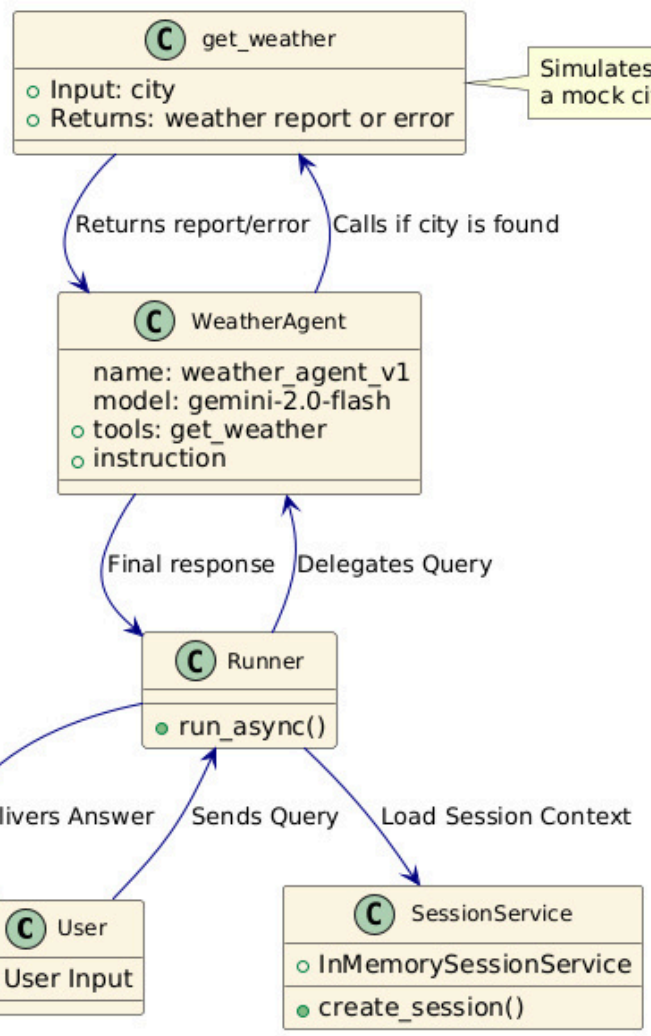
Analytics Vidhya

Building a Weather Bot Team with Google ADK



What is Google ADK?

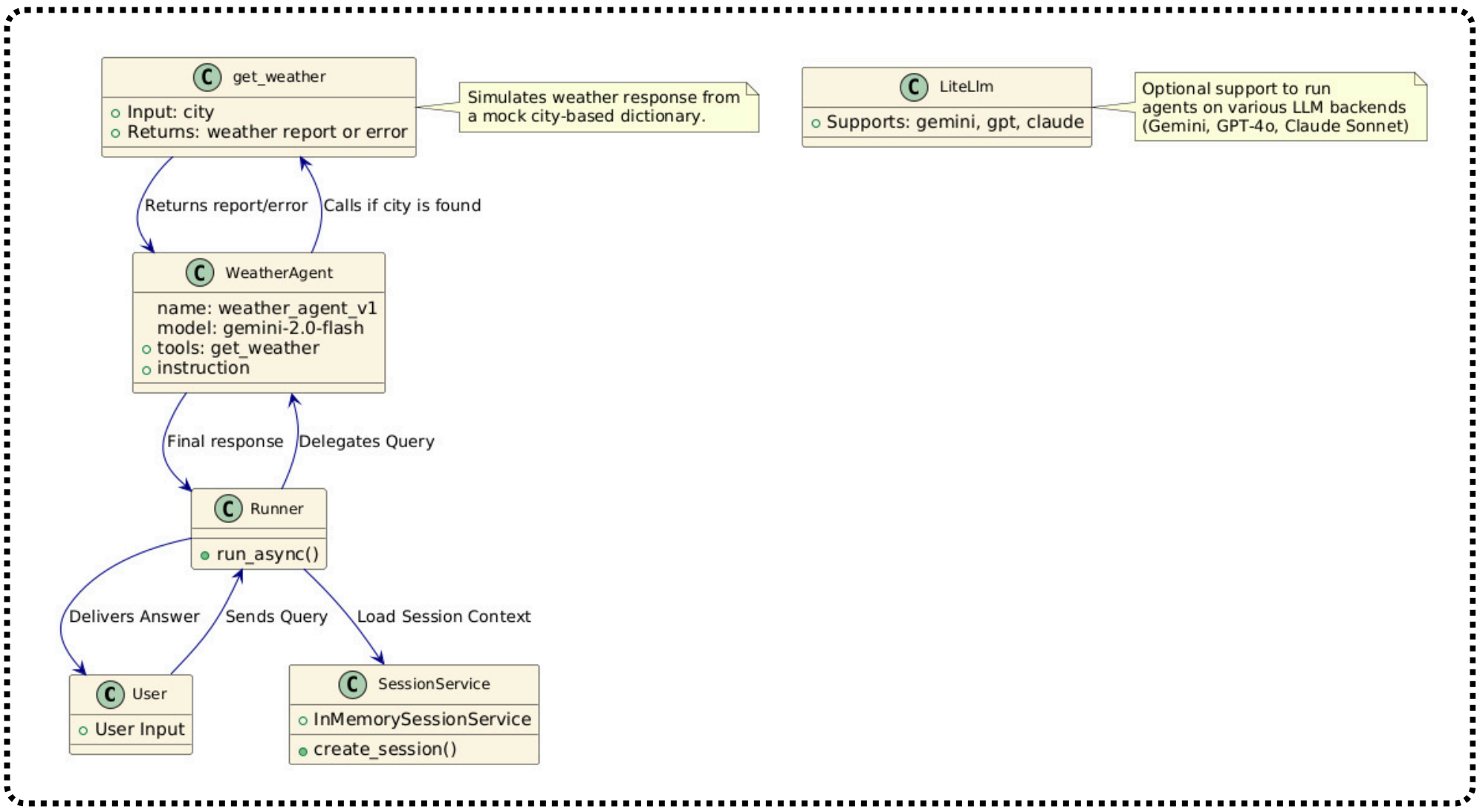
Google ADK (Agent Development Kit) is a framework for building AI-powered agents that can **automate tasks**, **interact with users,** and **integrate with Google's ecosystem** (e.g., Workspace, Cloud, Assistant). It provides tools to create context-aware, multimodal agents using LLMs like Gemini and other Google AI technologies.

- **Pre-built Templates:** Ready-to-use blueprints for common agent types (e.g., customer support, data analysis).
- **Multimodal Capabilities:** Agents can process text, voice, images, and structured data.
- **Google Ecosystem Integration:** Connect with Gmail, Docs, Calendar, Cloud APIs, etc.
- **LLM Orchestration:** Leverage Gemini or fine-tune models for domain-specific tasks.
- **Deployment Flexibility:** Deploy agents as APIs, chatbots, or embedded workflows.

Building a Weather Bot Team

This hands-on project aims to demonstrate how to:

- Design a modular multi-agent system using **Google's Agent Development Kit (ADK)**.
- Integrate **multiple language models** (e.g., Gemini, GPT, Claude) for task specialization.
- Implement intelligent **task delegation** across agents.
- Manage session memory for **contextual continuity**.
- Apply **safety mechanisms** through structured callbacks.



Step 1: Set-up your Environment & Install ADK

```
# Create a virtual environment

python -m venv .venv

# Activate the environment

# macOS/Linux:

source .venv/bin/activate

# Windows CMD:

.venv\Scripts\activate.bat

# Windows PowerShell:

.venv\Scripts\Activate.ps1

pip install google-adk
```

1. Create & Activate Virtual Environment
   - Isolate dependencies for clean project management
   - Activate using OS-specific commands
2. Install Google ADK
   - Core framework for agent development

Ensures dependency control and reproducibility

Step 2: Obtain your API Keys

- Get from Google AI Studio: https://aistudio.google.com/app/apikey
- Get from OpenAI Platform: https://platform.openai.com/api-keys
- Get from Anthropic Console: https://console.anthropic.com/settings/keys

Step 3: Define Tools

```python
def get_weather(city: str) -> dict:
    """Retrieves the current weather report for a specified city.


    Args:
        city (str): The name of the city (e.g., "Mumbai","Chennai","Delhi").


    Returns:
        dict: A dictionary containing the weather information.
              Includes a 'status' key ('success' or 'error').
              If 'success', includes a 'report' key with weather details.
              If 'error', includes an 'error_message' key.
    """
    # Best Practice: Log tool execution for easier debugging
    print(f"--- Tool: get_weather called for city: {city} ---")


    city_normalized = city.lower().replace(" ", "")  # Basic input normalization


    mock_weather_db = {
        "delhi": {"status": "success", "report": "The weather in Delhi is sunny with a temperature of
35°C."},
        "mumbai": {"status": "success", "report": "It's humid in Mumbai with a temperature of 30°C."},
        "bangalore": {"status": "success", "report": "Bangalore is experiencing light showers and a
temperature of 22°C."},
        "kolkata": {"status": "success", "report": "Kolkata is partly cloudy with a temperature of
29°C."},
        "chennai": {"status": "success", "report": "It's hot and humid in Chennai with a temperature of
33°C."},
    }


    if city_normalized in mock_weather_db:
        return mock_weather_db[city_normalized]
    else:
        return {"status": "error", "error_message": f"Sorry, I don't have weather information for
'{city}'."}


# Example usage
print(get_weather("Mumbai"))
```

Tools extend agents beyond text generation, enabling real actions like data fetching or calculations. We'll begin with a mock weather tool to focus on agent architecture, then replace it with live APIs later.

Step 4: Defining the Agent

In ADK, an Agent is the core component that manages the conversation flow, connecting the user, the LLM, and the tools it can use.

```python
AGENT_MODEL=model
weather_agent=Agent(
    name="weather_agent_v1",
    model=AGENT_MODEL,
    description="Provides weather information for specific cities.",
    instruction="You are a helpful weather assistant. Your primary goal is to provide current weather
reports. "
                "When the user asks for the weather in a specific city, "
                "you MUST use the 'get_weather' tool to find the information. "
                "Analyze the tool's response: if the status is 'error', inform the user politely about
the error message. "
                "If the status is 'success', present the weather 'report' clearly and concisely to the
user. "
                "Only use the tool when a city is mentioned for a weather request.",
    tools=[get_weather],


)

print(f"Agent '{weather_agent.name}' created using model '{AGENT_MODEL}'.")
```

Step 5: Set up Runner and Session Service

```python
# @title Setup Session Service and Runner
# ---Session Management ---
# Key Concept: SessionService stores conversation history & state.
# InMemorySessionService is a simple, non-persistent storage for this tutorial.


session_service=InMemorySessionService()


# Define constants for identifying the interaction context
APP_NAME="weathertutorial_app"
USER_ID="user_1"
SESSION_ID="session_001"


# Create the specific session where the conversation will happen
session=session_service.create_session(
    app_name=APP_NAME,
    user_id=USER_ID,
    session_id=SESSION_ID,
)


print(f"Session created: App='{APP_NAME}', User='{USER_ID}',
Session='{SESSION_ID}'")

# ---Runner ---
# Key Concept: Runner orchestrates the agent execution loop.


runner=Runner(
    agent=weather_agent,
    app_name=APP_NAME,
    session_service=session_service
)


print(f"Runner created for agent '{runner.agent.name}'.")
```

SessionService: Tracks conversation history & session state

- Basic version: **InMemorySessionService** (volatile storage for testing)

Runner: Orchestrates the interaction flow:

- Routes user input
- Manages LLM/tool execution
- Updates session data
- Generates real-time interaction logs

(Temporary memory storage can be replaced with persistent solutions later)

Step 6: Interact with the Agent

We'll use ADK's asynchronous Runner for non-blocking LLM/tool interactions. The call_agent_async helper function will:

- Format user queries into ADK's Content structure
- Process via runner.run_async()
- Stream and monitor events (tool calls, responses)
- Extract and display the final response
- This ensures smooth operation during potentially long-running operations.

```
# @title Run the Initial Conversation


# # We need an async function to await our interaction helper
# async def run_conversation():
#     await call_agent_async("What is the weather like in Mumbai")
#     await call_agent_async("How about Delhi?") # Expecting the tool's error
#messagawait call_agent_async("Tell me the weather in CHennai")


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

Output

```
>>> User Query: What is the weather like in Mumbai
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call']
--- Tool: get_weather called for city: Mumbai ---
<<< Agent Response: It's humid in Mumbai with a temperature of 30°C.


>>> User Query: How about Delhi?
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call']
--- Tool: get_weather called for city: Delhi ---
<<< Agent Response: The weather in Delhi is sunny with a temperature of 35°C.


>>> User Query: Tell me the weather in CHennai
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call']
--- Tool: get_weather called for city: Chennai ---
<<< Agent Response: It's hot and humid in Chennai with a temperature of 33°C.
```

To know more, checkout this article

How to Use Google ADK for Building Agents?

Create advanced conversational agents with Google ADK. This guide covers everything you need to kn...

Analytics Vidhya / Apr 24