronous system at its heart, and manages all the state updates. Agents have a few different ways of interacting with this state. One way, the way that I prefer and the primary method for interacting with state, is through the tool context. So

whenever a tool is called, there's an extra parameter that's available that is the context within which that tool has been called. Given that tool context, the tool itself now has access to the current state or the current memory of the agent, and it can use that memory for actually making either different decisions, creating different output, whatever is appropriate for the memory. The other way that agents can interact with state is by using an output key. So rather than using a tool call for updating some memory or accessing memory, you can take the actual output from an agent, that final response, and rather than just turning that back as the response from the agent, you can save that into state by defining an output key. So when you see this final true, the message related to the final true here, to be the text "Goodbye from Cypher," could get used for updating state itself. That's very convenient sometimes, and we'll do a little bit of that later on in some of these notebooks. Okay, in this next step, we're going to set up some memory. We always have the in-memory session that we're using right now. So the memory is saved just within RAM, that's not being persisted to a database. Of course, that's the best thing to do for a production system. But for convenience, having in-memory state is perfectly fine. We're going to update both of our tools to actually take advantage of using that in-memory state. So both for say hello and say goodbye, we're going to update those tools to take advantage of the tool context to update session state or memory. You should do this one at a time and take a careful look at the differences here. You'll notice that we're importing tool context from Google ADK. Tool context is the class that takes care of keeping track of the session state and in addition to other things that is useful for tools to have access to. So we're going to update the say hello function to be say hello stateful. It still accepts an argument for the user's name, but it now has this extra argument called tool context. It's the tool context that's going to let us get access to the tool context passed in for the session by the execution environment. So in the tool context, there's this state dictionary, and we're just going to update the username key, and the name that was passed into the function, we're going to use for updating the session memory or the session state for the username key. So this is pretty straightforward, username, username. Now we have a dictionary that is shared across all agents and across all tools that will have that value that was passed into this function. This is a way of sort of keeping track of things by natural use of functions and also then preserving them for later use. For debugging purposes, when this tool is called, we'll go ahead and print out the update to the session state, particularly to the username key. And then, of course, we'll do what we had done in the previous say hello. We'll send a query off to Neo4j using a query parameter that is the username. So let's go ahead and define that. Now that you've defined that, you can define the same thing for the say goodbye. And again, as you're defining this, this is just an update to the say goodbye that we had before. Now, the difference is that this say goodbye, even though it doesn't take an argument for anything related to a person, has no custom arguments here, it does get this tool context passed in. I should mention that this is something that Google ADK does automatically where it sees a tool where the last argument to the tool is a tool context, it automatically injects that to the tool call. So when say goodbye stateful is called here, it's going to get a tool context but nothing else. But because say hello stateful has set the state, the username into the state, say goodbye can actually access the username. So now we've remembered what the user's name is. We can access the state for the state key username and then have a default value in case the user didn't identify themselves. But we can get it from the dictionary, the username or default, assign that to the username, then use that for actually generating a goodbye statement. You can then define a new stateful agent for farewell, and again, that's going to be the same as we had as the previous farewell agent, but now it's going to call the say goodbye stateful tool. As before, you can combine those into a multi-agent system by having a root-level agent or a coordinator agent that's going to use both of those as sub-agents. And this is exactly the same as the other root agent

that we had, but now it's just using all the sub-agents that have ultimately, down at their core, these tools that are stateful-based. You now have a multi-agent system that takes advantage of memory. You can give it a try. We will use our friend, the make agent caller, passing in that new agent stateful, that is the top-level coordinator. We're going to call her for that. And to see the change in the session, we'll go ahead and get the session directly from the caller. Remember there was a utility function for getting that if you look back at the agent caller class. We'll go ahead and get the session and then based on that session, we'll print out the initial state when this is created. And we should see that there's nothing in the current state. As expected, the initial state is empty. You can now define a conversation, the same conversation we've been having. Say hello, say goodbye, and let's see what happens. Actually, I'm going to take out these verbose statements right now. You can decide to keep those in or add them if you'd like. Run it either way. But just to keep the output to a minimum, we'll go ahead and run this without the debugging statements. And what's really important here is that we saw that the initial state was empty, and we expect that once these calls are done, the initial state turns into a final state. We're going to get it again, and that the final state should actually have the username as defined by the call to say hello. Awesome. So our utility functions printed out this update to the username here. And we also see when we get the session and then get the state from that session, that the final state includes the username, ADK. Fantastic. Everything's working as expected. Now, if you'd like to, you can actually run this in an interactive environment, even within this notebook. You can set up a small helper function that's going to have a loop for getting messages from the user and then just calling our caller passing in that message. This will run as long as the user doesn't pass in a message that says "exit." When you set that up and then run it, you'll see a little box appear. So when you run the interactive session, you can just type in a message and just see what happens. I'll start by not revealing my name, see what happens. Okay, that's kind of funny. So the LLM decided that my name is "there," based on this message. That's not really great. Let's correct it. Let's say... And you can see just by passing in my name is ABK, that the tool context actually updated with ABK. So there was a tool call, so to say hello ABK. And because of that, the LLM correctly identified my name is ABK, saved that into memory. This is fantastic. If I say goodbye again, we should now get a nice message by ABK as before. Give this a try, interact with it, see what you can come up with. And of course, if you'd like to, change any of the code and see what the impact is of that. Okay, you've gotten all the way to the end of lesson three. You've created a basic agent that is a simulation that says hello, and then you created a multi-agent system with two sub-agents specialized in saying hello or saying goodbye and a root agent that coordinates their interactions with the user. With this in hand, you're ready to continue to actually doing agentic knowledge graph construction.