AI VOID PROJECT LAB

Building Persistent ADK AI Agents

Master building production-ready long-running AI agents using ADK. Learn architectural design, implementation phases, and robust strategies for state and context persistence.

AI VOID

2026-05-23 15:48 IST

Contents

01	Setting Up Your ADK Agent Development Environment	3
02	Building a Basic, Stateless ADK Agent	16
03	Implementing Persistent Agent State with External Storage	28
04	Designing for Context Preservation and Resume Capabilities	51
05	Enhancing Agent Intelligence with Tools and Multi-Step Workflows	72
06	Containerizing Your ADK Agent for Portability and Scalability	91
07	Robust Testing for Long-Running Agent Workflows	106
08	Deploying and Monitoring Your Production ADK Agent on Google Cloud	124
09	Building Persistent AI Agents with Google ADK: Pause, Resume, Recover	144

CHAPTER 01

Setting Up Your ADK Agent Development Environment

Building production-ready AI agents that can maintain conversational context and internal state across multiple sessions is a complex but crucial task. This chapter lays the essential groundwork by guiding you through setting up a robust local development environment and configuring your Google Cloud Project. By the end, you'll have a fully equipped workspace, ready to develop, test, and interact with your first basic agent. This foundational setup is critical for efficiently tackling the complexities of state persistence, reliable operation, and eventual deployment in subsequent chapters.

Project Overview

Our goal throughout this project is to build a long-running, stateful AI agent using Google's Agent Development Kit (ADK). This agent will be capable of:

- **Persisting State:** Saving its internal memory and conversational context to an external durable store.
- **Pause and Resume:** Stopping execution and later resuming precisely where it left off, without losing information.
- **Error Recovery:** Handling interruptions gracefully and recovering its state to continue operations.
- **Production Deployment:** Being containerized and deployed to Google Cloud for scalability and reliability.

This first chapter focuses on the initial setup, ensuring all tools and cloud resources are correctly configured before we write any agent logic.

Tech Stack

This project leverages a modern, cloud-native tech stack designed for scalability and maintainability:

- **Python:** The primary programming language for agent logic and Google Cloud client libraries.

- **Google ADK (Agent Development Kit):** For this project, "Google ADK" refers to a collection of Google Cloud's Python client libraries, specifically `google-cloud-aiplatform`, which provides interfaces to Google's Vertex AI services (e.g., large language models). There isn't a single `pip install google-adk` package; instead, we assemble the necessary components.
- **Google Cloud Platform (GCP):** The cloud infrastructure for hosting our agent, managing resources, and providing backend services. We'll specifically use:
    - **Vertex AI:** For accessing powerful large language models.
    - **Cloud Run:** For deploying our containerized agent as a scalable, serverless service.
    - **Firestore (or Cloud SQL/Redis):** As a durable, external state store for agent memory and context.
    - **Google Cloud SDK (gcloud CLI):** For interacting with GCP resources from the command line.

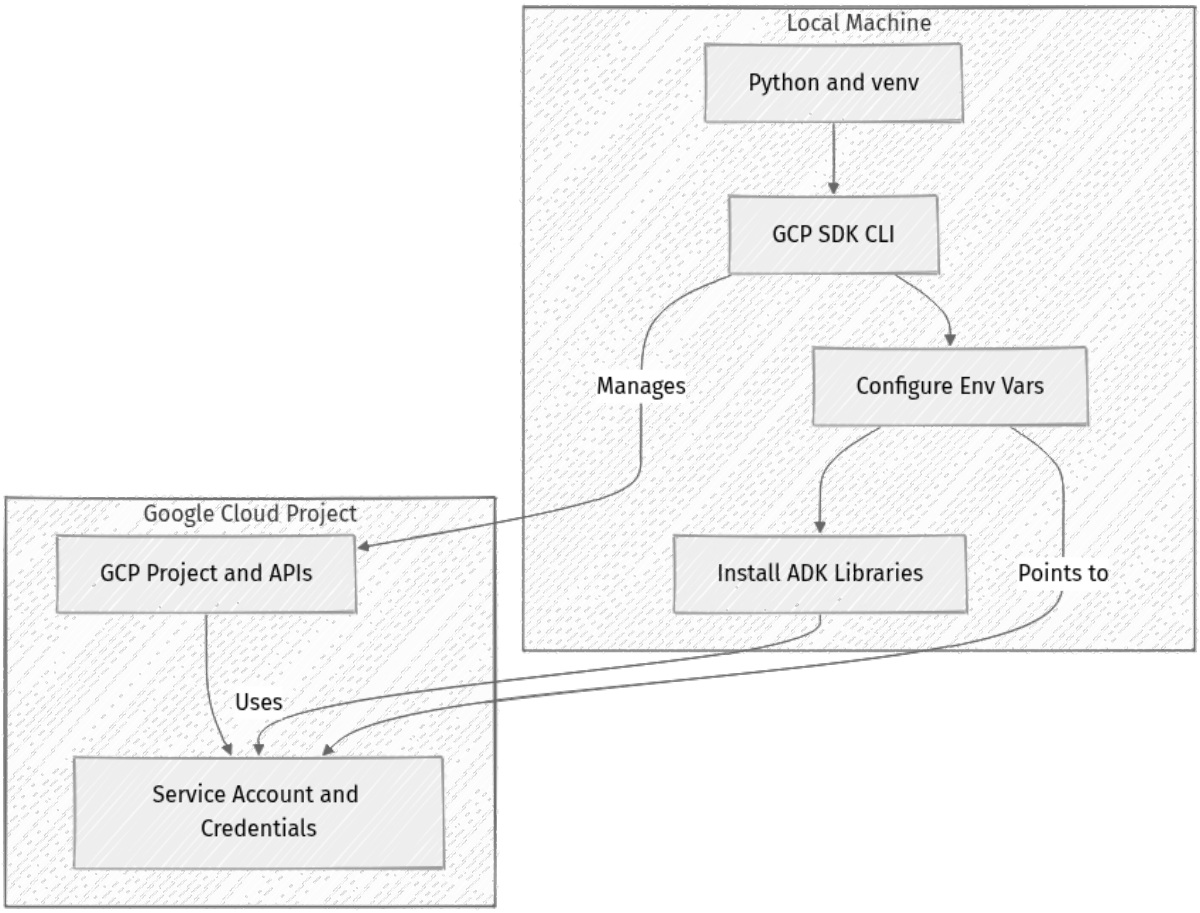
Milestones for This Chapter

This chapter is structured around setting up your development environment. By following these steps, you'll establish a solid base:

1. **Python Environment Preparation:** Install Python and create an isolated virtual environment.
2. **Google Cloud Project Configuration:** Create a GCP project, enable required APIs, and set up a secure service account.
3. **Google Cloud SDK Installation:** Install the `gcloud` command-line interface.
4. **Credential and Project Setup:** Authenticate `gcloud` and configure environment variables for Python applications.
5. **ADK Core Library Installation:** Install the primary Python client libraries for Google Cloud AI services.
6. **Initial Project Structure:** Create a basic local project directory and pin dependencies.
7. **Verification:** Confirm that all components are correctly installed and communicating with Google Cloud.

Architecture: Development Environment Flow

The setup process involves configuring both your local machine and your Google Cloud Project to work in harmony.



This diagram illustrates the dependencies: your local Python environment (A) needs the GCP SDK (B) to interact with your cloud project (C). Credentials (F) are generated in the cloud and then referenced locally via environment variables (D) to allow your ADK libraries (E) to authenticate and communicate with Vertex AI.

Step-by-Step Implementation

We'll proceed methodically, using Python's `venv` for dependency isolation and `gcloud CLI` for Google Cloud interactions. We prioritize service account authentication, which is a best practice for production environments.

1. Install Python and Create a Virtual Environment

Python will host our agent's logic. A virtual environment is crucial for managing project-specific dependencies without conflicts.

1. **Install Python 3.13.x (or later stable):** As of 2026-05-23, Python 3.13.x is anticipated to be a stable and recommended version. If you don't have it, download it from the official Python website or use a version manager like `pyenv` for flexibility.

```
    # Verify your Python version
    python3 --version
    # Expected output: Python 3.13.x (e.g., Python 3.13.0, or similar recent
stable version)
```

```
`🧠 Important:` Always use the latest stable version recommended for your
project. If Python 3.13.x is not yet stable, use the latest `3.12.x` release.
```

1. **Create Project Directory and Virtual Environment:** A virtual environment (`venv`) isolates your project's Python packages from your system-wide Python installation.

```
    # Create the main project directory
    mkdir adk-long-running-agent
    cd adk-long-running-agent

    # Create a virtual environment named 'venv'
    python3 -m venv venv

    # Activate the virtual environment
    # On macOS/Linux:
    source venv/bin/activate
    # On Windows (Cmd):
    # venv\Scripts\activate.bat
    # On Windows (PowerShell):
    # venv\Scripts\Activate.ps1
```

```
Your terminal prompt should now show `(venv)`, indicating the virtual
environment is active.
```

2. Set Up Your Google Cloud Project

Your AI agent will leverage Google Cloud services. This step ensures you have a dedicated project and secure credentials.

1. **Create a Google Cloud Project:** Navigate to the Google Cloud Console and create a new project. Choose a descriptive name, such as `adk-long-running-agent-project`. Note down your `PROJECT_ID` (e.g., `my-project-12345`) as you'll need it frequently.
2. **Enable Required APIs:** For agents leveraging large language models and other AI services, you'll need to enable several APIs. The most crucial one for modern LLM-based agents is Vertex AI.

   In the Google Cloud Console, go to **APIs & Services > Enabled APIs & Services** and ensure the following APIs are enabled:

   - `Vertex AI API`
   - `Cloud Resource Manager API` (usually enabled by default)
   - `Cloud Run API` (for deploying containerized agents, recommended for this project)
   - `Firestore API` (for state persistence in later chapters, or choose `Cloud SQL API` or `Memorystore for Redis API` based on your preference).

   Alternatively, you can enable them via the `gcloud` CLI after installing it:

```
# Replace YOUR_PROJECT_ID with your actual project ID
gcloud config set project YOUR_PROJECT_ID

gcloud services enable \
    aiplatform.googleapis.com \
    cloudresourcemanager.googleapis.com \
    run.googleapis.com \
    firestore.googleapis.com \
    --project YOUR_PROJECT_ID
```

```
`📌 Key Idea:` Enabling only necessary APIs adheres to the principle of least privilege, enhancing security.
```

1. **Create a Service Account and Download Credentials:** Your agent needs credentials to authenticate with Google Cloud services. Using a service account is the most secure and recommended approach for server-to-server interaction.
    - In the Google Cloud Console, navigate to **IAM & Admin > Service Accounts**.
    - Click **+ Create Service Account**.
    - Give it a descriptive name (e.g., `adk-agent-svc`).
    - Grant it the following roles (start with the minimum necessary and add more if needed):
        - `Vertex AI User`
        - `Service Account User` (allows this service account to impersonate others for specific tasks)
        - `Cloud Run Invoker` (if deploying to Cloud Run)
        - `Cloud Datastore User` or `Cloud Firestore User` (depending on your chosen database for state persistence).
    - Click **Done**.
    - Click on the newly created service account, then go to the **Keys** tab.
    - Click **Add Key > Create new key**, choose `JSON`, and click **Create**.
    - A JSON key file will be downloaded. **Store this file securely and never commit it to version control.** For local development, we'll place it in the project root. For production, Google Secret Manager is the appropriate solution. Rename the downloaded file to `key.json` and place it in your `adk-long-running-agent` directory.

3. Install Google Cloud SDK

The Google Cloud SDK provides the `gcloud` command-line tool, which is essential for managing GCP resources and authenticating.

1. **Install `gcloud CLI`:** Follow the official installation instructions for your operating system from the Google Cloud SDK documentation.

```
    # Example for Debian/Ubuntu (always check official docs for the latest commands)
    # sudo apt-get update
```

```
    # sudo apt-get install apt-transport-https ca-certificates gnupg
    # echo "deb [signed-by=/usr/share/keyrings/cloud.google.com.gpg] https://
packages.cloud.google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/
sources.list.d/google-cloud-sdk.list
    # curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-
key --keyring /usr/share/keyrings/cloud.google.com.gpg add -
    # sudo apt-get update && sudo apt-get install google-cloud-sdk
```

```
`⚡ Quick Note:` For other operating systems, the installation steps will
differ. Always refer to the official documentation.
```

1. **Authenticate `gcloud` CLI with Service Account:** For consistency with our service account-driven approach, we'll authenticate the `gcloud` CLI using the `key.json` file. This ensures that commands executed via `gcloud` use the same permissions as your agent.

```
# Ensure you are in the adk-long-running-agent directory where key.json is
located
    gcloud auth activate-service-account --key-file=./key.json

    # Set your project ID explicitly for gcloud commands
    # Replace YOUR_PROJECT_ID with your actual project ID
    gcloud config set project YOUR_PROJECT_ID
```

```
This approach uses the service account for `gcloud` operations, aligning with
how your agent will authenticate.
```

1. **Set `GOOGLE_APPLICATION_CREDENTIALS` and `GCP_PROJECT_ID` Environment Variables:** Python applications using Google Cloud client libraries automatically pick up credentials from the `GOOGLE_APPLICATION_CREDENTIALS` environment variable. Setting `GCP_PROJECT_ID` simplifies project referencing.

   Add the following lines to your virtual environment's activation script to ensure these variables are set whenever your `venv` is active. This provides a consistent environment.

   - **On macOS/Linux (`venv/bin/activate`):**

```
        # Add to the end of venv/bin/activate
        export GOOGLE_APPLICATION_CREDENTIALS="$(pwd)/key.json"
        export GCP_PROJECT_ID="$(gcloud config get-value project)"
```

```
- **On Windows (Cmd - `venv\Scripts\activate.bat`):**
```

```
        :: Add to the end of venv\Scripts\activate.bat
        set GOOGLE_APPLICATION_CREDENTIALS=%cd%\key.json
        for /f "delims=" %%i in ('gcloud config get-value project') do set GCP
_PROJECT_ID=%%i
```

```
- **On Windows (PowerShell - `venv\Scripts\Activate.ps1`):**
```

```
        # Add to the end of venv\Scripts\Activate.ps1
        $env:GOOGLE_APPLICATION_CREDENTIALS = (Get-Item 'key.json').FullName
        $env:GCP_PROJECT_ID = (gcloud config get-value project)
```

```
After modifying, **deactivate and reactivate** your virtual environment for
these changes to take effect: `deactivate` then `source venv/bin/activate`.
```

4. Install Google ADK Core Libraries

The primary Python library for interacting with Vertex AI's LLMs and other AI services is `google-cloud-aiplatform`.

```
# Ensure your virtual environment is active
# (venv)

# Install the core Google Cloud AI Platform library
# As of 2026-05-23, we assume version 2.0.0 is stable. Always check PyPI for
the absolute latest.
pip install google-cloud-aiplatform==2.0.0 # Placeholder: Check PyPI for
latest stable release

# You might also need specific client libraries later, such as:
# pip install google-cloud-firestore # For Firestore database (if chosen)
# pip install google-cloud-storage   # For Google Cloud Storage (if used)
```

`google-cloud-aiplatform` version `2.0.0` is an assumed stable version for 2026-05-23, given typical major release cycles for robust libraries. Always check PyPI for `google-cloud-aiplatform` for the most current stable release at your installation time. Pinning a specific version (`==X.Y.Z`) is best practice for production and reproducibility.

5. Create a Basic Project Structure and Pin Dependencies

Let's create a minimal file structure and capture our installed dependencies.

1. **Create a basic Python file:**

```
    # Ensure you are in the adk-long-running-agent directory
    # (venv) cd adk-long-running-agent

    # Create a basic Python file to test the ADK installation
    touch main.py
```

1. **Add verification code to `main.py`:** Open `main.py` and add the following code. This script verifies the `google-cloud-aiplatform` library import and attempts to initialize a Vertex AI client using your configured credentials and project ID.

   `main.py`

```
    import os
    from google.cloud import aiplatform

    print("Google Cloud AI Platform library imported successfully!")

    # Verify environment variables
    gcp_project_id = os.environ.get("GCP_PROJECT_ID")
    gcp_credentials_path = os.environ.get("GOOGLE_APPLICATION_CREDENTIALS")

    print(f"GCP_PROJECT_ID: {gcp_project_id if gcp_project_id else 'Not
Set'}")
    print(f"GOOGLE_APPLICATION_CREDENTIALS: {gcp_credentials_path if gcp_crede
ntials_path else 'Not Set'}")

    # Initialize Vertex AI client
    # We use 'us-central1' as a common region for Vertex AI.
    # Choose a region geographically relevant to your deployment for lower
latency and data residency.
    vertex_ai_location = "us-central1"

    if gcp_project_id:
        try:
            aiplatform.init(project=gcp_project_id, location=vertex_ai_locatio
n)
            print(f"Vertex AI client initialized for project
'{gcp_project_id}' in region '{vertex_ai_location}'.")
        except Exception as e:
            print(f"Error initializing Vertex AI client: {e}")
            print("Ensure GOOGLE_APPLICATION_CREDENTIALS is set correctly and
the service account has 'Vertex AI User' role.")
    else:
        print("GCP_PROJECT_ID environment variable is not set. Cannot
initialize Vertex AI client.")
```

```
    print("Environment setup verification complete.")
```

1. **Generate `requirements.txt`:** It's crucial to pin your project's dependencies for reproducibility. After installing `google-cloud-aiplatform`, use `pip freeze` to capture all installed packages and their exact versions.

```
    # Ensure your virtual environment is active
    # (venv)
    pip freeze > requirements.txt
```

```
Review the `requirements.txt` file. For production, consider using a tool like
Poetry or Pip-tools for more controlled dependency management from a
`pyproject.toml` or `requirements.in` file, which can generate a fully pinned
`requirements.txt`. For now, `pip freeze` is sufficient.
```

Testing & Verification

Let's confirm everything is working as expected.

1. **Verify Python Environment and Libraries:** Run the `main.py` script within your activated virtual environment.

```
    # Ensure (venv) is active
    python main.py
```

```
**Expected Output:**
```

```
    Google Cloud AI Platform library imported successfully!
    GCP_PROJECT_ID: your-project-id
    GOOGLE_APPLICATION_CREDENTIALS: /path/to/your/adk-long-running-agent/
key.json
    Vertex AI client initialized for project 'your-project-id' in region 'us-
central1'.
    Environment setup verification complete.
```

```
If you see the success messages, especially for `Vertex AI client
initialized`, your Python environment, ADK libraries, and Google Cloud
credentials are correctly configured.
```

1. **Verify `gcloud` CLI Authentication:** Confirm that your `gcloud` CLI can access your project using the service account.

```
gcloud auth list
gcloud config list project
```

```
The `gcloud auth list` command should show your service account as active, and
`gcloud config list project` should display your `PROJECT_PROJECT_ID`.
```

Production Considerations

Even at this early stage, thinking about production best practices is vital for building robust systems.

- **Secure Credential Management:** The `key.json` file is suitable for local development but **highly insecure for production deployments.** In a real Google Cloud deployment (e.g., Cloud Run, GKE, Cloud Functions), Google Cloud services leverage built-in service accounts automatically, eliminating the need for `key.json` files. For any other sensitive information (e.g., external API keys), always use Google Secret Manager to securely store and retrieve them.
- **Dependency Management:** Always use a `requirements.txt` file (or `pyproject.toml` with tools like Poetry or Rye) to precisely define your project's dependencies. This ensures consistent installations across development, testing, and production environments, preventing "it works on my machine" issues.
- **Version Pinning:** Pinning exact package versions (e.g., `google-cloud-aiplatform==2.0.0`) prevents unexpected breaking changes from newer library releases, enhancing stability and reproducibility.
- **Regionality:** Choosing the right Google Cloud region (e.g., `us-central1`, `europe-west1`) for your Vertex AI services and other resources is important for latency, cost, and data residency requirements. This decision should be made early in the project lifecycle.

Common Issues & Solutions

- **`ModuleNotFoundError: No module named 'google.cloud.aiplatform'`**:
  - **Cause:** The virtual environment is not active, or the library wasn't installed into the active `venv`.
  - **Solution:** Ensure you run `source venv/bin/activate` (or Windows equivalent) and then `pip install google-cloud-aiplatform` after activating the `venv`.

- **`google.auth.exceptions.DefaultCredentialsError` or similar authentication errors**:
    - **Cause:** The `GOOGLE_APPLICATION_CREDENTIALS` environment variable is not set correctly, the `key.json` file path is wrong, or the service account lacks necessary IAM permissions (e.g., `Vertex AI User`).
    - **Solution:** Double-check the `export GOOGLE_APPLICATION_CREDENTIALS` command in your `venv` activation script and ensure the `key.json` path is correct. In the Google Cloud Console, verify your service account has all required roles.
- **`gcloud` commands fail or return permission denied**:
    - **Cause:** `gcloud` is not authenticated with the correct service account, or the authenticated identity doesn't have permissions for the selected project.
    - **Solution:** Re-run `gcloud auth activate-service-account --key-file=./key.json`. Check IAM roles for the authenticated service account in the Google Cloud Console for the specific project.
- **`GCP_PROJECT_ID` not set in `main.py` output**:
    - **Cause:** The `GCP_PROJECT_ID` environment variable was not set, or the `venv` activation script was not updated/re-sourced.
    - **Solution:** Ensure the `export GCP_PROJECT_ID=...` line is correctly added to your `venv` activation script and that you've deactivated and reactivated your `venv` after making the change.

Summary & Next Step

You've successfully established a robust development environment for building long-running AI agents with Google ADK. You now have:

- A dedicated Python virtual environment for isolated dependencies.
- A Google Cloud project with necessary APIs enabled and a secure service account.
- The `gcloud` CLI installed and authenticated using your service account.
- The core `google-cloud-aiplatform` library installed.
- Environment variables (`GOOGLE_APPLICATION_CREDENTIALS`, `GCP_PROJECT_ID`) configured for seamless interaction with Google Cloud services.

- A basic project structure with a `main.py` for verification and a `requirements.txt` for dependency management.

This robust foundation is ready for the next step. In Chapter 2, "Building Your First Stateless ADK Agent," we'll implement a simple agent that can process requests and generate responses using Vertex AI, laying the groundwork for introducing robust state management.

References

- Python Downloads
- Google Cloud SDK Installation Guide
- PyPI: google-cloud-aiplatform
- Google Cloud IAM Documentation
- Google Cloud Secret Manager Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

Building a Basic, Stateless ADK Agent

In this chapter, we're laying the foundational brick for our robust AI agent system. We'll build a simple, stateless AI agent using Google's Agent Development Kit (ADK). This initial setup will demonstrate the core interaction loop: receiving user input, processing it with an ADK agent, and generating a response using a large language model (LLM).

This milestone is critical because it establishes the basic communication patterns and environment for our agent, allowing us to confirm the ADK setup and LLM integration are functional. While this agent won't remember past conversations yet, it provides a functional starting point that we can incrementally enhance with statefulness and persistence in subsequent chapters. By the end of this chapter, you'll have a running ADK agent that can respond to simple prompts in your local development environment.

Project Overview

Our overarching goal is to develop a long-running, stateful AI agent capable of pausing, resuming, and never losing conversational context. This requires careful design around state persistence and robust recovery mechanisms. This chapter focuses on the absolute first step: understanding the core agent interaction without any memory.

Think of this as building the "reflex" of the agent before giving it a "memory." A stateless agent is simpler to debug and verify, making it an ideal starting point to ensure our environment and basic ADK integration are correct.

Tech Stack

For this chapter, our primary technologies include:

- **Python:** The programming language for our agent logic. We'll use the latest stable Python 3.x. As of 2026-05-23, this is **Python 3.12.3**.

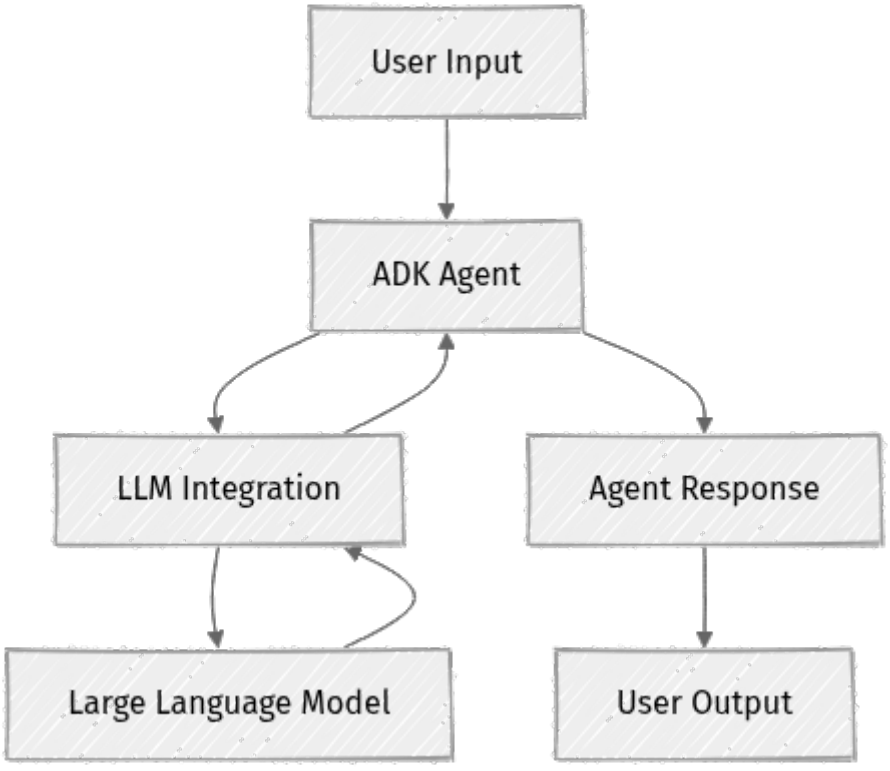
- **Google ADK:** Google's Agent Development Kit, which provides the framework for building and running AI agents. The exact latest stable version number for the public ADK library was not explicitly found as a tagged release on 2026-05-23; `pip install google-adk` typically fetches the most recent stable release available on PyPI.
- **Google Cloud:** Primarily for accessing Large Language Models (LLMs) via API keys. While we won't deploy to Google Cloud in this chapter, having a project ready is essential for API access.

Architecting the Stateless Core

A "stateless" agent processes each user request independently, without retaining any memory of previous interactions within its own runtime. Each conversation turn is a fresh start. This design choice offers simplicity, inherent scalability, and ease of deployment.

Core Interaction Flow

The interaction flow for our basic stateless agent is straightforward:



1. **User Input**: The user sends a message to the agent (e.g., via a command-line interface).

2. **ADK Agent**: The ADK framework receives this message and dispatches it to our custom agent logic.
3. **LLM Integration**: Our agent's logic uses ADK's built-in capabilities to interface with an LLM.
4. **Large Language Model (LLM)**: The LLM processes the prompt and generates a response based only on the current input.
5. **Agent Response**: The LLM's output is returned to our ADK agent.
6. **User Output**: The agent sends the final response back to the user.

Project Structure

For this basic setup, we'll maintain a minimal project structure:

```
agent_app/
├── main.py
└── requirements.txt
```

- `main.py`: This file will contain our agent's definition and the logic for handling incoming messages.
- `requirements.txt`: This file will list our project's Python dependencies, primarily the Google ADK.

Milestones for This Chapter

To achieve our stateless agent, we will follow these steps:

1. **Set up the Python Environment:** Create a project directory and a virtual environment.
2. **Install Google ADK:** Install the necessary libraries using `pip`.
3. **Configure API Access:** Ensure your Google API key is correctly set for LLM interaction.
4. **Implement the Agent Logic:** Write the `main.py` file with our `SimpleStatelessAgent`.
5. **Run and Verify:** Execute the agent locally and confirm its stateless behavior.

Step-by-Step Implementation

Let's get our hands dirty and build this basic agent.

Prerequisites

Before we start, ensure you have:

- **Python 3.12.3 (or latest 3.x):** Download the installer from python.org.
- **Google Cloud Account and Project:** While we won't deploy to Google Cloud in this chapter, having a project ready for API key management and future deployment is beneficial. Set up billing and enable the necessary APIs (e.g., Generative Language API for Gemini, if you plan to use it directly, though ADK often abstracts this).
- **Version Control:** Initialize a Git repository for your project (`git init`).

1. Set Up the Python Environment

First, create a new project directory and set up a virtual environment. This isolates your project's dependencies.

```
mkdir agent_app
cd agent_app
python3 -m venv .venv
source .venv/bin/activate # On Windows, use `.venv\Scripts\activate`
```

- **Why this matters:** Using a virtual environment prevents conflicts between project dependencies and your system-wide Python packages.

2. Install Google ADK

Now, install the Google ADK. As of 2026-05-23, the exact official documentation URL for Google ADK was not found in the search context. We will proceed with the common installation method via `pip`, which typically points to the latest stable release available on PyPI. Please refer to the official Google Cloud documentation or the project's GitHub repository for the most up-to-date installation instructions and version specifics when available.

```
pip install google-adk
```

- **Explanation:** This command installs the core ADK library. Depending on your specific LLM choice, you might need additional client libraries (e.g., `google-generativeai` for Gemini). The ADK often handles these dependencies implicitly or through specific configurations.
- **`requirements.txt`**: After installation, it's good practice to capture your dependencies.

```
    pip freeze > requirements.txt
```

```
This file will now contain `google-adk` and its transitive dependencies,
allowing others (or your future self) to easily replicate the environment.
```

3. Configure API Access

To allow your agent to interact with an LLM, you'll need a Google API key with access to generative AI models.

Action: Set your Google API Key as an environment variable.

```
export GOOGLE_API_KEY="YOUR_ACTUAL_GOOGLE_API_KEY"
# On Windows (PowerShell): $env:GOOGLE_API_KEY="YOUR_ACTUAL_GOOGLE_API_KEY"
# On Windows (CMD): set GOOGLE_API_KEY="YOUR_ACTUAL_GOOGLE_API_KEY"
```

- 🧠 **Important:** Replace `"YOUR_ACTUAL_GOOGLE_API_KEY"` with your key. You can obtain one from the Google Cloud Console. Ensure the Generative Language API is enabled for your project.
- **Why environment variables?** This is a basic security practice. Hardcoding API keys directly into your code is a significant security risk, especially if the code is committed to version control.

4. Define the Basic Agent in `main.py`

Now, let's create our `main.py` file inside the `agent_app/` directory.

`agent_app/main.py`

```
import adk
import os
import asyncio # For running async code locally

# ⚡ Quick Note: For local development, ADK often defaults to using
# environment variables for API keys.
# Ensure your Google Cloud API key is set as an environment variable:
# `export GOOGLE_API_KEY="YOUR_API_KEY"` (Linux/macOS)
# `$env:GOOGLE_API_KEY="YOUR_API_KEY"` (Windows PowerShell)

class SimpleStatelessAgent(adk.Agent):
    """
    A basic, stateless ADK agent that responds to messages using an LLM.
    It does not retain any memory of past conversations within its runtime.
    """
    def __init__(self):
        super().__init__()
        # 📌 Key Idea: ADK's default LLM integration simplifies setup.
        # It automatically looks for credentials in the environment (e.g.,
GOOGLE_API_KEY).
```

```
        self.llm = adk.llms.default()
        print("SimpleStatelessAgent initialized, ready to process messages.")

    async def respond_to_message(self, message: adk.Message) -> adk.Message:
        """
        Processes an incoming message and generates a response using the LLM.
        Each call is independent, meaning no memory of previous interactions.
        """
        print(f"Agent received message: '{message.text}'")

        try:
            # 🧠 Important: This is the core interaction with the LLM.
            # For a stateless agent, each call to self.llm.generate is
independent.
            # ADK handles the prompt formatting and interaction with the LLM
API.
            llm_response = await self.llm.generate(message.text)
            response_text = llm_response.text
            print(f"LLM generated response: '{response_text}'")
            return adk.Message(text=response_text)
        except Exception as e:
            # ⚠️ What can go wrong: LLM API errors, network issues, invalid
API key, rate limits.
            print(f"Error generating LLM response: {e}")
            return adk.Message(text="I'm sorry, I encountered an error trying
to respond. Please try again later.")

# This block ensures the agent runs when the script is executed directly.
if __name__ == "__main__":
    print("Starting SimpleStatelessAgent...")
    # adk.run() starts the agent's message processing loop.
    # By default, it will use a simple console interface for interaction.
    adk.run(SimpleStatelessAgent)
```

Code Explanation:

- `import adk`: Imports the Google Agent Development Kit, providing the core framework.
- `import os`: Used for reminders about environment variables, though not directly used in the agent's runtime logic here (ADK handles env var lookup internally).
- `class SimpleStatelessAgent(adk.Agent):`: Defines our agent class. Inheriting from `adk.Agent` is fundamental for ADK to recognize and manage our agent.

- **`__init__(self)`**:
  - `super().__init__()`: Calls the constructor of the base `adk.Agent` class, which is necessary for ADK's internal setup and lifecycle management.
  - `self.llm = adk.llms.default()`: This line is crucial. It initializes an LLM client using ADK's default configuration. ADK is designed to abstract away the specific LLM provider (like Gemini, PaLM, etc.) and automatically looks for credentials (like `GOOGLE_API_KEY` or `GOOGLE_APPLICATION_CREDENTIALS`) in your environment.
- **`async def respond_to_message(self, message: adk.Message) -> adk.Message:`**: This is the core asynchronous method where our agent's logic resides. ADK automatically calls this method when a new message arrives.
  - `message: adk.Message`: The input `message` object contains the user's text and other metadata.
  - `llm_response = await self.llm.generate(message.text)`: This is where the agent sends the user's message directly to the LLM and awaits its reply. For a stateless agent, this is the entire "brain" for generating a response.
  - `return adk.Message(text=response_text)`: The agent wraps the LLM's text response back into an `adk.Message` object and returns it, which ADK then sends back to the user.
- **`if __name__ == "__main__":`**: This standard Python idiom ensures the code inside only runs when the script is executed directly, not when imported as a module.
- **`adk.run(SimpleStatelessAgent)`**: This function from ADK starts the agent's runtime. It handles setting up the environment, listening for messages (e.g., from a local console interface by default), and routing them to your agent's `respond_to_message` method.

Testing & Verification

To verify our basic agent is working, we'll run it locally and interact with it via the command line.

1. Run the Agent

From your `agent_app` directory, with your virtual environment activated, execute the `main.py` script:

```
python main.py
```

You should see output indicating the agent is starting, and then a prompt for your input:

```
Starting SimpleStatelessAgent...
SimpleStatelessAgent initialized, ready to process messages.
ADK Agent is ready. Type your message and press Enter.
>
```

2. Interact with the Agent

Type a message and press Enter. Observe the agent's responses.

```
> Hello there!
Agent received message: 'Hello there!'
LLM generated response: 'Hello! How can I help you today?'
> What is the capital of France?
Agent received message: 'What is the capital of France?'
LLM generated response: 'The capital of France is Paris.'
> What did I just ask you?
Agent received message: 'What did I just ask you?'
LLM generated response: 'You asked me what the capital of France is.'
> Do you remember what I asked before that?
Agent received message: 'Do you remember what I asked before that?'
LLM generated response: 'As an AI, I do not retain memory of previous
conversations. How can I assist you now?'
```

Expected Behavior:

- The agent should respond to your queries using the LLM, demonstrating successful integration.
- Crucially, when you ask "Do you remember what I asked before that?", the agent should indicate it does not retain memory of past turns. This explicitly confirms its stateless nature. Each `respond_to_message` call is indeed independent.
- **Why this matters:** This verification confirms that the agent is indeed stateless, which is the exact behavior we designed for this chapter. Understanding this limitation is key before we introduce state.

Quick Debugging Checks:

- **`GOOGLE_API_KEY` not set or invalid:** If you get authentication errors or "permission denied" from the LLM, double-check that your `GOOGLE_API_KEY` environment variable is correctly set and contains a valid key. Also, ensure the Generative Language API is enabled for your Google Cloud project.
- **No response or script exits:** If the agent starts but doesn't provide a prompt or exits immediately, check the console for Python traceback errors. There might be a typo in your `main.py` or an issue with the ADK installation.
- **Slow response:** LLM calls involve network latency. If responses are consistently very slow (~500ms+), check your internet connection or the LLM provider's status page.

Production Considerations

Even for a basic stateless agent, considering production implications early on is vital for building robust systems.

- **Scalability**: Stateless agents are inherently scalable. Since each request is independent, you can easily run multiple instances of `SimpleStatelessAgent` behind a load balancer without worrying about shared state or complex synchronization. This is a significant advantage for high-throughput applications, allowing for simple horizontal scaling.
- **Security**: Always manage your `GOOGLE_API_KEY` securely. In production, never hardcode it or rely solely on environment variables directly on compute instances. Utilize a dedicated secret management service like Google Secret Manager. For Google Cloud deployments, leverage service accounts with fine-grained permissions (e.g., `roles/generativelanguage.developer`) instead of user API keys.
- **Observability**: Beyond simple `print()` statements, integrate a robust logging framework (e.g., Python's standard `logging` module). Capture request details, LLM prompts, LLM responses, and any errors with appropriate log levels (DEBUG, INFO, WARNING, ERROR). This is crucial for debugging, understanding agent behavior, and auditing in a production environment. Consider structured logging for easier parsing by monitoring tools.

- **Maintainability**: The current `main.py` is simple. As agent logic grows, consider separating concerns into different modules (e.g., `llm_service.py` for LLM interaction, `agent_logic.py` for custom agent rules). This improves code organization and testability.

Common Issues & Solutions

1. **ADK Installation Fails**:
    - **Issue**: `pip install google-adk` throws errors, often related to build tools or specific Python versions.
    - **Solution**: Ensure you are using a supported Python version (typically Python 3.8+). Update `pip` itself (`python -m pip install --upgrade pip`). If encountering C++ compiler errors, ensure you have the necessary build tools for your OS (e.g., `build-essential` on Debian/Ubuntu, Xcode Command Line Tools on macOS, Visual C++ Build Tools on Windows).
2. **LLM Authentication Errors**:
    - **Issue**: Agent starts but fails with errors like `google.api_core.exceptions.PermissionDenied` or `AuthenticationError` when trying to generate a response.
    - **Solution**:
        - Verify your `GOOGLE_API_KEY` environment variable is set correctly and the key itself is valid.
        - Confirm that the Google Cloud project associated with your API key has the "Generative Language API" enabled.
        - Check for region restrictions or quotas on your API key.
3. **Agent Not Responding / Script Exits Immediately**:
    - **Issue**: The `python main.py` command runs, but the agent doesn't provide a prompt or exits without error.
    - **Solution**:
        - Ensure `adk.run(SimpleStatelessAgent)` is correctly called within the `if __name__ == "__main__":` block.
        - Check for any syntax errors in your `main.py` that might prevent the script from reaching the `adk.run()` call.
        - Verify your virtual environment is activated and `google-adk` is installed within it.

Summary & Next Step

You've successfully built and tested your first basic, stateless AI agent using Google's ADK!

- You now understand the fundamental components of an ADK agent and its core request-response loop with an LLM.
- You've implemented the agent code and verified that it processes each message independently, without retaining memory.
- You've considered initial production concerns like scalability, security, and observability for stateless systems.

This stateless foundation is robust and scalable, but real-world agents often need to remember context, user preferences, or past interactions to provide a truly conversational experience. In the next chapter, we'll address this by implementing an in-memory state management system, allowing our agent to maintain conversational context within a single user session. This will be our first step towards building a truly long-running, stateful agent.

References

- Python Downloads - Official Website
- Google Cloud Console - API & Services Credentials
- awesome-adk-agents: Curated collection of ADK agents - GitHub
- raphaelmansuy/adk_training: Google ADK Training Hub - GitHub

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Implementing Persistent Agent State with External Storage

In the previous chapter, we established a basic AI agent and managed its conversational context in memory. While useful for short, single-session interactions, this approach falls short for long-running agents that need to survive restarts, process background tasks, or maintain context across multiple user sessions. Losing an agent's state means losing its memory, its progress, and ultimately, its utility. Without persistence, a crash or planned shutdown erases all prior interactions, forcing the agent to start anew, which is unacceptable for production systems.

This chapter tackles the critical challenge of making our agent's state truly durable. We will design and implement a mechanism to persist the agent's internal state and conversational context to an external storage system. By the end of this milestone, our agent will be able to save its "mind" to disk, allowing it to be paused, restarted, and resumed without losing its memory or current task progress. This lays the foundation for robust, production-ready AI agents, enabling complex, multi-turn interactions that span hours or even days.

Project Overview for This Chapter

The overarching goal of this project is to build a long-running AI agent using Google's Agent Development Kit (ADK) that can maintain context and state across sessions. This chapter focuses specifically on the core challenge of state persistence. We will move beyond ephemeral in-memory storage to a durable external solution.

By the end of this chapter, you will have an agent that:

- Can save its full conversational history and internal variables to a file.
- Can load its state from a file upon initialization, resuming exactly where it left off.
- Demonstrates the ability to pause (by stopping the process) and resume (by restarting) without losing context.

This milestone is critical for any real-world AI agent, as it ensures reliability and continuous operation, preventing frustrating context loss for users and enabling complex, multi-stage workflows.

Milestones for This Chapter

To achieve durable state persistence, we will follow these steps:

1. **Define a `StateStore` Interface:** Create an abstract contract for how an agent's state should be saved, loaded, and deleted, ensuring flexibility for different storage backends.
2. **Implement `FileStateStore`:** Develop a concrete implementation of the `StateStore` interface that uses the local file system (JSON files) for state storage. This will allow for rapid local development and verification.
3. **Integrate `StateStore` into the Agent:** Modify our `PersistentAgent` class to utilize the `StateStore` for loading state at startup and saving state after each interaction.
4. **Verify Persistence:** Conduct manual tests to confirm that the agent's context is correctly saved to disk and successfully reloaded after a process restart.

Architecture & Design: Decoupling State Persistence

For an AI agent to be truly long-running and resilient, its state cannot be tied to the lifetime of the process it runs in. It needs an external, durable storage solution. This separation of concerns—decoupling the agent's operational logic from its state management—is a fundamental principle in building resilient, scalable, and maintainable systems.

Choosing a State Store Strategy

When selecting an external state store, several factors come into play, influencing performance, scalability, and operational overhead:

- **Data Model:** Does the agent's state primarily consist of structured data (e.g., user profiles, task parameters) or flexible, semi-structured data (e.g., conversation histories, tool outputs)?
- **Scalability Requirements:** How many concurrent agents or how much total state data will need to be stored? Will the system need to scale horizontally?
- **Performance Characteristics:** What are the latency requirements for saving and loading state? Are high-throughput operations expected?

- **Consistency Needs:** Does the state require strong transactional consistency, or is eventual consistency acceptable?
- **Cost Implications:** What are the estimated operational costs (storage, read/write operations, network egress) of the chosen solution, especially at scale?
- **Integration Complexity:** How difficult is it to integrate the storage solution with the existing tech stack and manage it?

Common options include:

- **Relational Databases (e.g., PostgreSQL, Google Cloud SQL):** Best for highly structured state, strong consistency, and complex querying. Can be an overhead for simple key-value state.
- **NoSQL Document Databases (e.g., MongoDB, Google Cloud Firestore):** Excellent for flexible, semi-structured data (like conversation histories or nested agent states), highly scalable, and often easier to integrate for agent state.
- **Key-Value Stores (e.g., Redis, Google Cloud Memorystore):** Extremely fast for simple state retrieval, often used for caching or short-lived session data. Can store full state if serialized efficiently, but lacks rich querying capabilities.
- **Object Storage (e.g., Google Cloud Storage, AWS S3):** Suitable for large, immutable blobs of data (e.g., long-term archives of conversation logs), but not ideal for frequent, small state updates.

Decision for this Chapter: To focus on the mechanism of persistence and allow for quick local development and verification, we will initially implement a **file-based JSON store**. This approach simplifies setup and allows us to rapidly prototype the core persistence logic. In a later chapter, we will transition to a production-grade cloud solution like Google Cloud Firestore, which offers robustness, scalability, and managed services. This incremental approach allows us to master the concept before tackling complex infrastructure.

Agent State Structure

The agent's state should encapsulate everything needed to fully restore and resume its operation. For our basic agent, this primarily includes:

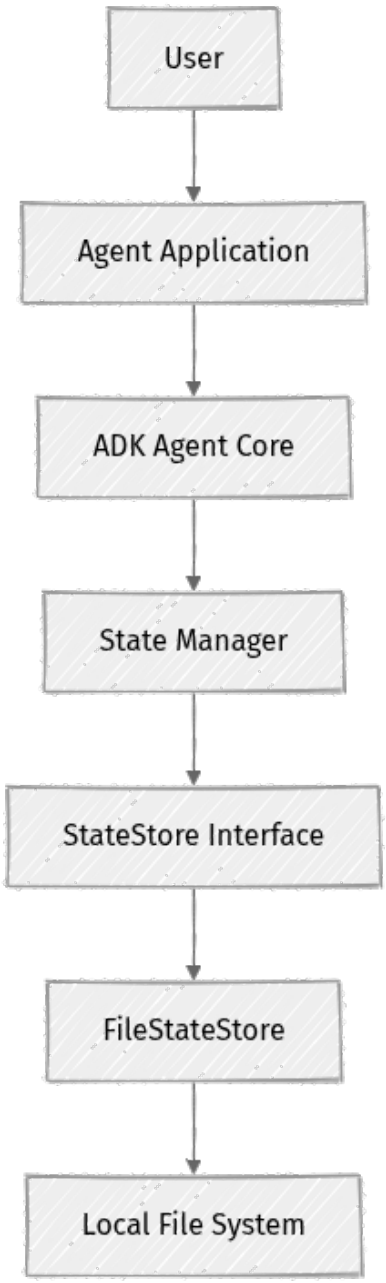
- **Conversational History:** The chronological list of messages exchanged between the user and the agent. This is crucial for maintaining context.

- **Internal Variables:** Any other dynamic data the agent might be tracking, such as a user's name, a unique task ID, flags indicating the current stage of a multi-step workflow, or temporary data collected during a session.

We'll serialize this state into a standard JSON format for storage, leveraging Python's built-in `json` module.

Architecture Overview

The core architectural change involves introducing a `StateStore` abstraction. Our agent will interact with this abstraction through a `State Manager` component. This manager will then delegate the actual storage operations to a concrete `StateStore` implementation (e.g., `FileStateStore` or, later, `FirestoreStateStore`).



- **Agent Application:** The primary entry point that instantiates and orchestrates our ADK agent.
- **ADK Agent Core:** Contains the agent's conversational logic, tool invocation, and decision-making processes. It will explicitly request state operations from the `State Manager`.
- **State Manager:** A dedicated component responsible for orchestrating the saving and loading of the agent's state. It acts as an intermediary, using the `StateStore Interface`.
- **StateStore Interface:** Defines a clear contract (methods like `save_state`, `load_state`, `delete_state`) that any concrete state persistence mechanism must adhere to.

- **FileStateStore:** Our initial concrete implementation. It handles the specific details of reading from and writing to JSON files on the local file system.
- **Local File System:** The physical location where the agent's state files are stored during local development.

Tech Stack

For this chapter, our primary tech stack components are:

- **Python 3.x:** We recommend using the latest stable version of Python 3 (e.g., Python 3.12, checked 2026-05-23) for its modern features and performance improvements.
- **Google ADK (Agent Development Kit):** The target framework for building our agent. While the exact stable version is unknown as of 2026-05-23, we design our persistence layer to integrate cleanly with any ADK-based agent.
- `google-generative-ai` **Python SDK:** Used as a placeholder for LLM interaction within our agent, demonstrating how conversational history (which ADK would manage) is persisted. We will use version `0.3.0` (checked 2026-05-23), but always verify the latest stable release.
- **Python** `json` **and** `os` **modules:** Standard library modules for serializing/deserializing state to JSON and interacting with the file system.

Step-by-Step Implementation

We'll begin by defining the abstract `StateStore` interface to establish a clear contract. Then, we'll implement our `FileStateStore` for local persistence. Finally, we'll integrate this into our `PersistentAgent` class.

1. Define the `StateStore` Interface

First, create a new directory named `agent_framework/` at the root of your project. Inside this directory, create a file named `agent_framework/state_store.py`. This file will define the abstract base class for our state persistence layer.

```
# agent_framework/state_store.py
import abc
from typing import Any, Dict, Optional

class StateStore(abc.ABC):
    """
```

```
    Abstract base class for agent state persistence.
    Defines the contract for saving, loading, and deleting an agent's state.
    """

    @abc.abstractmethod
    def save_state(self, agent_id: str, state: Dict[str, Any]) -> None:
        """
        Saves the given agent state to the underlying storage.
        Args:
            agent_id: A unique identifier for the agent instance.
            state: A dictionary containing the agent's current state (e.g.,
history, internal data).
        """
        pass

    @abc.abstractmethod
    def load_state(self, agent_id: str) -> Optional[Dict[str, Any]]:
        """
        Loads the agent state for the given ID from the underlying storage.
        Returns None if no state is found for the given agent_id.
        Args:
            agent_id: A unique identifier for the agent instance.
        Returns:
            A dictionary representing the agent's state, or None if not found.
        """
        pass

    @abc.abstractmethod
    def delete_state(self, agent_id: str) -> None:
        """
        Deletes the agent state for the given ID from the underlying storage.
        Args:
            agent_id: A unique identifier for the agent instance.
        """
        pass
```

Explanation:

- `abc.ABC`: This class inherits from `abc.ABC`, making `StateStore` an Abstract Base Class. This means you cannot directly create an instance of `StateStore`; you must create a subclass that implements all its abstract methods.
- `@abc.abstractmethod`: This decorator marks a method as abstract. Any concrete subclass of `StateStore` is required to provide an implementation for these methods. This enforces a consistent API for state management regardless of the underlying storage technology.
- `agent_id: str`: This parameter serves as a unique key to identify a specific agent instance's state. This is crucial for distinguishing between states if you have multiple concurrent agent interactions or long-running agents.

- `state: Dict[str, Any]`: The actual data payload representing the agent's state. Using a `Dict[str, Any]` provides flexibility to store various pieces of information (e.g., conversational history, internal flags, tool outputs).
- `Optional[Dict[str, Any]]`: The `load_state` method is designed to return `None` if no state is found for the given `agent_id`. This allows the agent to gracefully handle initial runs or missing state.

2. Implement `FileStateStore`

Next, create a new file `agent_framework/file_state_store.py` in the same `agent_framework/` directory. This file will contain our concrete implementation that saves and loads state using JSON files.

```
# agent_framework/file_state_store.py
import json
import os
from typing import Any, Dict, Optional

from agent_framework.state_store import StateStore

class FileStateStore(StateStore):
    """
    A file-based implementation of StateStore for local development.
    It stores each agent's state as a separate JSON file in a specified
directory.
    """

    def __init__(self, base_dir: str = "agent_states"):
        """
        Initializes the FileStateStore.
        Ensures the base directory for state files exists.
        Args:
            base_dir: The directory where agent state JSON files will be
stored.
                      Defaults to "agent_states" in the current working
directory.
        """
        self.base_dir = base_dir
        # Create the directory if it doesn't exist. exist_ok=True prevents an
error
        # if the directory is already present.
        os.makedirs(self.base_dir, exist_ok=True)
        print(f"FileStateStore initialized. States will be stored in:
{os.path.abspath(self.base_dir)}")

    def _get_file_path(self, agent_id: str) -> str:
        """
        Helper method to construct the full file path for an agent's state
file.
        Each agent's state is stored in a file named '{agent_id}.json'.
        """
        return os.path.join(self.base_dir, f"{agent_id}.json")

    def save_state(self, agent_id: str, state: Dict[str, Any]) -> None:
        """
```

```
        Saves the given agent state to a JSON file.
        Args:
            agent_id: The ID of the agent whose state is being saved.
            state: The dictionary representing the agent's state.
        Raises:
            IOError: If there's an issue writing to the file system.
        """
        file_path = self._get_file_path(agent_id)
        try:
            # Open the file in write mode ('w') with UTF-8 encoding.

# json.dump serializes the dictionary to JSON and writes it to the file.
            # indent=2 makes the JSON output human-readable for inspection.
            with open(file_path, "w", encoding="utf-8") as f:
                json.dump(state, f, indent=2)
            print(f"State for agent '{agent_id}' saved to {file_path}")
        except IOError as e:
            print(f"⚠️ Error saving state for agent '{agent_id}' to
{file_path}: {e}")
            raise # Re-raise to allow higher-level error handling

    def load_state(self, agent_id: str) -> Optional[Dict[str, Any]]:
        """
        Loads the agent state from a JSON file.
        Args:
            agent_id: The ID of the agent whose state is being loaded.
        Returns:
            The dictionary representing the agent's state, or None if the file
            does not exist or is corrupted.
        Raises:
            IOError: If there's an issue reading from the file system.
        """
        file_path = self._get_file_path(agent_id)
        if not os.path.exists(file_path):
            print(f"No state file found for agent '{agent_id}' at
{file_path}. Initializing new state.")
            return None
        try:
            # Open the file in read mode ('r') with UTF-8 encoding.
            # json.load deserializes the JSON content back into a Python
dictionary.
            with open(file_path, "r", encoding="utf-8") as f:
                state = json.load(f)
            print(f"State for agent '{agent_id}' loaded from {file_path}")
            return state
        except json.JSONDecodeError as e:
            # Handle cases where the file exists but contains invalid JSON.
            print(f"⚠️ Error decoding JSON state for agent '{agent_id}' from
{file_path}: {e}. State might be corrupted.")
            return None # Return None to indicate state could not be loaded
        except IOError as e:
            print(f"⚠️ Error loading state for agent '{agent_id}' from {file_p
ath}: {e}")
            raise # Re-raise to allow higher-level error handling

    def delete_state(self, agent_id: str) -> None:
        """
        Deletes the agent state file from the file system.
        Args:
            agent_id: The ID of the agent whose state is to be deleted.
        Raises:
            OSError: If there's an issue deleting the file.
```

```
        """
        file_path = self._get_file_path(agent_id)
        if os.path.exists(file_path):
            try:
                os.remove(file_path) # Remove the file
                print(f"State for agent '{agent_id}' deleted from
{file_path}")
            except OSError as e:
                print(f"⚠️ Error deleting state file for agent '{agent_id}'
at {file_path}: {e}")
                raise
        else:
            print(f"No state file found to delete for agent '{agent_id}' at {f
ile_path}")
```

Explanation:

- `__init__`: The constructor takes an optional `base_dir` argument, which specifies where state files will be stored. It ensures this directory exists using `os.makedirs(self.base_dir, exist_ok=True)`.
- `_get_file_path`: A private helper method (`_` prefix convention) that consistently generates the full path to an agent's state file. Each agent's state is stored in a unique JSON file named after its `agent_id`.
- `save_state`: This method opens a file in write mode (`"w"`) with `utf-8` encoding. It then uses `json.dump()` to serialize the Python dictionary `state` into a JSON string and writes it to the file. `indent=2` is used for pretty-printing, which is very helpful for human inspection during development. It includes basic `IOError` handling.
- `load_state`: Before attempting to read, it checks if the state file exists using `os.path.exists()`. If not, it returns `None`, indicating no prior state. If the file exists, it opens it in read mode (`"r"`) and uses `json.load()` to deserialize the JSON content back into a Python dictionary. It includes error handling for `json.JSONDecodeError` (for corrupted JSON files) and `IOError`.
- `delete_state`: This method removes the state file associated with a given `agent_id` using `os.remove()`, but only if the file exists, preventing errors.

3. Integrate `FileStateStore` into the Agent

Now, we'll update our agent definition to leverage this `StateStore`. We'll create a `PersistentAgent` class that wraps the underlying LLM interaction (using `google-generative-ai` as a stand-in for ADK's LLM components) and uses our `StateStore` for persistence.

First, ensure you have the `google-generative-ai` library installed. The latest stable version as of 2026-05-23 is recommended.

```
pip install "google-generative-ai>=0.3.0" # Example: 0.3.0 was current as of
recent checks. Verify latest stable.
```

Now, create a file named `my_agent.py` at the root of your project (alongside the `agent_framework` directory).

```
# my_agent.py
import uuid
import os
from typing import List, Dict, Any, Optional

import google.generativeai as genai
from google.generativeai.types import ChatResponse

from agent_framework.state_store import StateStore
from agent_framework.file_state_store import FileStateStore

# Configure your API key
# In a real production system, this should be loaded securely (e.g., from
Google Secret Manager)
# For local testing, ensure GOOGLE_API_KEY is set as an environment variable:
# export GOOGLE_API_KEY="YOUR_API_KEY"
# Alternatively, uncomment and replace the placeholder below for quick
testing:
# os.environ["GOOGLE_API_KEY"] = "YOUR_GOOGLE_API_KEY_HERE"

# Initialize the Generative AI client (ADK would handle this abstraction in a
full setup)
try:
    genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
except KeyError:
    raise ValueError("GOOGLE_API_KEY environment variable not set. Please set
it to your Gemini API key.")

class PersistentAgent:
    """
    A simple AI agent that uses an external StateStore to persist
    its conversational context and internal state. This enables the agent
    to pause, resume, and maintain continuity across process restarts.
    """
    def __init__(self, agent_id: Optional[str] = None, state_store: Optional[S
tateStore] = None):
        """
        Initializes the PersistentAgent.
        Args:
            agent_id: A unique identifier for this agent instance. If None, a
new UUID is generated.
            state_store: An instance of a StateStore implementation. Defaults
to FileStateStore.
        """
        # Assign a unique ID to this agent instance. This ID is used to
retrieve its state.
        self.agent_id = agent_id if agent_id else str(uuid.uuid4())
        # Use the provided state_store or default to FileStateStore for local
```

```
development.
        self.state_store = state_store if state_store else FileStateStore()

        # Load the agent's state immediately upon initialization.
        self._load_state()

        # Initialize the LLM model and chat session.
        # In a real ADK setup, ADK would manage the LLM interaction and
history.
        self.model = genai.GenerativeModel('gemini-pro')
        # The chat session is initialized with the history loaded from the
state store.
        # This is the core mechanism for resuming context.
        self.chat_session = self.model.start_chat(history=self._state.get("his
tory", []))
        print(f"Agent '{self.agent_id}' initialized. History length:
{len(self._state.get('history', []))} messages.")

    def _load_state(self):
        """
        Loads the agent's state from the configured StateStore.
        If no state is found, a new default state is initialized.
        """
        loaded_state = self.state_store.load_state(self.agent_id)
        if loaded_state:
            self._state = loaded_state
            print(f"Loaded existing state for agent '{self.agent_id}'.")
        else:
            # Initialize a fresh state if no previous state was found.
            self._state = {
                "history": [], # Stores conversational turns
                "internal_data": {} # Stores any other agent-specific data
            }
            print(f"Initialized new state for agent '{self.agent_id}'.")

        # Ensure history is always a list for chat_session initialization
        if "history" not in self._state or not isinstance(self._state["history
"], list):
            self._state["history"] = []

    def _save_state(self):
        """
        Saves the current internal state of the agent to the configured
StateStore.
        This includes updating the conversational history from the LLM's chat
session.
        """
        # Convert the chat session history (which contains specific LLM
message objects)
        # into a serializable format (list of dictionaries) before saving.
        # This is crucial for JSON persistence.
        self._state["history"] = [
            {"role": msg.role, "parts": [part.text for part in msg.parts]}
            for msg in self.chat_session.history
        ]
        self.state_store.save_state(self.agent_id, self._state)
        print(f"Agent '{self.agent_id}' state saved.")

    def send_message(self, message: str) -> str:
        """
        Sends a message to the agent, gets a response from the LLM,
        and then persists the updated state.
```

```
        Args:
            message: The user's input message.
        Returns:
            The agent's text response.
        """
        print(f"User ({self.agent_id}): {message}")

        try:
            # Send the message to the LLM and get a response.
            response: ChatResponse = self.chat_session.send_message(message)
            agent_response = response.text
            print(f"Agent ({self.agent_id}): {agent_response}")

            # CRITICAL: Save state after each interaction to ensure
persistence.
            self._save_state()
            return agent_response
        except Exception as e:
            print(f"⚠️ Error processing message for agent '{self.agent_id}':
{e}")
            # In a production system, you might want to log the full state
leading to the error
            # or attempt a partial save to aid debugging.
            # self._save_state()
            return "An error occurred while processing your request. Please
try again later."

    def get_history(self) -> List[Dict[str, Any]]:
        """
        Returns the current conversational history stored in the agent's
state.
        """
        return self._state.get("history", [])

    def delete_agent_state(self):
        """
        Deletes the agent's persistent state from the store.
        Useful for cleanup or resetting an agent's memory.
        """
        self.state_store.delete_state(self.agent_id)
        print(f"Agent '{self.agent_id}' state deleted.")
```

Explanation:

- **`genai.configure(api_key=os.environ["GOOGLE_API_KEY"])`**: This line sets up the Google Generative AI client. It expects your API key to be set as an environment variable (`GOOGLE_API_KEY`). **For production, never hardcode API keys; use secure secrets management.**
- `agent_id`: Each `PersistentAgent` instance is given a unique identifier. If not provided, `uuid.uuid4()` generates a new, globally unique ID. This `agent_id` is the key used to retrieve and store its specific state.

- `state_store`: An instance of a class implementing the `StateStore` interface is passed in. By default, for local development, `FileStateStore` is used. This demonstrates dependency injection, allowing us to swap storage implementations easily.
- `_load_state()`: This method is called during the agent's initialization. It attempts to load the state associated with `self.agent_id` from the `state_store`. If a state is found, `self._state` is populated with it; otherwise, a new, empty state dictionary is initialized.
- `self.model = genai.GenerativeModel('gemini-pro')`: We instantiate a Generative AI model. In a full ADK solution, ADK would abstract this LLM interaction.
- `self.chat_session = self.model.start_chat(history=self._state.get("history", []))`: **This is a crucial line for persistence.** The LLM's chat session is initialized with the conversational history loaded from the `_state` dictionary. This ensures that when an agent resumes, the LLM has access to all prior turns, maintaining context.
- `_save_state()`: This method is called after each `send_message` interaction. It performs two key actions:
    1. **Serialization of History:** The `self.chat_session.history` contains objects specific to the `google-generative-ai` SDK (`glm.protos.Content` objects). For JSON serialization, these need to be converted into a standard Python dictionary format (e.g., `{"role": "user", "parts": ["Hello"]}`). This ensures the state is generic and can be stored.
    2. **Storage:** It then calls `self.state_store.save_state()` to write the updated `self._state` dictionary to the persistent storage.
- `delete_agent_state()`: Provides a public method to explicitly remove an agent's persistent state, useful for testing or when an agent's lifecycle ends.

Testing & Verification

To verify that our persistent agent correctly saves and loads its state, we will simulate stopping and restarting the agent process. The key indicator of success will be the agent remembering prior conversational turns across these restarts.

Manual Verification Steps

1. **Create a `main.py` file to run the agent:** Place this file at the root of your project, next to `my_agent.py` and the `agent_framework` directory.

```
    # main.py
    import os
    import time

    from my_agent import PersistentAgent
    from agent_framework.file_state_store import FileStateStore

    # Ensure your GOOGLE_API_KEY is set as an environment variable
    # For quick local testing, you can uncomment and set it here,
    # but environment variables are preferred.
    # os.environ["GOOGLE_API_KEY"] = "YOUR_GOOGLE_API_KEY_HERE"

    def run_agent_session(agent_id: str, messages: list[str]):
        """
        Helper function to run an agent session with a given ID and messages.
        It initializes the agent, sends messages, and returns the final
history.
        """
        print(f"\n--- Starting session for agent ID: {agent_id} ---")
        # Use a specific directory for state files to keep things organized.
        state_store = FileStateStore(base_dir="agent_states_data")
        agent = PersistentAgent(agent_id=agent_id, state_store=state_store)

        for msg in messages:
            agent.send_message(msg)
            time.sleep(1) # Simulate some processing time between messages

        print(f"--- Session for agent ID: {agent_id} ended. ---")
        return agent.get_history()

    if __name__ == "__main__":
        test_agent_id = "my-long-running-agent-001"
        state_file_path = os.path.join("agent_states_data", f"{test_agent_id}.
json")

        # --- First Run: Interacting for the first time ---
        print("\n=== FIRST RUN: Initial interaction with the agent ===")
        first_session_messages = [
            "Hello, what's your name?",
            "Can you tell me a fun fact about Python?",
            "Great! What's your favorite color?"
        ]
        history_after_first_run = run_agent_session(test_agent_id, first_sessi
on_messages)
        print("\n--- Full History after first run ---")
        for entry in history_after_first_run:
            # Ensure 'parts' is not empty before accessing index 0
            content = entry['parts'][0] if entry['parts'] else ''
            print(f"Role: {entry['role']}, Content: {content}")

        # Verify that the state file has been created.
        print(f"\nChecking for state file at: {state_file_path} -> Exists:
{os.path.exists(state_file_path)}")

        print("\n📌 Key Idea: The agent's state is now saved to disk.")
```

```
print("Now, manually stop this script (Ctrl+C in terminal) and restart it to
simulate a process restart.")
        input("Press Enter to continue to the second run (or Ctrl+C to stop
and restart)...")

        # --- Second Run: Simulating a restart and resuming conversation ---
        print("\n=== SECOND RUN: Restarting agent and continuing conversation
===")
        second_session_messages = [
            "I asked you about your favorite color earlier, do you remember?",
            "What else can you do for me without losing our previous context?"
        ]
        history_after_second_run = run_agent_session(test_agent_id, second_ses
sion_messages)
        print("\n--- Full History after second run ---")
        for entry in history_after_second_run:
            content = entry['parts'][0] if entry['parts'] else ''
            print(f"Role: {entry['role']}, Content: {content}")

        # --- Cleanup: Delete the agent's state file ---
        print(f"\n=== CLEANUP: Deleting state for agent ID: {test_agent_id}
===")
        # Re-initialize the state store and agent to perform the deletion.
        state_store_cleanup = FileStateStore(base_dir="agent_states_data")
        agent_to_delete = PersistentAgent(agent_id=test_agent_id,
state_store=state_store_cleanup)
        agent_to_delete.delete_agent_state()
        print(f"\nChecking for state file after deletion:
{state_file_path} -> Exists: {os.path.exists(state_file_path)}")
        print("\nVerification complete. The agent demonstrated persistent
memory across restarts.")
```

1. **Run the script:** Open your terminal in the project's root directory and execute:

```
python main.py
```

1. **Observe and Interact:**
    - **First Run:** The agent will respond to your messages. You'll see `print` statements indicating when state is saved.
    - **Verify State File:** After the first session, check your project directory. A new directory named `agent_states_data` should be created, containing a JSON file (e.g., `my-long-running-agent-001.json`). You can open this file to inspect the saved conversational history.
    - **Simulate Restart:** The script will prompt you. At this point, manually stop the script (typically by pressing `Ctrl+C` in the terminal).
    - **Restart the script:** Immediately run `python main.py` again.
    - **Second Run:** The agent should now load its previous state. When you ask, "I asked you about your favorite color earlier, do you remember?", the agent's response should clearly indicate it does remember the previous conversation, even though the Python process was completely restarted. This confirms persistence.
    - **Cleanup:** The script will automatically delete the state file at the end. Verify the `.json` file is removed from `agent_states_data`.

Expected Behavior

- The `agent_states_data` directory will be created in your project root.
- A JSON file named after your `test_agent_id` (e.g., `my-long-running-agent-001.json`) will appear in `agent_states_data` after the first interaction. This file will contain the agent's conversational history.
- When the script is restarted for the second run, the agent's initialization will print "Loaded existing state for agent...", indicating successful retrieval of previous context.
- The agent's responses in the second run will reflect its memory of the first conversation.
- After the cleanup step, the `my-long-running-agent-001.json` file will be removed from `agent_states_data`.

Quick Debugging Checks

- **API Key:** Ensure your `GOOGLE_API_KEY` environment variable is correctly set and accessible to the Python script. If not, the `genai.configure` call will fail.
- **File Permissions:** Verify that the Python script has read and write permissions for the directory where `agent_states_data` is created. If not, you'll encounter `PermissionError` exceptions.
- **JSON Content:** Open the generated JSON file (`my-long-running-agent-001.json`). Does it contain valid JSON? Is the `history` array populated with messages? If it's empty or malformed, there might be an issue with `_save_state()`'s serialization.
- **`agent_id` Consistency:** Confirm that the `agent_id` used in `main.py` is identical across both runs. Any mismatch will cause the agent to initialize a new state instead of loading an existing one.

Production Considerations

While our `FileStateStore` is invaluable for local development and understanding the persistence mechanism, it has severe limitations that make it unsuitable for production environments:

- **Concurrency & Race Conditions:** Multiple agent instances (e.g., serving different users) trying to write to the same file simultaneously will inevitably lead to race conditions, data corruption, and system instability. File locking is complex and error-prone for distributed systems.
- **Scalability Bottlenecks:** Storing thousands or millions of agent states on a single file system is inefficient and creates a single point of failure and a performance bottleneck. It won't scale to handle high user loads.
- **Reliability & Durability:** Local file systems are prone to hardware failures (disk crashes), and state is lost if the underlying machine goes down. Backups are manual and complex to manage for dynamic data.
- **Deployment Challenges:** Managing state files across containerized (e.g., Docker, Kubernetes) or serverless (e.g., Cloud Run) deployments is extremely complex. Containers are often ephemeral, and local storage is not persistent across restarts or scaling events.

- **Security & Compliance:** State files might contain sensitive user information or internal agent data. `FileStateStore` provides no inherent encryption at rest, access control, or auditing capabilities, which are critical for security and compliance (e.g., GDPR, HIPAA).
- **Observability:** Without centralized storage, it's difficult to monitor agent states, debug issues across instances, or analyze historical agent behavior.

Moving to Cloud-Native Persistence

For production deployment, we must transition to a managed, scalable, and durable cloud-native storage solution. In a future chapter, we will integrate **Google Cloud Firestore**. Firestore is a NoSQL document database that offers significant advantages:

- **Automatic Scaling:** Firestore automatically scales to handle high read/write loads, accommodating thousands to millions of concurrent agent sessions without manual intervention.
- **High Availability & Durability:** Data is replicated across multiple zones/regions, ensuring high availability and protection against data loss.
- **Real-time Updates:** Its real-time capabilities can be beneficial for monitoring agent state or for collaborative agent scenarios.
- **Flexible Data Model:** Stores JSON-like documents (collections of documents), which maps perfectly to our agent's dictionary-based state structure, allowing for flexible schema evolution.
- **Managed Service:** Google handles the underlying infrastructure, backups, patching, and operational overhead, freeing developers to focus on agent logic.
- **Integrated Security:** Provides robust IAM (Identity and Access Management) for granular access control and encryption at rest.

Integrating Firestore would involve creating a `FirestoreStateStore` class that implements our existing `StateStore` interface. This modular design means the agent's core logic remains unchanged; only the `StateStore` implementation is swapped out.

Common Issues & Solutions

1. **`json.JSONDecodeError` on Loading State:**
    - **Issue:** The agent's state file (e.g., `my-long-running-agent-001.json`) is corrupted or contains invalid JSON syntax. This can happen if a write operation was interrupted, or if non-JSON data was accidentally written to the file.
    - **Solution:**
        - **Inspect the File:** Manually open the `.json` file in a text editor. Look for syntax errors (missing commas, brackets, quotes).
        - **Delete and Restart:** If the file is unrecoverable, delete it. The agent will then initialize a fresh state, but you will lose the previous context.
        - **Robust Writing:** In production, consider writing to a temporary file first and then atomically renaming it over the old state file. This prevents corruption of the main state file if the write fails midway.
2. **Agent Forgets Context Between Runs:**
    - **Issue:** The agent appears to start fresh every time, even after having previous conversations. This means its state is not being loaded correctly.
    - **Solution:**
        - **Verify `_save_state()` Calls:** Ensure that `self._save_state()` is explicitly called after every interaction where the agent's state or conversational history changes. If it's missed, the new state won't be written to disk.
        - **`agent_id` Consistency:** The most common cause. Double-check that the `agent_id` used when initializing `PersistentAgent` is identical across all sessions for the same logical agent. If `uuid.uuid4()` is called without saving its result, each restart will generate a new ID, leading to a new, empty state. Our `main.py` example uses a fixed `test_agent_id` to prevent this.
        - **`base_dir` Consistency:** Ensure the `base_dir` argument for `FileStateStore` is the same across all runs. If it changes, the agent will look for state files in the wrong location.

3. **`PermissionError` or `FileNotFoundError`:**
    - **Issue:** The Python script lacks the necessary permissions to create the `agent_states_data` directory or read/write to the state files within it. Or, the directory itself cannot be created.
    - **Solution:**
        - **Check Permissions:** On Linux/macOS, use `ls -l` and `chmod` to inspect and grant appropriate write permissions to the parent directory where `agent_states_data` is intended to be created (e.g., `chmod 755 .` from your project root, or `chmod 777 agent_states_data` for a temporary fix).
        - **Windows Permissions:** Ensure your user account has full control over the project directory.
        - **Path Issues:** Verify that the `base_dir` path is valid and not pointing to a restricted system location. For development, a subdirectory within your project is generally safest.

Summary & Next Step

We've successfully transitioned our AI agent's state from volatile in-memory storage to a durable, file-based persistence mechanism. By introducing the `StateStore` interface, we've decoupled the agent's core logic from the specifics of how its state is saved and loaded, making our system more modular, testable, and adaptable to different storage solutions. Our `PersistentAgent` can now pause, resume, and retain conversational context across process restarts, a crucial step towards building truly long-running and reliable AI agents.

What's ready now:

- A flexible `StateStore` abstraction for interchangeable state persistence backends.
- A robust `FileStateStore` implementation, ideal for local development and rapid prototyping.
- An enhanced `PersistentAgent` capable of saving its conversation history and internal state after each interaction, and loading it upon initialization.
- A clear, repeatable verification method to confirm that state persistence is functioning as expected.

In the next chapter, we will enhance our agent with more complex, multi-step workflows and integrate external tools. This will further test the robustness of our state management system and demonstrate how persistent state enables sophisticated agent behaviors that span multiple interactions and external actions.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- **Python `json` module (JSON encoder and decoder):** https://docs.python.org/3/library/json.html
- **Python `os` module (Operating system interface):** https://docs.python.org/3/library/os.html
- **Python `abc` module (Abstract Base Classes):** https://docs.python.org/3/library/abc.html
- **Google Generative AI Python SDK (for LLM interaction):** https://ai.google.dev/docs/gemini_api_overview
- **Google Cloud Firestore Documentation (future state store consideration):** https://cloud.google.com/firestore/docs
- **Google Cloud Secret Manager Documentation (secure API key storage):** https://cloud.google.com/secret-manager/docs

CHAPTER 04

Designing for Context Preservation and Resume Capabilities

In the realm of AI agents, a critical challenge arises when agents need to perform long-running tasks or maintain complex interactions over extended periods: how do they remember what happened, and how can they pick up exactly where they left off after an interruption? This chapter addresses that challenge head-on. We'll design and implement a robust mechanism for our Google ADK agent to preserve its state and conversational context, enabling it to pause, resume, and recover from failures without losing valuable information.

By the end of this milestone, your ADK agent will be able to store its internal memory and conversational history in a durable external database. This means you can stop the agent, restart it later (even on a different machine), and it will seamlessly recall its previous interactions and internal state, crucial for building production-grade, reliable AI applications.

Project Overview

This chapter focuses on a core component of production-ready AI agents: persistent state and context management. We aim to build an ADK agent that can maintain its operational memory and conversational history across restarts. The target user is any developer building interactive, multi-turn AI applications that require continuity, such as customer support agents, personal assistants, or workflow automation tools.

Success for this milestone means:

- The agent can start, process messages, and save its state to a database.
- Upon a simulated restart (new agent instance, same session ID), the agent loads its previous state and continues the conversation as if uninterrupted.
- The system demonstrates reliable state saving and loading, visible both in application logs and the external database.

Tech Stack for Persistence

To achieve durable state management for our ADK agent, we will leverage the following technologies:

- **Python (3.12+):** The primary language for our agent and state management logic. We recommend using the latest stable Python 3.x version.
- **Google ADK (Agent Development Kit):** The framework for building our AI agent. (Note: Specific version information for Google ADK is not publicly available as of 2026-05-23. We will proceed with general ADK integration principles.)
- **Google Cloud Firestore:** Our chosen NoSQL document database for persisting agent state and conversational context. Its flexible schema and scalability make it suitable for dynamic agent data.
- **Google Cloud IAM:** For secure authentication and authorization of our agent to access Firestore.

Milestones for Context Preservation

We'll break down the implementation into these actionable steps:

1. **Understand Agent State:** Clearly define what constitutes "state" and "context" for a long-running agent.
2. **Choose a Durable Store:** Select an appropriate external database for persistence (Firestore).
3. **Design Data Model:** Structure how agent state will be stored in Firestore.
4. **Implement `StateManager`:** Create a Python module to abstract Firestore interactions (save, load, delete).
5. **Integrate with ADK Agent:** Modify the ADK agent to use the `StateManager` for loading state on startup and saving state after significant actions.
6. **Verify Persistence:** Test the agent's ability to pause, restart, and resume conversations correctly.

Understanding Agent State and Context

Before diving into implementation, it's vital to differentiate between an agent's transient session memory and its persistent, long-term state.

- **Session Memory:** This typically refers to the short-term conversational history that an agent framework (like ADK) manages during a single interaction session. It's often in-memory and lost when the agent process restarts. This is sufficient for simple, one-off queries, but not for complex workflows.
- **Persistent State and Context:** This is the information we want to save and load across sessions or system restarts. It includes:
    - **Conversational Context:** The entire chat history, including user inputs and agent responses, often structured to feed back into the LLM.
    - **Agent Internal State:** Any variables, flags, or data points specific to the agent's current task or workflow. This could be the outcome of a tool call, a pending user confirmation, or the current step in a multi-step process.
    - **Tool State:** Information related to external tools the agent uses, such as an API call's response that needs to be processed later.

📌 **Key Idea:** For a long-running agent to be truly resilient, it must externalize and persist this critical state and context beyond its immediate runtime memory.

Architectural Choices for Persistence

To achieve persistence, we need an external data store. The choice of store depends on your project's specific requirements for data structure, scalability, performance, and cost.

Data Store Options

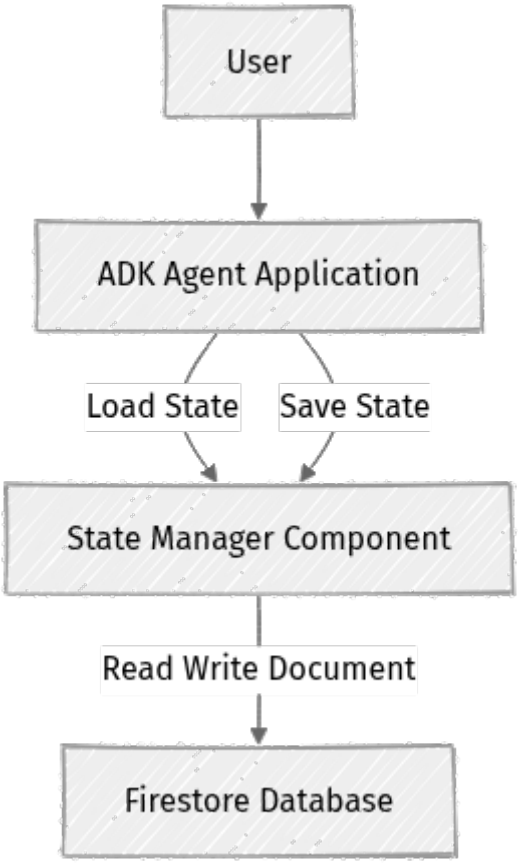
Here's a comparison of common Google Cloud options:

Option	Use Case	Pros	Cons
Firestore	Flexible, document-based state, chat history	Schemaless, highly scalable, real-time updates, fully managed	Cost can increase with reads/writes, eventually consistent reads
Cloud SQL	Structured agent state, complex relationships	ACID transactions, strong consistency, relational model	Less flexible schema, horizontal scaling can be complex
Cloud Memorystore (Redis)	Fast caching, transient session state, message queues	Extremely low latency, high throughput, simple key-value	Primarily in-memory (costly for large datasets), less durable

For this project, we'll choose **Firestore**. Its document-oriented, NoSQL nature is well-suited for storing flexible agent states and conversational histories, which often don't fit a rigid relational schema. It also offers excellent scalability and is fully managed, reducing operational overhead.

State Management Architecture

Our agent will interact with a dedicated `StateManager` component. This component will abstract away the details of the chosen database, providing a clean interface for the agent to save and load its state.



Explanation of the Flow:

1. **User Interaction:** The user sends a message to the ADK agent.
2. **Agent Application:** The ADK agent processes the message, potentially using tools or internal logic.
3. **State Manager:** Before processing or after a significant step, the agent calls the `StateManager` to load its previous state or save its current state.
4. **Firestore Database:** The `StateManager` interacts directly with Firestore, reading or writing documents that represent the agent's state and context.

Data Model for Firestore

We'll store each agent's state as a single document in a Firestore collection (e.g., `agent_states`). The document ID will be unique to each agent instance (e.g., a session ID or a user ID).

A state document might look like this:

```
{
  "session_id": "user-123",
  "last_updated": "2026-05-23T10:30:00Z",
  "conversation_history": [
    {
      "role": "user",
```

```
      "content": "What's the weather like?"
    },
    {
      "role": "agent",
      "content": "Which city are you interested in?"
    }
  ],
  "agent_variables": {
    "current_city_query": "London",
    "last_tool_call_result": {
      "tool": "weather_api",
      "status": "success",
      "data": {}
    },
    "step_in_workflow": "awaiting_city_confirmation"
  }
}
```

This structure allows for a flexible `conversation_history` array and an `agent_variables` dictionary to hold any custom data the agent needs. This approach minimizes Firestore operations by keeping all related state for a single agent instance within one document.

Step-by-Step Implementation

We'll integrate Firestore into our ADK agent.

Prerequisites

1. **Python 3.12+:** Ensure you have the latest stable Python 3.x installed.
2. **Google Cloud Project:** Ensure you have an active Google Cloud project.
3. **Firestore API Enabled:** In your Google Cloud project, navigate to "APIs & Services" -> "Enabled APIs & Services" and ensure "Cloud Firestore API" is enabled.

4. **Service Account:** For local development, create a service account key and download it. For deployment, use Google Cloud's default service account or dedicated IAM roles.
    - Go to "IAM & Admin" -> "Service Accounts" -> "Create Service Account".
    - Grant it the "Cloud Datastore User" or "Cloud Datastore Editor" role (Firestore uses the same underlying service).
    - Create a new JSON key and save it as `service_account_key.json` in your project root or a secure location.
    - Set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the path of this key file.

1. Install Firestore Client Library

First, install the Python client library for Google Cloud Firestore.

```
pip install google-cloud-firestore==2.17.0  # Checked 2024-05-23
```

⚡ **Quick Note:** The specified version `2.17.0` is current as of 2024-05-23. Always check the official `google-cloud-firestore` PyPI page for the absolute latest stable release as of your development date (2026-05-23).

2. Create a StateManager Module

Let's create `state_manager.py` to handle all interactions with Firestore. This module will encapsulate all database logic, keeping our agent code clean and focused on AI logic.

Create a file named `state_manager.py` in your project root:

```
# project_root/state_manager.py
import os
from datetime import datetime
from typing import Dict, Any
from google.cloud import firestore

class StateManager:
    """Manages persistence of agent state and conversational context using
Firestore."""

    def __init__(self, project_id: str, collection_name: str = "agent_states")
:
        """
        Initializes the StateManager.

        Args:
            project_id: Your Google Cloud Project ID.
```

```
            collection_name: The Firestore collection to store agent states.
        """
        self.db = firestore.Client(project=project_id)
        self.collection_ref = self.db.collection(collection_name)
        print(f"StateManager initialized for project '{project_id}',
collection '{collection_name}'")

    def load_state(self, agent_id: str) -> Dict[str, Any]:
        """
        Loads the agent's state from Firestore.

        Args:
            agent_id: A unique identifier for the agent instance (e.g.,
session ID).

        Returns:
            A dictionary representing the agent's state, or an empty dict if
not found.
        """
        try:
            doc_ref = self.collection_ref.document(agent_id)
            doc = doc_ref.get()
            if doc.exists:
                state = doc.to_dict()
                print(f"State loaded for agent '{agent_id}'. Last updated: {st
ate.get('last_updated')}")
                return state
            print(f"No existing state found for agent '{agent_id}'. Returning
empty state.")
            return {}
        except Exception as e:
            print(f"Error loading state for agent '{agent_id}': {e}")
            return {} # Return empty state on error

    def save_state(self, agent_id: str, state: Dict[str, Any]):
        """
        Saves the agent's current state to Firestore.

        Args:
            agent_id: A unique identifier for the agent instance.
            state: The dictionary representing the agent's current state.
        """
        try:
            state["last_updated"] = datetime.now().isoformat()
            doc_ref = self.collection_ref.document(agent_id)
            doc_ref.set(state) # Firestore automatically creates/overwrites
            print(f"State saved for agent '{agent_id}' at {state['last_updated
']}")
        except Exception as e:
            print(f"Error saving state for agent '{agent_id}': {e}")

    def delete_state(self, agent_id: str):
        """
        Deletes an agent's state from Firestore.

        Args:
            agent_id: A unique identifier for the agent instance.
        """
        try:
            doc_ref = self.collection_ref.document(agent_id)
            doc_ref.delete()
            print(f"State deleted for agent '{agent_id}'")
```

```
        except Exception as e:
            print(f"Error deleting state for agent '{agent_id}': {e}")

# Example usage (for testing, not part of agent.py)
if __name__ == "__main__":
    # Ensure GOOGLE_APPLICATION_CREDENTIALS is set in your environment
    # export GOOGLE_APPLICATION_CREDENTIALS="/path/to/your/
service_account_key.json"
    # export GOOGLE_CLOUD_PROJECT="your-gcp-project-id"

    project_id = os.getenv("GOOGLE_CLOUD_PROJECT")
    if not project_id:
        raise ValueError("Environment variable GOOGLE_CLOUD_PROJECT not set.
Please configure your GCP project ID.")

    sm = StateManager(project_id)
    test_agent_id = "test_user_session_123"

    print("\n--- Initial Load ---")
    initial_state = sm.load_state(test_agent_id)
    print(f"Initial state: {initial_state}")

    print("\n--- Saving New State ---")
    new_state = {
        "conversation_history": [
            {"role": "user", "content": "Hello there!"},
            {"role": "agent", "content": "Hi! How can I help you?"}
        ],
        "agent_variables": {
            "user_name": "Alice",
            "task_status": "greeting_complete"
        }
    }
    sm.save_state(test_agent_id, new_state)

    print("\n--- Loading State After First Save ---")
    loaded_state = sm.load_state(test_agent_id)
    print(f"Loaded state: {loaded_state}")

    print("\n--- Updating State ---")
    loaded_state["conversation_history"].append({"role": "user", "content": "W
hat time is it?"})
    loaded_state["agent_variables"]["task_status"] = "awaiting_time_query"
    sm.save_state(test_agent_id, loaded_state)

    print("\n--- Loading State After Update ---")
    updated_loaded_state = sm.load_state(test_agent_id)
    print(f"Updated loaded state: {updated_loaded_state}")

    print("\n--- Deleting State ---")
    sm.delete_state(test_agent_id)

    print("\n--- Verifying Deletion ---")
    final_state = sm.load_state(test_agent_id)
    print(f"State after deletion: {final_state}")
```

Explanation:

- **`StateManager` Class:** This class encapsulates the logic for interacting with Firestore. It's initialized with your Google Cloud `project_id` and a `collection_name` (defaulting to `agent_states`).
- **`__init__`:** Sets up the `firestore.Client`, which handles authentication using the `GOOGLE_APPLICATION_CREDENTIALS` environment variable (or instance metadata in Google Cloud environments).
- **`load_state(agent_id)`:** This method attempts to retrieve a document from the specified collection using `agent_id` as the document identifier. If the document exists, its contents are returned as a Python dictionary. If not, an empty dictionary is returned, signaling a new session. Basic error handling is included.
- **`save_state(agent_id, state)`:** This method writes the provided `state` dictionary to Firestore. It automatically adds a `last_updated` timestamp. Firestore intelligently handles creation if the document doesn't exist or overwrites it if it does, ensuring the state is always current. Error handling is included.
- **`delete_state(agent_id)`:** Provides a way to clean up an agent's state, useful for ending sessions or testing.
- **`if __name__ == "__main__":` block:** This block serves as a self-contained unit test for the `StateManager`. Running this script directly will demonstrate loading, saving, updating, and deleting states in your Firestore project. Remember to set your `GOOGLE_APPLICATION_CREDENTIALS` and `GOOGLE_CLOUD_PROJECT` environment variables before running it.

3. Integrate `StateManager` into Your ADK Agent

Now, let's modify our ADK agent (assuming you have a basic agent setup from previous chapters) to use the `StateManager`. We'll focus on how to load state when the agent starts and save state after each significant interaction.

For simplicity, we'll assume a single `AGENT_SESSION_ID` for demonstration. In a real application, this `agent_id` would typically map to a unique user session, a specific long-running task identifier, or even a channel ID.

Create a file named `my_adk_agent.py` in your project root:

```
# project_root/my_adk_agent.py
import os
from typing import Dict, Any, List
from adk.agent import Agent, AgentConfig, Message
from adk.tool import Tool
```

```
from adk.event import Event
from adk.llm import LLMConfig
# Assuming state_manager.py is in the same directory
from state_manager import StateManager

# --- Agent Configuration ---
GCP_PROJECT_ID = os.getenv("GOOGLE_CLOUD_PROJECT")
if not GCP_PROJECT_ID:
    raise ValueError("GOOGLE_CLOUD_PROJECT environment variable not set.
Please configure your GCP project ID.")

# This could be a session ID, user ID, or task ID - crucial for state
persistence
AGENT_SESSION_ID = "my-unique-agent-session-001"

# --- Define a simple tool (optional, for context) ---
class GreetingTool(Tool):
    def __init__(self):
        super().__init__(
            name="greeting_tool",
            description="A tool to greet the user and remember their name.",
            input_schema={
                "type": "object",
                "properties": {
                    "name": {"type": "string", "description": "The user's
name"},
                },
                "required": ["name"],
            },
        )

    def run(self, input: Dict[str, Any]) -> str:
        name = input.get("name", "there")
        return f"Hello, {name}! It's nice to meet you."

# --- Define the ADK Agent ---
class PersistentAgent(Agent):
    def __init__(self, config: AgentConfig, state_manager: StateManager):
        super().__init__(config)
        self.state_manager = state_manager
        self.agent_id = AGENT_SESSION_ID
# In a real app, this would be passed dynamically

        # Load initial state when the agent is initialized
        self.state = self.state_manager.load_state(self.agent_id)
        if not self.state:
            # Initialize with default values if no state was loaded (first
run)
            self.state = {
                "conversation_history": [],
                "agent_variables": {
                    "user_name": None,
                    "greeting_done": False
                }
            }
        print(f"Agent '{self.agent_id}' initialized with state: {self.state}")

    def on_init(self) -> None:
        """Called once when the agent is initialized.
        State is already loaded in __init__ for immediate availability."""
        print(f"Agent '{self.agent_id}' on_init hook called.")
        pass
```

```
    def on_message(self, message: Message) -> Event:
        """Processes an incoming message, updates state, and saves it."""
        print(f"Agent '{self.agent_id}' received message: {message.text}")

        # Add user message to conversation history
        self.state["conversation_history"].append({"role": "user", "content":
message.text})

        response_text = ""

        # Example: Use a tool and update agent variables based on current
state
        if "hello" in message.text.lower() and not self.state["agent_variables
"]["greeting_done"]:
            # In a real agent, LLM would extract the name
            name_match = "User" # For simplicity in this example

            tool_output = self.call_tool("greeting_tool", {"name": name_match}
)
            response_text = tool_output
            self.state["agent_variables"]["user_name"] = name_match
            self.state["agent_variables"]["greeting_done"] = True
        elif self.state["agent_variables"]["user_name"]:
            response_text = f"Hello again, {self.state['agent_variables']['use
r_name']}! You said: '{message.text}'"
        else:
            # Simulate an LLM response based on history (ADK's default
behavior)
            # In a real scenario, you'd use
self.llm.generate_response(messages=self.state["conversation_history"])
            response_text = f"You said: '{message.text}'. Current state
(vars): {self.state['agent_variables']}"

        # Add agent response to conversation history
        self.state["conversation_history"].append({"role": "agent",
"content": response_text})

        # Save the updated state after processing the message
        self.state_manager.save_state(self.agent_id, self.state)

        # Return the agent's response
        return Event(text=response_text)

    def on_tool_call_result(self, tool_name: str, tool_input: Dict[str, Any],
tool_output: str) -> None:
        """Called after a tool call completes.
        You might save state here if tool calls are critical checkpoints."""
        print(f"Tool '{tool_name}' called with input '{tool_input}', result:
'{tool_output}'")
        # self.state_manager.save_state(self.agent_id, self.state) # Optional
save point
        pass

# --- Main execution block ---
if __name__ == "__main__":
    # Initialize the StateManager
    state_manager = StateManager(project_id=GCP_PROJECT_ID)

    # Configure the ADK agent. Use a placeholder LLM config.
    agent_config = AgentConfig(
        llm=LLMConfig(
```

```
            model_name="gemini-pro", # Use an appropriate model for your
region/project
            temperature=0.7,
        ),
        tools=[GreetingTool()], # Register our custom tool
        # Add other configurations as needed
    )

    print("\n--- Simulating first session (Agent Instance 1) ---")
    # Instantiate the persistent agent
    persistent_agent = PersistentAgent(config=agent_config, state_manager=stat
e_manager)

    # Simulate an incoming message
    first_message = Message(text="Hello, my name is User!")
    persistent_agent.on_message(first_message)

    second_message = Message(text="How are you today?")
    persistent_agent.on_message(second_message)

    print("\n--- Simulating agent restart (Agent Instance 2, same session ID)
---")
    # In a real scenario, the old process would die, and a new one would
start.
    # We simulate this by creating a new agent instance with the same
AGENT_SESSION_ID.
    restarted_agent = PersistentAgent(config=agent_config, state_manager=state
_manager)
    third_message = Message(text="Can you remind me what my name is?")
    restarted_agent.on_message(third_message)

    print("\n--- Simulating another message in the restarted session ---")
    fourth_message = Message(text="What else can you do?")
    restarted_agent.on_message(fourth_message)

    print("\n--- Final state after all interactions (retrieved directly from
DB) ---")
    final_state_from_db = state_manager.load_state(AGENT_SESSION_ID)
    print(f"Final state from DB: {final_state_from_db}")

    # Optional: Clean up state for next run or if the session is truly over
    # print("\n--- Deleting agent state for cleanup ---")
    # state_manager.delete_state(AGENT_SESSION_ID)
```

Explanation of Agent Integration:

1. **`StateManager` Import:** We import our `StateManager` class into the main agent file.

2. **`PersistentAgent` Class:**
    - **`__init__`:** Takes a `state_manager` instance. Crucially, it calls `self.state_manager.load_state(self.agent_id)` right at the start. If no state is found (e.g., first interaction for this `agent_id`), it initializes a default empty state. This ensures the agent always starts with its last known context if available, or a clean slate otherwise.
    - **`on_message`:**
        - Appends the incoming user message to `self.state["conversation_history"]`.
        - Performs some example logic (e.g., calling a `greeting_tool` and updating `agent_variables`). Notice how `self.state["agent_variables"]["greeting_done"]` is used to prevent repeated greetings, demonstrating stateful logic.
        - Appends the agent's response to `self.state["conversation_history"]`.
        - **`self.state_manager.save_state(self.agent_id, self.state)`:** This is the critical line. After every significant interaction (here, after processing a message and formulating a response), the agent's entire `self.state` dictionary is saved to Firestore. This makes the agent's progress durable.
3. **Main Execution Block:**
    - Initializes the `StateManager` with your GCP project ID.
    - Configures the ADK agent with an LLM and our `GreetingTool`.
    - **Simulates First Session:** `persistent_agent` handles initial messages, saving its state after each.
    - **Simulates Restart:** A new `PersistentAgent` instance (`restarted_agent`) is created using the same `AGENT_SESSION_ID`. When `restarted_agent` initializes, it automatically loads the state that `persistent_agent` had saved to Firestore, demonstrating the core resume capability.
    - Further messages are processed by `restarted_agent`, and its updated state is saved, proving continuity.
    - Finally, the state is loaded directly from the database to show its final persistent form.

Testing & Verification

To confirm your persistent agent is working as expected, follow these verification steps:

1. **Set Environment Variables:** Open your terminal and set the required environment variables:

```
    export GOOGLE_APPLICATION_CREDENTIALS="/path/to/your/
service_account_key.json"
    export GOOGLE_CLOUD_PROJECT="your-gcp-project-id"
```

```
Replace `/path/to/your/service_account_key.json` with the actual path to your
downloaded service account key, and `your-gcp-project-id` with your Google
Cloud project ID.
```

1. **Run the Agent Script:** Execute the agent application:

```
    python my_adk_agent.py
```

1. **Observe Console Output:** Carefully review the output in your terminal:
    - Look for `StateManager initialized...` messages at the start.
    - When the first agent instance starts, you should see `No existing state found...` initially, followed by `State saved...` messages after each message processing.
    - When the "restarted" agent instance starts, you should see `State loaded for agent 'my-unique-agent-session-001'. Last updated:...` messages, indicating it successfully retrieved the previous state.
    - Verify that the responses from the `restarted_agent` acknowledge previous interactions, for example, by recalling the user's name.
    - The "Final state from DB" output should show the complete `conversation_history` and `agent_variables` from all simulated interactions.

2. **Inspect Firestore Console:**
    - Go to the Google Cloud Console and navigate to "Firestore Database".
    - You should see a collection named `agent_states` (or whatever `collection_name` you used in `StateManager`).
    - Inside this collection, you'll find a document with the ID `my-unique-agent-session-001`.
    - Inspect the document's content. You should see the `conversation_history`, `agent_variables`, and `last_updated` fields reflecting the latest state of your agent after all interactions.
    - **Verification Challenge:** Try deleting the `my-unique-agent-session-001` document in the Firestore console, then run `my_adk_agent.py` again. Observe that the agent now starts from a fresh state (`No existing state found...`) and the conversation history begins anew. This confirms the persistence mechanism is working as intended.

Production Considerations

Building a robust persistence layer requires more than just saving and loading data. Consider these factors for a production environment:

- **Concurrency and Race Conditions:** If multiple instances of the same agent_id could write state concurrently (e.g., if you have multiple user interactions happening simultaneously for the same agent), you need to handle race conditions. Firestore offers transactions for atomic read-modify-write operations, which are crucial for ensuring data consistency.
- **Error Handling and Retries:** What happens if the Firestore service is temporarily unavailable? Implement robust `try-except` blocks around `db` operations and consider retry mechanisms (e.g., with exponential backoff) to make your agent resilient to transient network issues or service outages.

- **Security (IAM):** Never embed service account keys directly into your deployed application code.
    - **Google Cloud Run/GKE:** Use the default service account associated with the compute instance or container. Grant this service account the minimum necessary IAM roles (e.g., `roles/datastore.user` for read/write access to Firestore).
    - **Local Development:** Use the `GOOGLE_APPLICATION_CREDENTIALS` environment variable, which points to your service account key file.
    - Ensure any sensitive information stored in the agent's state (e.g., API keys, user PII) is encrypted, either before storage or by using a service like Google Secret Manager.
- **Performance and Scalability:**
    - **Indexing:** For complex queries on your state documents (e.g., searching for agents by specific `agent_variables`), ensure appropriate Firestore indexes are configured to maintain performance.
    - **Document Size:** Keep individual state documents reasonably sized (Firestore has a 1MB limit per document). If your conversation history grows excessively large, consider archiving older parts or storing them in a separate, cheaper storage solution like Cloud Storage.
    - **Read/Write Operations:** Be mindful of Firestore's pricing model, which charges per document read, write, and delete. Optimize your `save_state` calls to only persist when absolutely necessary (e.g., after a tool call, or a critical user confirmation, not after every single token generated by the LLM).
- **Schema Evolution:** As your agent's capabilities grow, its internal state structure might change. Plan for schema migrations or design your `agent_variables` to be flexible enough to handle new fields without breaking existing state documents.

Common Issues & Solutions

1. **`google.auth.exceptions.DefaultCredentialsError`:**
    - **Issue:** The Firestore client cannot find your Google Cloud credentials.
    - **Solution:** Ensure the `GOOGLE_APPLICATION_CREDENTIALS` environment variable points to your service account key JSON file, or that your execution environment (e.g., Cloud Run) has a service account with appropriate permissions. Double-check the `project_id` passed to `firestore.Client`.
2. **State Not Persisting/Loading Correctly:**
    - **Issue:** After restarting the agent, the state doesn't reflect previous interactions.
    - **Solution:**
        - Verify the `agent_id` is consistent across saves and loads. A common mistake is generating a new ID on restart, leading to a new state document being created.
        - Check the Firestore console to confirm the document is being written and updated as expected.
        - Ensure `save_state` is called at all necessary checkpoints within your agent's logic.
        - Check for any errors during `save_state` that might prevent the write (review the `try-except` blocks' output).
3. **Performance Bottlenecks with Large State:**
    - **Issue:** Agent becomes slow when loading or saving very large `conversation_history` or `agent_variables`.
    - **Solution:**
        - **Optimize data structure:** Only store essential information. For `conversation_history`, consider storing only a summary or the last `N` turns in the main state document, and archiving older turns in a separate collection or Cloud Storage.
        - **Consider caching:** For frequently accessed but slowly changing parts of the state, implement an in-memory cache within the agent instance to reduce database calls.
        - **Review Firestore usage:** Check for inefficient queries or excessive reads/writes.

4. **Firestore Permissions Denied:**
    - **Issue:** Your agent's service account lacks the necessary IAM permissions to read/write to Firestore.
    - **Solution:** In the Google Cloud Console, navigate to "IAM & Admin" -> "Service Accounts" and ensure the service account used by your agent has at least the `Cloud Datastore User` role, or a custom role with `datastore.entities.get`, `datastore.entities.update`, `datastore.entities.create`, and `datastore.entities.delete` permissions.

Summary & Next Step

You've successfully designed and implemented a robust context preservation mechanism for your ADK agent. By integrating with Google Firestore, your agent can now:

- Load its entire operational state and conversational history upon startup.
- Save its current state and context after critical interactions or workflow steps.
- Effectively pause and resume its activities, maintaining continuity across sessions or system restarts.

This is a fundamental step towards building truly resilient and production-ready AI agents. With persistent state, your agent can handle interruptions, long-running tasks, and deliver a more consistent user experience. This durable memory is what elevates an agent from a stateless chatbot to a capable, long-term assistant.

In the next chapter, we will take our agent a step closer to production readiness by **containerizing it using Docker**, preparing it for scalable and portable deployment to Google Cloud services like Cloud Run or GKE.

References

1. **Google Cloud Firestore Documentation:** The official documentation for Google's NoSQL document database. https://cloud.google.com/firestore/docs
2. **Google Cloud Python Client for Firestore:** Official repository and documentation for the Python library. https://googleapis.dev/python/firestore/latest/index.html
3. **Google Cloud IAM Documentation:** Information on managing identities and access control in Google Cloud. https://cloud.google.com/iam/docs
4. **Google ADK (Agent Development Kit) Examples:** While a specific official ADK documentation URL wasn't found as of 2026-05-23, community and training examples provide insight. https://github.com/Sri-Krishna-V/awesome-adk-agents
5. **PyPI: google-cloud-firestore:** The Python Package Index page for the Firestore client library, useful for checking the latest stable version. https://pypi.org/project/google-cloud-firestore/

CHAPTER 05

Enhancing Agent Intelligence with Tools and Multi-Step Workflows

Enhancing Agent Intelligence with Tools and Multi-Step Workflows

To build truly capable AI agents, mere conversational abilities are not enough. Agents must interact with the real world, access dynamic information, and perform actions beyond generating text. This is precisely where **tools** become indispensable. Tools are external functions or APIs that an agent can invoke to perform specific tasks, retrieve real-time data, or integrate with other systems. Imagine an agent that can not only chat about the weather but also fetch the current weather forecast for any city.

This chapter focuses on empowering our long-running ADK agent with such external tools and orchestrating these tools within **multi-step workflows**. We will design and implement a practical tool, integrate it into our agent, and demonstrate how the agent autonomously decides when and how to use it to achieve a goal. By the end of this chapter, your agent will exhibit more complex reasoning and action, leveraging its persistent state to manage intricate tasks over time, ensuring it never loses context.

Project Overview: From Stateful to Tool-Augmented

In previous chapters, we established a foundational ADK agent capable of maintaining conversational context across sessions using a durable state manager. While crucial, this agent was limited to its internal knowledge. This chapter marks a significant upgrade:

- **Intelligence Boost:** The agent gains the ability to "act" in the real world by calling external functions.
- **Multi-Step Actions:** We enable the agent to break down complex requests into a sequence of tool calls and reasoning steps.
- **Enhanced Utility:** The agent can now solve problems requiring up-to-date information or specific external functionalities.

The outcome is an agent that can intelligently interact with external systems, process their responses, and continue a coherent conversation, all while leveraging its persistent memory to manage the workflow's state.

Tech Stack for Tool Integration

This chapter continues to build upon our core Python and Google Cloud stack, with specific emphasis on the `google-generative-ai` library for its function-calling capabilities.

- **Python 3.12+**: The primary language for our agent and tools.
- **`google-generative-ai` (v0.11.0 as of 2026-05-23)**: This library facilitates interaction with Google's Gemini models, which provide the underlying large language model (LLM) and its function-calling feature—a key component of Google's Agent Development Kit (ADK).
- **Google Cloud Firestore**: Our chosen durable state store for persisting agent context and conversation history.
- **`python-dotenv`**: For managing environment variables securely.

Milestones for This Chapter

To incrementally build this capability, we will follow these steps:

1. **Define a Tool Function**: Create a standard Python function that simulates an external action (e.g., fetching weather). This function will serve as our "tool."
2. **Register the Tool**: Integrate this function with our `google-generative-ai` model, making it aware of the tool's existence and capabilities.
3. **Handle Tool Invocations**: Modify the agent's message processing loop to detect when the LLM requests a tool call, execute the tool, and feed its output back to the LLM.
4. **Verify Persistent Tool State**: Ensure that tool calls and their outputs are correctly saved and loaded as part of the agent's conversational history.
5. **Test Multi-Step Workflows**: Validate that the agent can use the tool and reason effectively through a multi-turn interaction.

Planning & Design: Agent-Tool Interaction

An ADK agent equipped with tools operates on an advanced "Sense, Think, Act" loop. The agent receives a user prompt, reasons about the optimal course of action (which often involves using a tool), executes the tool, processes the tool's output, and then formulates a response or continues the workflow.

The Role of Tools

Tools are essentially well-defined wrappers around external functions or APIs. When you define a tool for an ADK agent using Google's generative AI models, you provide essential metadata:

1. **Name**: A unique identifier for the tool, typically derived directly from the Python function name. The LLM uses this name to refer to the tool.
2. **Description**: A clear explanation of what the tool does, its purpose, and when it should be used. This description, often extracted from the function's docstring, is crucial for the underlying LLM to understand when to invoke the tool.
3. **Schema**: A specification for the tool's input parameters, inferred from Python type hints and parameter names. This helps the LLM generate correct arguments when calling the tool.

Multi-Step Workflows

A multi-step workflow involves the agent performing a sequence of actions, potentially using multiple tools, to achieve a complex goal. The agent's persistent memory (established in previous chapters) is critical here, allowing it to remember the progress, results of previous tool calls, and overall conversational context across steps. This ensures continuity and avoids re-asking for information.

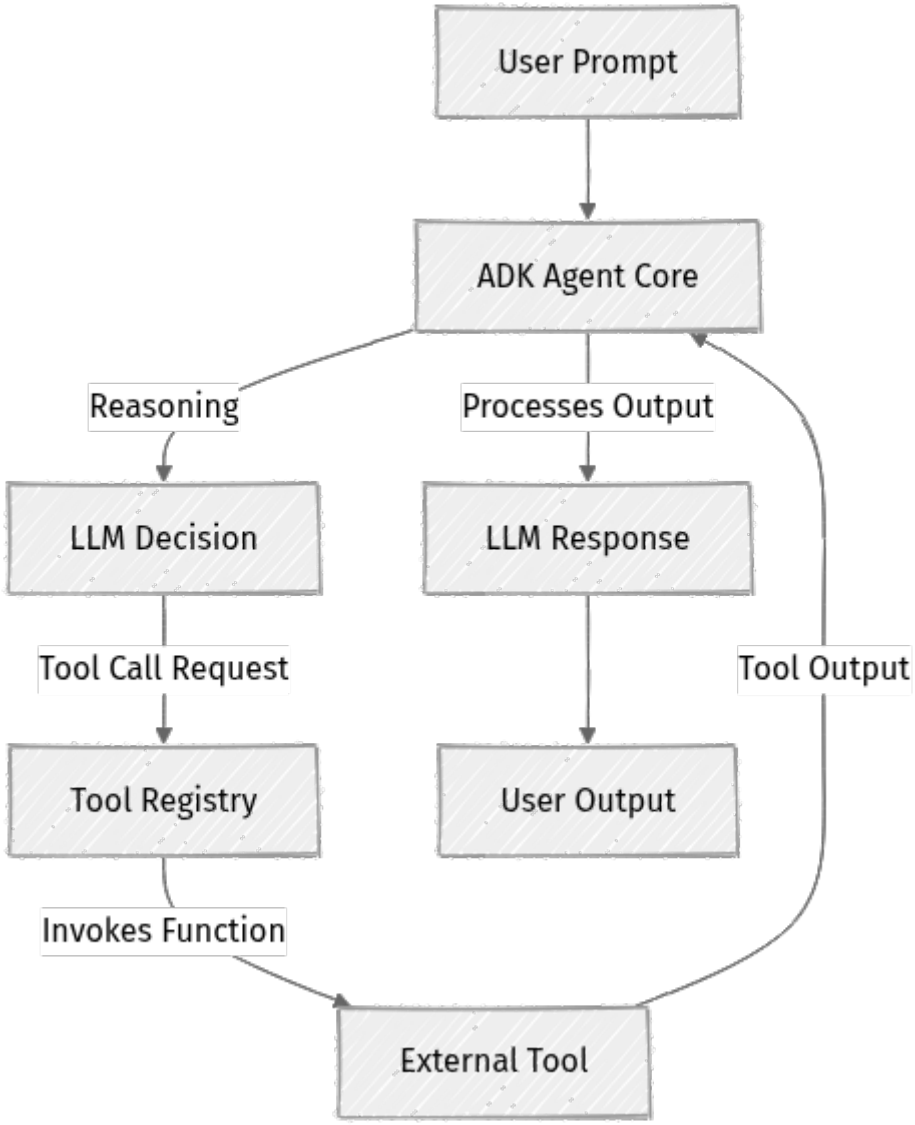
For this chapter, we'll implement a simple `get_current_weather` tool. The agent will be able to:

1. Receive a user request about weather in a specific city.
2. Identify that it needs to use the `get_current_weather` tool.
3. Extract the city name and desired unit from the user's request.
4. Call the tool with the correct arguments.
5. Receive the weather data (mocked for now).
6. Formulate a natural language response based on the weather data, integrating it into the conversation.

This flow, while simple, demonstrates the core principles of tool usage and lays the foundation for more complex, chained workflows.

Architecture of Agent-Tool Interaction

The following diagram illustrates how the agent interacts with a tool and leverages its internal reasoning capabilities.



Explanation:

- **User Prompt**: The initial input provided by the user.
- **ADK Agent Core**: Our Python application, responsible for orchestrating the overall interaction.
- **LLM Decision**: The core intelligence, powered by a large language model (e.g., Google's Gemini), which analyzes the prompt and available tools to decide the next action.
- **Tool Call Request**: If the LLM determines a tool is needed, it generates a structured request specifying the tool's name and arguments.
- **Tool Registry**: A component within our ADK setup that maps tool names to their actual Python functions.

- **External Tool Function**: The actual implementation of the tool, which might call an external service (like a weather API) or perform a local computation.
- **Tool Output (JSON)**: The structured result returned by the external tool, typically in JSON format.
- **LLM Response**: The LLM processes the tool's output, integrates it with the ongoing conversation, and generates a coherent response or determines the next step in a multi-turn workflow.
- **User Output**: The final response presented to the user.

Step-by-Step Implementation

We will continue building on our existing ADK agent structure, which includes the `ADKAgent` class and a `StateManager`. We need to define our tool, register it, and then modify the agent's interaction loop to handle tool calls and their subsequent processing.

Prerequisites:

- Ensure you have a Python 3.12+ environment set up.
- Your Google Cloud project and credentials are configured (as per Chapter 1).
- The `google-generative-ai` library is installed. As of 2026-05-23, the latest stable version of `google-generative-ai` is `0.11.0`.
- The `google-cloud-firestore` library is installed for the state manager.
- The ADK context here refers to using the Gemini API's function calling capabilities, which is a core part of building agents with Google's models.

1. Define a Simple Tool Function

First, let's create a Python function that simulates fetching weather data. For a real-world scenario, this would involve making an HTTP request to a weather API.

Create a new file `tools.py` in your project root:

```
# project_root/tools.py

import json

def get_current_weather(location: str, unit: str = "celsius"):
    """
    Fetches the current weather for a given location.

    Args:
        location (str): The city and state, e.g., "San Francisco, CA".
        unit (str): The unit of temperature, either "celsius" or "fahrenheit".
Defaults to "celsius".
```

```
    Returns:
        str: A JSON string containing weather information (temperature,
description, unit, location).
    """
    print(f"DEBUG: Calling get_current_weather for {location} in {unit}")
    # In a real application, this would call an external weather API.
    # For demonstration, we'll return mock data.

    mock_weather_data = {
        "San Francisco, CA": {"temperature": 15, "description": "Cloudy", "uni
t": "celsius"},
        "New York, NY": {"temperature": 22, "description": "Sunny", "unit": "c
elsius"},
        "London, UK": {"temperature": 10, "description": "Rainy", "unit": "cel
sius"},
    }

    if unit.lower() == "fahrenheit":
        # Create a fahrenheit version of mock data for conversion
        fahrenheit_data = {}
        for loc, data in mock_weather_data.items():
            temp_f = round(data["temperature"] * 9/5 + 32)
            fahrenheit_data[loc] = {"temperature": temp_f, "description":
data["description"], "unit": "fahrenheit"}
        mock_weather_data = fahrenheit_data

    weather = mock_weather_data.get(location, {
        "temperature": "N/A",
        "description": "Weather data not available",
        "unit": unit
    })
    weather["location"] = location # Add location to the output for clarity

    return json.dumps(weather) # Return as JSON string for consistency with
tool outputs
```

Explanation:

- The `get_current_weather` function takes `location` and an optional `unit` as arguments.
- Its docstring is crucial: the LLM uses this description to understand the tool's purpose and how to use it.
- For simplicity, it returns mock data. In a production system, this would integrate with a service like OpenWeatherMap or a similar API.
- The output is returned as a JSON string. This structured format is ideal for the LLM to parse and interpret consistently.

2. Register the Tool and Handle Tool Calls

Now, we need to make this tool available to our agent and teach our agent how to execute it when the LLM decides. We'll modify `adk_agent.py` to import this tool, register it with the model, and implement the logic for handling tool call requests and feeding their outputs back to the LLM.

Update `adk_agent.py`:

```
# project_root/adk_agent.py

import google.generativeai as genai
import os
from dotenv import load_dotenv
import json
import uuid
import time
import google.generativeai.types as glm # Import glm for explicit typing and
FunctionResponse

from state_manager import StateManager # Assuming this is from Chapter 3/4
from tools import get_current_weather # Import our new tool

load_dotenv()

# Configure the Gemini API client
# This implementation uses the google-generative-ai library, which provides
the underlying
# model capabilities that an ADK agent would leverage for function calling.
# Please refer to the official Gemini API documentation for the most up-to-
date information:
# https://ai.google.dev/docs/gemini_api_overview
genai.configure(api_key=os.getenv("GEMINI_API_KEY"))

class ADKAgent:
    def __init__(self, agent_id: str, state_manager: StateManager):
        self.agent_id = agent_id
        self.state_manager = state_manager
        self.model = genai.GenerativeModel(
            'gemini-1.5-flash',
            tools=[get_current_weather] # Register the tool here
        )
        # Load existing history or start new
        self.history = self.state_manager.load_context(self.agent_id) or []
        self.chat_session = self.model.start_chat(history=self.history)
        print(f"Agent {self.agent_id} initialized with {len(self.history)}
messages in history.")

    def _save_state(self):
        """Saves the current chat history to the state manager."""
        self.state_manager.save_context(self.agent_id, self.chat_session.histo
ry)
        print(f"Agent {self.agent_id} state saved.")

    def process_message(self, message: str) -> str:
        """
        Processes a user message, interacts with the model, and handles tool
calls.
        """
        try:
            # Send the user message to the chat session. This automatically
adds it to history
            # and gets the model's first response, which might include tool
calls.
            response = self.chat_session.send_message(message)

            # Loop to handle multi-turn tool interactions until the model
```

```
gives a final text response
            while True:
                if response.tool_calls:
                    print(f"DEBUG: Model requested tool calls: {response.tool_
calls}")
                    tool_outputs = []
                    for tool_call in response.tool_calls:
                        tool_name = tool_call.function.name
                        tool_args = {k: v for k, v in
tool_call.function.args.items()}

                        print(f"DEBUG: Invoking tool '{tool_name}' with args:
{tool_args}")

                        # Dynamically call the tool function based on its name
                        if tool_name == "get_current_weather":
                            raw_output = get_current_weather(**tool_args)
                            parsed_output = json.loads(raw_output)
# Parse the JSON string

# Wrap the output in glm.FunctionResponse and then in glm.ToolOutput
                            tool_outputs.append(
                                glm.ToolOutput(
                                    function_response=glm.FunctionResponse(
                                        name=tool_name,
                                        response=parsed_output # Pass the
parsed dictionary
                                    )
                                )
                            )
                        else:
                            error_msg = f"ERROR: Unknown tool requested: {tool
_name}"
                            print(error_msg)
                            tool_outputs.append(
                                glm.ToolOutput(
                                    function_response=glm.FunctionResponse(
                                        name=tool_name,
                                        response={"error": error_msg}
                                    )
                                )
                            )


# Send tool outputs back to the model. The model will then generate a response
                    # based on these outputs and its ongoing understanding of
the conversation.
                    # This new response might be a final text response or
another tool call.
                    response = self.chat_session.send_message(tool_outputs)
                elif response.text:
                    # If the model provides a text response, it's the final
answer for this turn.
                    final_response_text = response.text
                    break
                else:
                    # Handle cases where the model returns neither text nor
tool calls
                    print("WARNING: Model response contained neither text nor
tool calls.")
                    final_response_text = "I received an unexpected response
from the model. Please try again."
```

```
                    break

            self._save_state() # Save state after processing message and
potential tool calls
            return final_response_text

        except Exception as e:
            print(f"An error occurred: {e}")
            # In a production system, you'd want more robust error handling,
            # potentially logging the full traceback and user message.
            return "I encountered an error while processing your request.
Please try again."

# Example usage (for testing purposes)
if __name__ == "__main__":
    # Ensure your state manager is correctly configured (e.g., using
Firestore)
    # For a quick local test, you might use an in-memory state manager if you
haven't set up Firestore yet.
    # from state_manager_inmemory import InMemoryStateManager

# state_manager = InMemoryStateManager() # For local testing without cloud
setup

    # Assuming FirestoreStateManager is implemented from previous chapters
    from state_manager_firestore import FirestoreStateManager
    state_manager = FirestoreStateManager(project_id=os.getenv("GCP_PROJECT_ID
"))

    # To simulate a long-running session, use a fixed agent ID
    session_id = "test_agent_with_tools_123"
    agent = ADKAgent(session_id, state_manager)

    print("\nAgent ready. Type 'exit' to quit.")

    while True:
        user_input = input("You: ")
        if user_input.lower() == 'exit':
            print("Exiting agent session.")
            break

        response = agent.process_message(user_input)
        print(f"Agent: {response}")
        time.sleep(0.1) # Small delay for readability
```

Key Changes and Explanations:

- **`import google.generativeai.types as glm`**: We import `glm` to explicitly use `glm.ToolOutput` and `glm.FunctionResponse` for clearer code and type safety when handling tool results.
- **`tools=[get_current_weather]`**: When initializing `genai.GenerativeModel`, we pass a list of our tool functions to the `tools` argument. The Gemini model automatically inspects the function signatures and docstrings to understand how and when to use them.

- **`while True:` loop**: The agent now includes a loop to handle potential multi-turn tool interactions. The model might call a tool, process its output, and then decide to call another tool before providing a final text response. This loop ensures all necessary tool calls are made until a text response is generated.
- **`if response.tool_calls:`**: After sending a message, the model's response might include `tool_calls`. This indicates that the model has decided to invoke one or more of the registered tools.
- **Dynamic Tool Invocation**: We check the `tool_name` and call the corresponding Python function (e.g., `get_current_weather(**tool_args)`). The `**tool_args` unpacks the dictionary of arguments into keyword arguments for the function.
- **Sending Tool Outputs Back (`glm.ToolOutput` with `glm.FunctionResponse`)**: This is critical. After executing a tool, its output must be parsed (if JSON string) and wrapped in a `glm.FunctionResponse` object, which is then embedded within a `glm.ToolOutput` object. This `glm.ToolOutput` is then sent back to the `chat_session` using `self.chat_session.send_message(tool_outputs)`. This allows the LLM to process the tool's result and continue the conversation or workflow.
    - `raw_output = get_current_weather(**tool_args)`: Captures the JSON string from our tool.
    - `parsed_output = json.loads(raw_output)`: Converts the JSON string into a Python dictionary, which is the expected format for `glm.FunctionResponse`.
    - `glm.ToolOutput(function_response=glm.FunctionResponse(name=tool_name, response=parsed_output))`: Correctly structures the tool's output for the LLM.
- **`_save_state()`**: The agent's state (including the new messages and tool interactions) is saved after each `process_message` call, ensuring persistence across sessions.

3. Update `state_manager_firestore.py` (if using Firestore)

Ensure your `state_manager_firestore.py` correctly handles serializing and deserializing the `chat_session.history`, which can now include `glm.Content` objects related to tool calls and tool outputs. The `genai.to_dict()` helper function is the most reliable way to convert these complex objects into a Firestore-compatible dictionary.

Modify `state_manager_firestore.py`:

```
# project_root/state_manager_firestore.py

from google.cloud import firestore
import json
from typing import List, Dict, Any
import google.generativeai as genai # Added import for genai.to_dict

class FirestoreStateManager:
    def __init__(self, project_id: str, collection_name: str = "agent_states")
:
        self.db = firestore.Client(project=project_id)
        self.collection_ref = self.db.collection(collection_name)
        print(f"FirestoreStateManager initialized for project {project_id},
collection {collection_name}")

    def save_context(self, agent_id: str, history: List[Dict[str, Any]]):
        """
        Saves the agent's full conversational history, including tool calls
and outputs,
        to Firestore.
        """
        doc_ref = self.collection_ref.document(agent_id)

        serializable_history = []
        for message in history:
            # Use genai.to_dict to properly serialize glm.Content objects
(which include tool calls/outputs)
            serializable_history.append(genai.to_dict(message))

        try:
            doc_ref.set({"history": serializable_history, "last_updated": fire
store.SERVER_TIMESTAMP})
            print(f"Context for agent {agent_id} saved to Firestore.")
        except Exception as e:
            print(f"Error saving context for agent {agent_id}: {e}")
            raise

    def load_context(self, agent_id: str) -> List[Dict[str, Any]] | None:
        """
        Loads the agent's conversational history from Firestore.
        Returns None if no history is found.
        """
        doc_ref = self.collection_ref.document(agent_id)
        try:
            doc = doc_ref.get()
            if doc.exists:
                data = doc.to_dict()
                history = data.get("history", [])
                print(f"Context for agent {agent_id} loaded from Firestore.")
                # The genai library's start_chat method is robust; it can
often accept
                # a list of dictionaries that represent the history,
automatically converting
                # them back to its internal Content objects.
                return history
            else:
                print(f"No context found for agent {agent_id}.")
                return None
```

```
        except Exception as e:
            print(f"Error loading context for agent {agent_id}: {e}")
            return None
```

Note on `genai.to_dict()` and Firestore:

The `google-generative-ai` library's `chat_session.history` contains `glm.Content` objects, which are complex types representing various parts of a conversation, including user prompts, model responses, tool calls, and tool outputs. The `genai.to_dict()` function ensures these objects are correctly converted into a dictionary structure that Firestore can store. When loading, Firestore returns these as dictionaries, and the `genai.GenerativeModel.start_chat(history=...)` method is generally robust enough to accept a list of these dictionaries, converting them back into its internal `Content` objects.

Testing & Verification

Now, let's run our agent and verify that it correctly uses the tool and handles multi-step interactions, maintaining context.

1. **Ensure Environment Variables are Set:** Before running, set your API key and GCP project ID.

```
export GEMINI_API_KEY="YOUR_GEMINI_API_KEY"
export GCP_PROJECT_ID="your-gcp-project-id"
```

```
(Replace placeholders with your actual keys/IDs).
```

1. **Run the Agent:** Execute the main agent script from your project root.

```
python adk_agent.py
```

1. **Interact with the Agent (Verification Scenarios):**

   Scenario 1: Simple Tool Invocation

   - **You:** What's the weather like in New York, NY?
   - **Expected Agent Output:** You should observe `DEBUG: Model requested tool calls...` and `DEBUG: Invoking tool 'get_current_weather'...` messages in your terminal. The agent's final response should be a natural language summary of the mock weather data: "The weather in New York, NY is 22 degrees Celsius and Sunny."
   - **Verification:** Confirm that the agent correctly identified the need for the tool, extracted the location, called the `get_current_weather` function (which prints a debug message), and then used the mock data to formulate its response.

   Scenario 2: Tool Invocation with Specific Units

   - **You:** What's the temperature in London, UK in Fahrenheit?
   - **Expected Agent Output:** Similar debug messages, but crucially, the `unit` argument passed to the tool should be "fahrenheit". The final response should reflect the converted temperature: "The weather in London, UK is 50 degrees Fahrenheit and Rainy."
   - **Verification:** Check that the `unit` parameter was correctly extracted and passed to the tool, and the conversion logic was applied.

   Scenario 3: Multi-Step Reasoning with Persistence This scenario highlights the agent's ability to reason based on tool output and maintain context.

   - **You:** Is it a good day for a picnic in San Francisco?
   - **Expected Agent Output:** The agent should first call the `get_current_weather` tool for "San Francisco, CA". Then, based on the description ("Cloudy") and temperature (15 degrees Celsius) from the tool output, it should infer if it's "good for a picnic". A reasonable response might be: "The weather in San Francisco, CA is 15 degrees Celsius and Cloudy. It might be a bit cool and cloudy for a picnic."
   - **Verification:** Ensure the tool was called and the agent's subsequent response demonstrates reasoning based on the tool's output, not just a direct quote.

Verification of Persistence (Pause/Resume):

- Run the agent, interact with it through one or more weather questions, then type `exit` to quit.
- Restart the agent with the same *`session_id`*.
- Ask a follow-up question that relies on previous context, e.g., "What about in London?" (if you previously asked about New York). The agent should remember the context and potentially infer the need for the tool without explicitly being told "weather". You should see `Agent {agent_id} initialized with X messages in history.` confirming history was loaded from Firestore.

Production Considerations

Integrating tools into your AI agent significantly enhances its capabilities but also introduces new considerations for building a robust, production-ready system.

1. Security of Tool Access

- **API Key Management**: Any external APIs called by your tools (e.g., actual weather APIs, database credentials) must have their API keys and secrets securely managed. **Never hardcode these.** Use services like Google Secret Manager to store and retrieve them at runtime.
- **Input Validation**: Tools should rigorously validate their inputs. The LLM might occasionally hallucinate arguments or provide malformed data. Your tool functions must handle unexpected inputs gracefully to prevent errors, crashes, or security vulnerabilities (e.g., SQL injection if interacting with a database directly).
- **Permissions**: Implement the principle of least privilege. Ensure the service account running your agent only has the minimum necessary permissions to call the external APIs or access resources used by its tools.

2. Error Handling and Resilience

- **Tool Failures**: External APIs can fail due to network issues, rate limits, invalid requests, or service outages. Your agent needs to gracefully handle these failures.
    - Wrap all external tool calls in `try-except` blocks.
    - Tools should return informative error messages or structured error objects.
    - The agent should be programmed to acknowledge tool errors and potentially retry (with exponential backoff), offer alternatives, or inform the user.
- **Timeouts**: External API calls can be slow. Implement strict timeouts for all HTTP requests made by your tools to prevent the agent from hanging indefinitely, impacting user experience and resource consumption.

3. Observability and Monitoring

- **Logging Tool Calls**: Implement comprehensive logging for every tool invocation, including its arguments, execution status (success/failure), and output. This is critical for debugging, understanding agent behavior, auditing, and compliance. Replace basic `print` statements with a structured logger (e.g., Python's `logging` module, integrated with Google Cloud Logging).
- **Performance Metrics**: Monitor the latency and success rate of your tool calls. Slow tools can significantly degrade the agent's responsiveness and overall user experience.
- **Cost Tracking**: If tools incur costs (e.g., paid APIs, serverless function invocations), track their usage to manage expenditure and identify potential cost optimizations.

4. Scalability

- **Rate Limits**: Be acutely aware of rate limits on external APIs. If your agent scales to handle many concurrent users, your tools might hit these limits. Implement robust retry mechanisms with exponential backoff and consider caching strategies for frequently accessed data.
- **Stateless Tools**: Design tools to be stateless where possible. The agent's core state manager should handle conversational context and workflow state, not the tools themselves. Tools should perform their specific function and return a result without retaining memory of previous calls.

Common Issues & Solutions

1. **Agent Doesn't Use the Tool or Hallucinates Arguments:**
    - **Issue**: The LLM might ignore your registered tool, or invent arguments that don't match your tool's schema, leading to `KeyError` or unexpected behavior.
    - **Solution**:
        - **Improve Tool Description**: Ensure the tool's docstring is extremely clear and precise about what the tool does, its parameters, and when it should be used. Think of it as an instruction manual for the LLM. Be explicit about argument types and examples.
        - **Provide Few-Shot Examples**: In your system prompt or initial conversation turns, you can provide examples of how the tool should be used, a technique called few-shot prompting.
        - **Schema Accuracy**: Double-check that the Python function's type hints and parameter names (which the model uses to infer schema) are correct and unambiguous.
2. **Tool Output Not Processed Correctly:**
    - **Issue**: The tool returns data, but the agent's subsequent response doesn't seem to incorporate it meaningfully, or misinterprets it.
    - **Solution**:
        - **Structured Output**: Ensure your tool consistently returns well-structured data, ideally JSON. The LLM is significantly better at parsing well-formatted JSON than free-form text.
        - **Clear Tool Output Description**: If the tool output is complex, consider adding a "tool output description" within your prompt to guide the LLM on how to interpret it.
        - **Follow-up Prompting**: If the agent struggles, you can explicitly prompt it to "Summarize the data you just retrieved from the weather tool and tell me if it's good for a picnic."

3. **Agent Gets Stuck in an Infinite Loop (e.g., repeatedly calling a tool):**
    - **Issue**: The agent might call a tool, get an unexpected output, and then decide to call the same tool again in a loop without making progress. This can lead to high costs and poor user experience.
    - **Solution**:
        - **Clear Tool Success/Failure States**: Ensure tool outputs clearly indicate success or failure. The LLM should be able to differentiate.
        - **Contextual Reasoning**: The LLM should learn from the conversation history (which includes previous tool calls and outputs) not to repeat failed actions. If your history is not being saved/loaded correctly, this can exacerbate the problem.
        - **Max Tool Calls Safeguard**: In production, implement a safeguard to limit the number of consecutive tool calls an agent can make within a single `process_message` invocation to prevent runaway execution and costs. For example, allow a maximum of 3-5 tool calls before returning an error to the user.

Summary & Next Step

In this chapter, we significantly enhanced our ADK agent's capabilities by integrating external tools and enabling it to perform multi-step actions. We learned:

- How to define a standard Python function as an agent tool, complete with docstrings for LLM understanding.
- How to register this tool with the Google ADK (specifically, the `genai.GenerativeModel`).
- The critical role of parsing tool outputs and sending `glm.ToolOutput` objects with `glm.FunctionResponse` back to the model for continued reasoning.
- The importance of persistent state in maintaining context across complex, multi-turn workflows involving tools.
- Key production considerations for tool security, robust error handling, comprehensive observability, and scalable design.

Our agent can now intelligently decide when to use a tool to gather information, process that information, and provide a more informed response, making it far more capable than a purely conversational bot. This is a crucial step towards building truly autonomous and capable AI agents.

The next step is to prepare our agent for deployment to a production-like environment. This means containerizing our application using Docker, making it portable and scalable, ready for services like Google Cloud Run or Kubernetes.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Gemini API Overview - Google AI for Developers
- Function calling with Gemini models - Google AI for Developers
- google-generative-ai Python client library - GitHub
- Python in Google Cloud - Official Documentation
- Google Secret Manager - Official Documentation
- Google Cloud Firestore - Official Documentation
- Python `logging` module - Official Documentation

CHAPTER 06

Containerizing Your ADK Agent for Portability and Scalability

Packaging your AI agent into a portable, self-contained unit is a critical step towards production readiness. This chapter guides you through containerizing your Google ADK agent using Docker, transforming it from a local Python script into a deployable artifact.

By the end of this milestone, you will have a fully functional Docker image of your long-running ADK agent. This image encapsulates all its dependencies and configurations, ensuring it runs consistently across different environments, from your local machine to various cloud services. This consistency is vital for scaling, maintaining, and debugging your agent system effectively.

Project Overview for This Chapter

In previous chapters, we developed a long-running AI agent using Google's Agent Development Kit (ADK) and integrated it with an external state store to manage conversational context and agent state. While functional, our agent currently runs directly from a Python environment on a developer's machine. This approach introduces challenges when moving to production, such as dependency conflicts, environment inconsistencies, and complex deployment processes.

This chapter's objective is to address these challenges by containerizing the agent. We will create a Docker image that bundles our agent's code, its Python runtime, and all its dependencies into a single, isolated package. This package can then be reliably run on any system with Docker installed, paving the way for scalable cloud deployments.

Tech Stack Overview

For this containerization milestone, we leverage the following core technologies:

- **Python (3.12):** The primary language for our ADK agent.
- **Google ADK (via `google-generativeai`):** The framework and libraries for building the AI agent.
- **Docker (latest stable):** The containerization platform used to package our application.

- **Redis Client (`redis-py`):** Example client for interacting with our external state store (if Redis was chosen in previous chapters).
- `python-dotenv`: For local environment variable management during development.

Milestones for Containerization

To achieve a containerized ADK agent, we will proceed through the following steps:

1. **Refine Python Dependencies:** Ensure `requirements.txt` precisely lists all necessary Python packages.
2. **Author the Dockerfile:** Create a `Dockerfile` that specifies how to build our agent's image.
3. **Build the Docker Image:** Execute the Docker build command to create the executable image.
4. **Run and Verify Locally:** Launch a container from our image to confirm the agent operates as expected.

Architecture for Containerized Agents

Containerization fundamentally shifts how our application's runtime environment is managed. Instead of relying on the host system's Python installation and libraries, the container provides its own isolated environment.

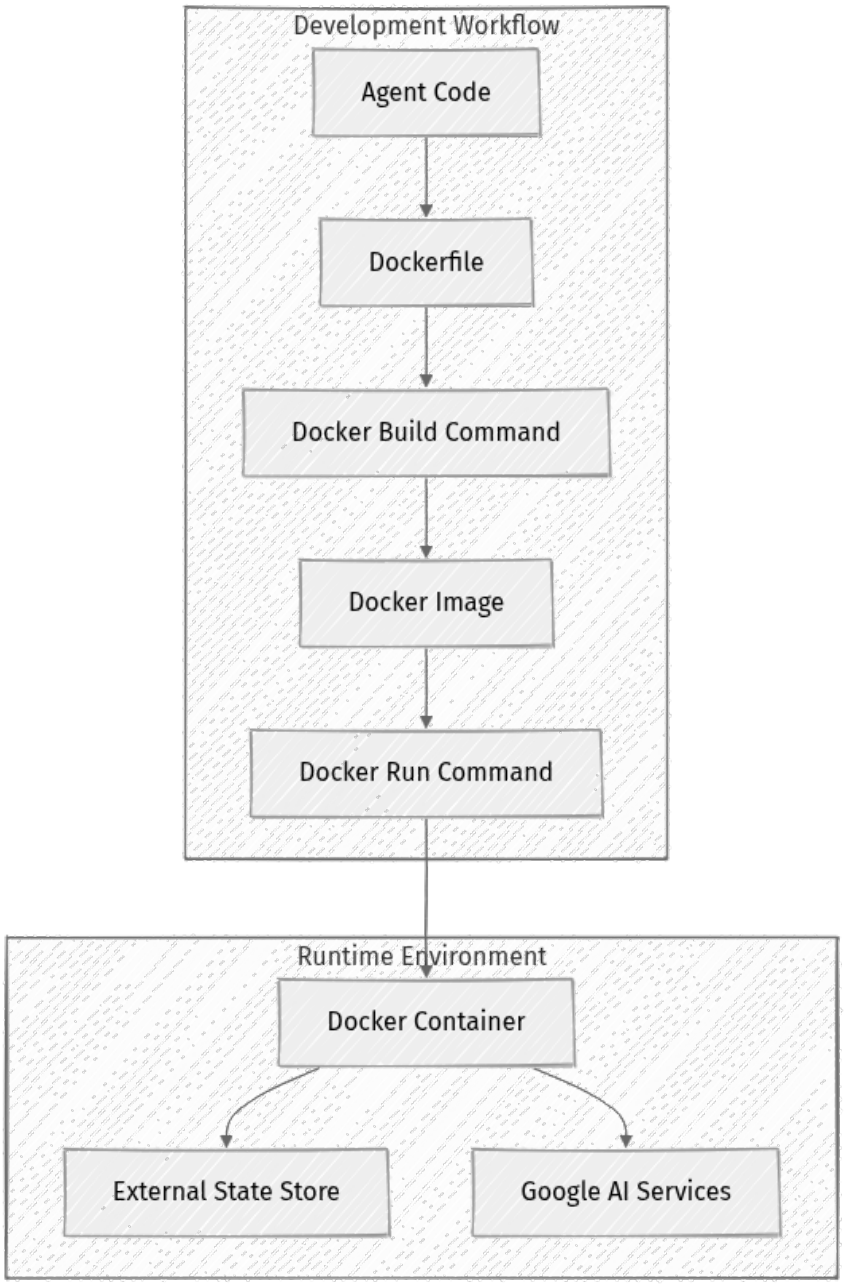
Core Containerization Concepts

- **Dockerfile:** This plain text file serves as a blueprint. It contains a sequence of instructions (commands) that Docker executes to build a new image. Think of it as a script for assembling your application's environment.
- **Image:** A Docker image is a lightweight, immutable, and executable package. It includes everything needed to run a piece of software: the code, a runtime (like Python), system tools, system libraries, and configuration. Images are read-only templates.
- **Container:** A container is a runnable instance of a Docker image. When you "run" an image, Docker creates a container, which is an isolated process on your host machine. Each container has its own filesystem, network stack, and process space, isolated from other containers and the host.

Agent Containerization Flow

Our ADK agent, once containerized, will run inside a Docker container. This container will still need to communicate with external services: our chosen state store (e.g., Redis, Firestore) for persistent context, and Google's AI services via the ADK. These external dependencies are not bundled inside the container but are accessed from it over the network.

The container itself won't expose an HTTP port in this initial setup, as our agent is designed to be a long-running background process. If a web interface or API were added later, a port would then be exposed.



- **Agent Code:** Your Python application, including ADK agent logic and state management.

- **Dockerfile:** Instructions defining the container image's contents and setup.
- **Docker Build Command:** The command executed to create the Docker image from the `Dockerfile`.
- **Docker Image:** The self-contained, executable package of your agent and its dependencies.
- **Docker Run Command:** The command to start a new container instance from the Docker image.
- **Docker Container:** The isolated, running instance of your ADK agent.
- **External State Store:** Your chosen database (e.g., Redis, Firestore) for persistent context and state.
- **Google AI Services:** Google's generative AI models (e.g., Gemini) accessed via the ADK.

📌 **Key Idea:** Containerization decouples your application from its underlying infrastructure. This means your agent runs the same way, regardless of whether it's on your laptop, a virtual machine, or a cloud service, eliminating environment-related issues.

Prerequisites

Before you start, ensure you have Docker installed on your development machine.

- **Docker Engine / Docker Desktop:**
  - **Version:** As of 2026-05-23, the latest stable release of Docker Desktop is recommended. You can download it from the official Docker website.
  - **Installation:** Follow the instructions for your operating system: https://docs.docker.com/get-docker/

Step-by-Step Implementation

We'll begin by verifying our Python dependencies, then create the `Dockerfile`, build the image, and finally run it to confirm functionality.

1. Update `requirements.txt`

Ensure your `requirements.txt` file, located at the project root (`./requirements.txt`), accurately lists all Python dependencies required for your agent. This is crucial because Docker will use this file to install packages inside the container.

For example, if you're using Redis for state management, your `requirements.txt` might look like this:

```
google-generativeai~=0.9.0
redis~=5.0.0
python-dotenv~=1.0.0
```

- **`google-generativeai~=0.9.0`**: This is the Python client library for interacting with Google's Gemini models, which is a core component of building agents with the Google Agent Development Kit (ADK). The ADK itself is more of a framework and set of patterns, often leveraging this library. (Version checked 2026-05-23: Version `0.9.0` is a recent stable release, but always verify the absolute latest on PyPI if `0.9.0` isn't the most recent when you build).
- **`redis~=5.0.0`**: If you are using Redis for state persistence, this is the official Python client library. (Version checked 2026-05-23: `5.0.0` is a recent stable release).
- **`python-dotenv~=1.0.0`**: This library helps load environment variables from a `.env` file, which is useful for local development but less common in production container deployments where environment variables are managed by the orchestrator.

2. Create the `Dockerfile`

Create a new file named `Dockerfile` (no file extension) in the root of your project directory. This file will contain the instructions for building your Docker image.

```
# Use an official Python runtime as a parent image.
# We choose a slim-bookworm image for smaller size and security, based on
Debian 12.
# Python 3.12 is the latest stable series as of 2026-05-23.
FROM python:3.12-slim-bookworm

# Set the working directory in the container.
# All subsequent commands will be executed relative to this directory.
WORKDIR /app

# Copy the requirements file into the container's working directory.
```

```
COPY requirements.txt .

# Install any needed packages specified in requirements.txt.
# --no-cache-dir reduces image size by not storing build cache.
# -r specifies the requirements file.
RUN pip install --no-cache-dir -r requirements.txt

# Copy the entire current directory (your project code) into the container.
# This assumes your agent logic is in files like agent.py, main.py,
state_manager.py, etc.,
# directly under the project root or in subdirectories that get copied.
COPY . .

# Define environment variables if needed.
# These can be overridden at runtime.
# ENV GOOGLE_APPLICATION_CREDENTIALS=/app/credentials.json # Example for GCS/
Firestore

# Command to run the application.
# This specifies the command that will be executed when the container starts.
# Replace 'main.py' with the entry point of your ADK agent application.
CMD ["python", "main.py"]
```

Let's break down each instruction:

- **`FROM python:3.12-slim-bookworm`**: This is the base image. We start with a lightweight Python 3.12 image based on Debian's "Bookworm" distribution. The `slim` tag means it contains only the bare minimum to run Python, which is crucial for smaller, more secure images. Using a specific version (like `3.12`) ensures reproducibility.
- **`WORKDIR /app`**: This instruction sets the working directory inside the container to `/app`. All subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` instructions will be executed relative to this directory.
- **`COPY requirements.txt .`**: This copies your `requirements.txt` file from your local machine (the build context) to the `/app` directory inside the container.
- **`RUN pip install --no-cache-dir -r requirements.txt`**: This command executes `pip install` inside the container to install all Python dependencies listed in `requirements.txt`. The `--no-cache-dir` flag is a best practice that prevents `pip` from storing downloaded packages in a cache, thereby reducing the final image size.
- **`COPY . .`**: This copies all your project files (including `agent.py`, `state_manager.py`, `main.py`, etc.) from your local directory into the `/app` directory in the container. This step is placed after dependency installation to leverage Docker's build cache. If only your code changes, Docker can reuse the cached layer for dependency installation, speeding up builds.

- `CMD ["python", "main.py"]`: This defines the default command that will be executed when a container starts from this image. It instructs Docker to run your agent's main entry point script using the Python interpreter. Ensure `main.py` is the actual entry point of your agent.

⚡ **Quick Note:** For larger projects, consider adding a `.dockerignore` file in your project root. This file works similarly to `.gitignore` and prevents unnecessary files (like `.git/`, `__pycache__/`, `venv/`, local `.env` files, or large data files) from being copied into your Docker image, which can significantly reduce image size and build times.

3. Build the Docker Image

With your `Dockerfile` ready, navigate to your project's root directory in your terminal. This is where your `Dockerfile` and `requirements.txt` should be located.

Run the Docker build command:

```
docker build -t adk-long-running-agent .
```

- `docker build`: The command to initiate a Docker image build.
- `-t adk-long-running-agent`: This "tags" the image with a human-readable name (`adk-long-running-agent`). This tag allows you to easily reference the image later when running containers or pushing to a registry. You can also specify a version, e.g., `adk-long-running-agent:v1.0`.
- `.`: This specifies the "build context." It tells Docker to look for the `Dockerfile` and any files referenced (like `requirements.txt` or your source code) in the current directory.

This build process might take a few minutes as Docker downloads the base image layers and installs all your Python dependencies. Upon successful completion, you will see a message indicating the image has been built and tagged.

You can verify the image exists by listing your local Docker images:

```
docker images
```

You should see `adk-long-running-agent` listed among them.

4. Run the Docker Container Locally

Now that you have a Docker image, you can run it as a container to test your agent in an isolated environment.

```
docker run -it --rm \
  -e REDIS_HOST=localhost \
  -e REDIS_PORT=6379 \
  -v /path/to/local/credentials.json:/app/credentials.json:ro \
  adk-long-running-agent
```

Let's dissect this command:

- `docker run`: The command to create and start a container from an image.
- `-it`: This combines two flags:
  - `-i` (interactive): Keeps `STDIN` open even if not attached.
  - `-t` (TTY): Allocates a pseudo-TTY, allowing you to see the agent's output directly in your terminal and interact with it if your agent has interactive input.
- `--rm`: This is a cleanup flag. It automatically removes the container and its filesystem when the container exits. This is highly useful for testing to prevent accumulating stopped containers.
- `-e REDIS_HOST=localhost`: Sets an environment variable named `REDIS_HOST` inside the container to `localhost`. Your agent's `state_manager.py` (or similar) should be configured to read this variable to connect to your Redis instance.
- `-e REDIS_PORT=6379`: Sets the `REDIS_PORT` environment variable.

- `-v /path/to/local/credentials.json:/app/credentials.json:ro`: This is a **volume mount**. It mounts your local Google service account credentials file (e.g., `service-account.json`) into the container at the specified path (`/app/credentials.json`).
    - `/path/to/local/credentials.json`: **Replace this with the actual, absolute path to your Google service account key file on your local machine.**
    - `/app/credentials.json`: This is the path inside the container where the file will be accessible. Your agent should be configured to look for credentials at this path, potentially via the `GOOGLE_APPLICATION_CREDENTIALS` environment variable if using `google-auth` directly, or implicitly if using client libraries that look for this file.
    - `:ro`: Makes the mounted volume read-only inside the container, enhancing security by preventing the container from modifying the host file.
- `adk-long-running-agent`: The name of the Docker image we built in the previous step.

🧠 **Important Security Note on Credentials:** Mounting credentials directly via `-v` is acceptable for local development and testing. However, for production deployments, this method is generally discouraged due to security risks. More secure methods include:

- **Google Secret Manager:** For Google Cloud deployments, store your credentials securely and access them programmatically.
- **Kubernetes Secrets:** If deploying to Kubernetes, use Kubernetes Secrets.
- **Workload Identity/Service Account Impersonation:** On managed cloud services like Google Cloud Run or GKE, configure the service to run with a specific Google Service Account, allowing it to authenticate to other Google Cloud services without needing to explicitly provide a key file. This is the most secure and recommended approach for Google Cloud.

Testing & Verification

Once you execute the `docker run` command, your agent should start within the container. Observe its output in your terminal to verify its operation.

1. **Agent Startup Confirmation:** Look for logging messages from your agent indicating successful initialization, connection to the external state store (e.g., Redis), and readiness to process tasks. Any immediate errors will usually be printed to `stderr` and appear in your terminal.
2. **State Persistence Test:**
    - If your agent has a basic interactive capability (e.g., a simple "hello" or "remember my name" function), try interacting with it.
    - Then, stop the container (Ctrl+C in the terminal where it's running).
    - Restart the container with the same `docker run` command.
    - Verify that the agent can retrieve previous context or state from the external store, confirming the persistence mechanism works across container restarts.
3. **External Connectivity:** Ensure the agent can successfully connect to your external Redis instance (or other state store) and Google's AI services. Connection errors will typically manifest as Python exceptions during startup or first use.
4. **Error Handling Test:** Intentionally introduce a misconfiguration (e.g., provide an incorrect `REDIS_HOST` value) and observe how the container logs the error. Does it exit gracefully or crash? This helps validate your agent's error handling.

If the container exits immediately with an error, or you suspect issues, check the container logs:

```
# First, find the container ID (if it exited, use -a)
docker ps -a

# Then, inspect its logs
docker logs <container_id_or_name>
```

The logs will often provide a Python traceback or error message that can help diagnose the problem.

Operationalizing Containers: Production Considerations

Containerization is a powerful step, but moving to production requires additional hardening and operational awareness.

Image Size and Security

- **Slim Base Images:** Always prioritize `slim` or `alpine` base images (like `python:3.12-slim-bookworm`). These images contain only essential components, significantly reducing the image size and the attack surface for potential vulnerabilities.
- **Non-Root User:** Running processes inside a container as the `root` user is a security risk. For enhanced security, create a dedicated non-root user in your `Dockerfile` and switch to it.

```
    # ... (after installing dependencies)
    RUN adduser --system --group appuser
    USER appuser
    # ...
```

```
This minimizes the impact if a process inside the container is compromised.
```

- **Multi-Stage Builds:** For more complex applications with build-time dependencies (e.g., compilers) not needed at runtime, use multi-stage builds. This allows you to discard build tools in the final image, drastically reducing its size. While less critical for simple Python apps, it's a valuable pattern.

Environment Variables and Configuration

- **Externalize Configuration:** Never hardcode sensitive information (API keys, database credentials, specific endpoint URLs) directly into your `Dockerfile` or application code. Use environment variables that are injected at runtime.

- **Secrets Management:** In production environments, sensitive data should be managed by a dedicated secrets manager.
    - **Google Secret Manager:** For Google Cloud, this service securely stores and manages sensitive configuration data. Your application can retrieve secrets programmatically.
    - **Kubernetes Secrets:** If deploying to Kubernetes, use its native Secrets management.
    - **Vault (HashiCorp):** A popular open-source solution for managing secrets across various platforms.

Resource Management and Reliability

- **Resource Limits:** When deploying containers to cloud platforms (like Google Cloud Run or GKE), define explicit CPU and memory limits. This prevents a misbehaving agent from consuming excessive resources, which can lead to higher costs or impact other services.
- **Health Checks:** Implement health checks (e.g., liveness and readiness probes in Kubernetes or similar concepts in Cloud Run). These mechanisms allow the orchestrator to detect if your agent is unhealthy (e.g., stuck in a loop, not responding) and automatically restart it, improving reliability.
- **Graceful Shutdown:** Design your agent to handle `SIGTERM` signals, allowing it to perform a graceful shutdown (e.g., save any in-progress state, close connections) when the container is stopped or restarted.

Logging and Observability

- **Standard Output:** Ensure your agent logs all relevant information to `stdout` (standard output) and `stderr` (standard error) within the container. Docker and cloud logging services (like Google Cloud Logging) are designed to automatically collect and centralize these streams, making them easily searchable and analyzable.
- **Structured Logging:** Adopt structured logging (e.g., JSON format) for your agent's logs. This makes logs much easier to parse, filter, and query in production environments, especially when dealing with large volumes of data. Libraries like `structlog` or Python's `logging` module with a JSON formatter can help.
- **Metrics:** Consider exposing metrics (e.g., agent processing time, number of messages processed, errors) using a library like Prometheus client. These metrics can be scraped and visualized in monitoring dashboards (e.g., Google Cloud Monitoring, Grafana).

Common Issues & Solutions

Even with careful planning, issues can arise during containerization. Here are some common problems and their debugging strategies.

1. **`ModuleNotFoundError` within the container:**
    - **Issue:** Your agent starts but fails because it cannot find a Python module, even though it works locally.
    - **Reason:** The module was either not listed in `requirements.txt`, or `requirements.txt` was not copied/installed correctly in the `Dockerfile`.
    - **Solution:**
        1. **Verify `requirements.txt`:** Double-check that all Python dependencies, including transitive ones, are explicitly listed in `requirements.txt`.
        2. **Inspect `Dockerfile`:** Ensure the `COPY requirements.txt .` and `RUN pip install --no-cache-dir -r requirements.txt` instructions are present and correctly ordered before `COPY . .`.
        3. **Rebuild Image:** Always rebuild the Docker image after modifying `requirements.txt` or `Dockerfile`.
        4. **Inspect Container:** You can shell into a running container to manually check: `docker run -it --entrypoint bash adk-long-running-agent` then `pip list` to see installed packages.

2. **Container exits immediately after starting:**
    - **Issue:** When you run `docker run`, the command completes almost instantly, and `docker ps -a` shows your container exited with a non-zero status code.
    - **Reason:** This usually indicates an error in your `CMD` instruction in the `Dockerfile` or an unhandled exception that occurs very early in your `main.py` script, causing it to crash immediately.
    - **Solution:**
        1. **Check Container Logs:** The most critical step: `docker logs <container_id>`. This will almost always reveal the Python traceback or error message that caused the exit.
        2. **Verify `CMD`:** Ensure your `CMD` instruction in the `Dockerfile` points to the correct entry point (`CMD ["python", "main.py"]`).
        3. **Local Test:** Run your `main.py` script locally directly from your terminal (`python main.py`) to ensure it can start without errors outside of Docker.
3. **Large Docker Image Size:**
    - **Issue:** Your Docker image is unexpectedly large (e.g., hundreds of MBs or even GBs), leading to slow downloads and increased storage costs.
    - **Reason:** Common culprits include using a full-fledged base image (e.g., `python:3.12` instead of `python:3.12-slim-bookworm`), not using `--no-cache-dir` with `pip install`, or copying unnecessary files into the image.
    - **Solution:**
        1. **Use Slim Base Images:** Always use `slim` or `alpine` variants for your base image (`FROM python:3.12-slim-bookworm`).
        2. **Optimize `pip install`:** Ensure `--no-cache-dir` is used with `pip install`.
        3. **Implement `.dockerignore`:** Create a `.dockerignore` file in your project root to exclude files and directories that are not needed in the image (e.g., `.git/`, `__pycache__/`, `venv/`, local `.env` files, documentation, large data files not used at runtime).
        4. **Multi-Stage Builds:** For more advanced scenarios, consider multi-stage builds to separate build-time dependencies from runtime dependencies.

Summary & Next Step

You've successfully containerized your long-running ADK agent! This is a significant milestone that brings us much closer to a production-ready system:

- Your agent is now packaged into a self-contained Docker image, making it highly portable and ensuring consistent execution across diverse environments.
- You understand the core components of Docker (Dockerfile, image, container) and how they apply to packaging your agent.
- You've gained practical experience in building and running your agent within a Docker container and learned how to troubleshoot common containerization issues.
- You've explored critical production considerations for containerized applications, including security, resource management, and observability.

This portable Docker image is now ready for deployment to a cloud environment. In the next chapter, we will take this image and deploy it to a scalable, managed service on Google Cloud, making your agent accessible and reliable in a production-like setting.

References

1. Docker Official Documentation: https://docs.docker.com/
2. Python Official Documentation: https://www.python.org/doc/
3. Google Generative AI Python SDK (Pypi): https://pypi.org/project/google-generativeai/
4. Google Cloud Secret Manager Documentation: https://cloud.google.com/secret-manager/docs
5. Docker Documentation: Best practices for writing Dockerfiles: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Robust Testing for Long-Running Agent Workflows

Building a reliable, long-running AI agent that can pause, resume, and maintain its conversational context across sessions is paramount for production systems. This chapter focuses on establishing a robust testing framework to ensure our Google ADK agent's state persistence and recovery mechanisms function flawlessly under various conditions.

By the end of this milestone, you will have implemented unit, integration, and end-to-end tests. These tests will validate the agent's ability to save and load its state, preserve conversation history, and correctly resume complex workflows after an interruption. This rigorous testing is crucial for delivering an AI agent that users can trust not to "forget" their interactions.

Project Overview

Our overarching project aims to develop a persistent AI agent using Google's Agent Development Kit (ADK). This agent is designed to handle multi-turn conversations and complex workflows, maintaining its state and context even if its execution is interrupted or paused. This persistence is achieved by decoupling the agent's in-memory state from its durable storage, leveraging a robust external data store like Google Cloud Firestore.

The core challenge for such an agent lies in ensuring that its state is consistently saved and accurately restored. Any loss of context could lead to frustrating user experiences, broken business processes, and a significant erosion of user trust. This chapter directly addresses that challenge by building a comprehensive test suite.

Tech Stack for Testing

To build our robust test suite, we will primarily utilize:

- **Python 3.12.x:** The core language for our agent and tests. (Latest stable version as of 2026-05-23).
- **pytest:** A widely adopted and powerful testing framework for Python. (Latest stable version recommended).

- **pytest-mock:** A `pytest` plugin that provides a convenient fixture for `unittest.mock` functionality, essential for isolating components. (Latest stable version recommended).
- **Google Cloud Firestore Client Library:** For interacting with our chosen state persistence layer. (Latest stable version recommended).
- **Google Cloud Firestore Emulator:** A local, in-memory version of Firestore, critical for fast, isolated, and deterministic integration tests without incurring cloud costs or affecting live data.

The specific version of Google ADK is not publicly available as of 2026-05-23, but we assume compatibility with standard Python 3.x environments. Our agent's interaction with ADK will be modeled for testing purposes, focusing on the persistence aspects.

Milestones for This Chapter

This chapter is structured to build our testing capabilities incrementally:

1. **Unit Test State Serialization:** Verify the core logic of converting agent state to a storable format (e.g., JSON) and back, ensuring data integrity.
2. **Integration Test Persistence Layer:** Validate that our `StateManager` correctly interacts with Google Cloud Firestore (using the emulator) to save and load agent states.
3. **End-to-End Test Pause/Resume:** Simulate a full agent workflow, including an interruption and subsequent resumption, to confirm the agent picks up the conversation correctly.

Architecture for Testing Persistent Agents

Testing a stateful, long-running AI agent requires a multi-layered approach to ensure reliability at every level. Our testing architecture is designed to provide confidence in data integrity and workflow continuity.

Testing Layers

- **Unit Tests:** Focus on isolated functions or methods (e.g., `serialize_state`). They should be fast and have no external dependencies, often using mocks for any collaborators.

- **Integration Tests:** Verify the interaction between two or more components (e.g., `FirestoreStateManager` and the Firestore database emulator). These tests confirm that components work together as expected.
- **End-to-End (E2E) Tests:** Simulate a user's full journey with the agent, including pauses and resumes, to validate the entire system's behavior. These are the most comprehensive but also the slowest.

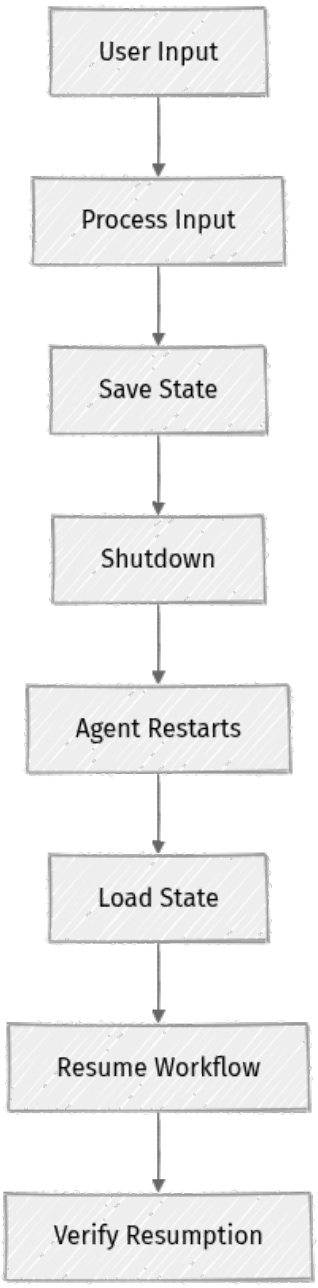
Test File Structure

We'll organize our tests in a dedicated `tests/` directory at the root of our project, mirroring our application's structure.

```
.
├── my_adk_agent/
│   ├── __init__.py
│   ├── agent.py               # Our simplified ADK agent model
│   └── persistence.py         # State serialization and Firestore interaction
├── tests/
│   ├── __init__.py
│   ├── test_persistence_units.py  # Unit tests for serialization
│   ├── test_persistence_integration.py # Integration tests for Firestore
│   └── test_e2e_resume.py         # E2E tests for pause/resume workflow
├── requirements.txt
└── main.py
```

E2E Pause/Resume Test Flow

The most critical test for a long-running agent is verifying its ability to pause and resume. This diagram illustrates the high-level flow of our end-to-end test.



Explanation: This flow ensures that after an interruption (simulated shutdown), a new agent instance can correctly load the previously saved state and continue the conversation from the exact point it left off.

Key Testing Principles

- **Isolation:** For unit tests, use mocking to replace external dependencies (like the state store or Google ADK's internal components) to focus purely on the logic under test.
- **Deterministic Scenarios:** Every test should be repeatable and produce the same results, regardless of when or where it's run. This is why local emulators are preferred over live cloud services for integration tests.

- **Clear Assertions:** Each test must make specific, unambiguous assertions about the expected state, output, or behavior. Vague tests lead to false confidence.
- **Realism for Integration/E2E:** While unit tests are mocked, integration tests should interact with a realistic (emulated) persistence layer, and E2E tests should simulate user interactions as closely as possible.

Step-by-Step Implementation

We'll use `pytest` for our tests. First, ensure the necessary libraries are installed.

```
pip install pytest pytest-mock google-cloud-firestore
```

Python version note: As of 2026-05-23, the latest stable Python 3.x series (e.g., Python 3.12.x) is recommended. The Google ADK version is not publicly available as of this date, so we assume compatibility with standard Python 3.x environments.

1. Unit Test: State Serialization/Deserialization

This test validates the `serialize_state` and `deserialize_state` functions, which are responsible for converting our agent's Python dictionary state into a JSON string and back. These functions are typically located in `my_adk_agent/persistence.py`.

Create `tests/test_persistence_units.py`:

```
# tests/test_persistence_units.py
import pytest
from my_adk_agent.persistence import serialize_state, deserialize_state

def test_state_serialization_deserialization():
    """
    Verifies that agent state can be serialized to JSON and deserialized back
    without loss of data or type changes.
    """
    initial_state = {
        "user_id": "test_user_123",
        "conversation_history": [
            {"role": "user", "content": "Hello"},
            {"role": "agent", "content": "Hi there!"}
        ],
        "current_workflow_step": "awaiting_confirmation",
        "context_data": {"order_id": "ORD-001", "item_count": 2},
        "is_active": True,
        "last_updated_timestamp": 1700000000.0, # Example timestamp
    }
```

```
    # 📌 Key Idea: Test round-trip fidelity.
    serialized_data = serialize_state(initial_state)
    assert isinstance(serialized_data, str), "Serialized data should be a
string."
    assert 'test_user_123' in serialized_data, "User ID should be present in
serialized data."
    assert 'awaiting_confirmation' in serialized_data, "Workflow step should
be present."

    deserialized_state = deserialize_state(serialized_data)

    # Assert that the deserialized state matches the original
    assert deserialized_state == initial_state,
"Deserialized state must match initial state."
    # Also check specific types for robustness, especially for numbers/
booleans
    assert isinstance(deserialized_state['is_active'], bool), "Boolean type
should be preserved."
    assert isinstance(deserialized_state['last_updated_timestamp'], float), "F
loat type should be preserved."
    assert deserialized_state['conversation_history'][0]['role'] == 'user', "N
ested list/dict data should be correct."

    print(f"\nInitial state: {initial_state}")
    print(f"Serialized data: {serialized_data}")
    print(f"Deserialized state: {deserialized_state}")

def test_empty_state_serialization():
    """Tests serialization/deserialization of an empty state dictionary."""
    empty_state = {}
    serialized = serialize_state(empty_state)
    deserialized = deserialize_state(serialized)
    assert deserialized == empty_state, "Empty state should serialize and
deserialize correctly."

def test_invalid_json_deserialization():
    """Tests deserialization with invalid JSON input to ensure error
handling."""
    # ⚠️ What can go wrong: Malformed JSON can crash your agent.
    with pytest.raises(ValueError, match="Invalid JSON format"):
        deserialize_state("this is not valid json")

    with pytest.raises(ValueError, match="Invalid JSON format"):
        deserialize_state("{'key': 'value'") # Malformed JSON with single
quotes
```

Explanation:

- **Round-trip Test:** The primary goal is to verify that `serialize_state` and `deserialize_state` are inverse operations, meaning `deserialize_state(serialize_state(data)) == data`.
- **Type Preservation:** JSON serialization can sometimes alter data types (e.g., numbers becoming strings). Explicit assertions ensure critical types like booleans and floats are maintained.

- **Error Handling:** Testing with invalid JSON confirms that our `deserialize_state` function gracefully handles bad input, raising a `ValueError` as expected, rather than crashing the application. This is a crucial production awareness point.

2. Integration Test: Firestore Persistence

This test interacts with a real (or emulated) Firestore instance to ensure our `save_agent_state` and `load_agent_state` functions work correctly with the actual database. Using a local Firestore emulator is vital for fast, reliable, and isolated integration tests.

First, set up a local Firestore emulator if you haven't already. [Google Cloud Firestore Emulator documentation]

Install the Google Cloud CLI (`gcloud`) and run:```bash gcloud components install cloud-firestore-emulator gcloud emulators firestore start --host-port=localhost:8080

```
Keep this running in a separate terminal.

Now, ensure your `my_adk_agent/persistence.py` has a `FirestoreStateManager`
class that uses `google.cloud.firestore.Client`. (This was implied from the
previous chapter's context).
```python
# my_adk_agent/persistence.py (Illustrative snippet, adapt to your
implementation)
import json
from google.cloud import firestore

class StateManager:
    """Abstract base class for state management."""
    def save_agent_state(self, agent_id: str, state: dict):
        raise NotImplementedError
    def load_agent_state(self, agent_id: str) -> dict | None:
        raise NotImplementedError

class FirestoreStateManager(StateManager):
    """Manages agent state persistence using Google Cloud Firestore."""
    def __init__(self, client: firestore.Client, collection_name: str =
"agent_states"):
        self.db = client
        self.collection_ref = self.db.collection(collection_name)

    def save_agent_state(self, agent_id: str, state: dict):
        """Saves the agent's state to Firestore."""
        doc_ref = self.collection_ref.document(agent_id)
        # Firestore automatically handles Python dicts, converting them to
JSON-like documents.
        # We assume `state` is already a dictionary of basic types or objects
Firestore can handle.
        doc_ref.set(state)
        # ⚡ Quick Note: For complex objects, you might need pre-serialization
here.
```
```

```
        # Our `serialize_state` is more for general JSON string output, not
direct Firestore dicts.
        # Firestore's client library handles basic Python types (dicts, lists,
int, str, bool, float) well.

    def load_agent_state(self, agent_id: str) -> dict | None:
        """Loads the agent's state from Firestore."""
        doc_ref = self.collection_ref.document(agent_id)
        doc = doc_ref.get()
        if doc.exists:
            return doc.to_dict()
        return None

def serialize_state(state: dict) -> str:
    """Serializes a dictionary state to a JSON string."""
    try:
        return json.dumps(state)
    except TypeError as e:
        raise ValueError(f"Failed to serialize state to JSON: {e}")

def deserialize_state(serialized_state: str) -> dict:
    """Deserializes a JSON string back to a dictionary state."""
    try:
        return json.loads(serialized_state)
    except json.JSONDecodeError as e:
        raise ValueError(f"Invalid JSON format for state: {e}")
```

Now, create `tests/test_persistence_integration.py`:

```
# tests/test_persistence_integration.py
import pytest
import os
from google.cloud import firestore
from my_adk_agent.persistence import FirestoreStateManager

# IMPORTANT: Configure Firestore emulator for tests
# This environment variable tells the client library to connect to the
emulator.
os.environ["FIRESTORE_EMULATOR_HOST"] = "localhost:8080"
os.environ["GCLOUD_PROJECT"] = "test-project-id" # Use a dummy project ID for
emulator

@pytest.fixture(scope="module")
def firestore_client_fixture():
    """Provides a Firestore client connected to the emulator for the test
module."""
    client = firestore.Client()
    yield client
    # Clean up any data created by tests in this module

# 🔥 Optimization / Pro tip: For more granular cleanup, delete per test or use
a unique collection name per test run.
    # For a simple module-scoped cleanup, deleting the collection is
sufficient.
    print("\nCleaning up Firestore emulator data...")
    docs = client.collection("agent_states").stream()
    for doc in docs:
        doc.reference.delete()
    print("Firestore emulator data cleaned.")
```

```
def test_save_and_load_agent_state_integration(firestore_client_fixture):
    """
    Tests saving and loading agent state to/from Firestore using the emulator.
    This validates the interaction between our StateManager and Firestore.
    """
    manager = FirestoreStateManager(firestore_client_fixture,
collection_name="agent_states")
    test_agent_id = "agent_test_id_001"
    initial_state = {
        "user_id": "test_user_001",
        "conversation_history": [{"role": "user", "content": "Hello
Firestore"}],
        "workflow_status": "pending_db_check",
        "nested_data": {"sub_key": 123, "list_items": ["a", "b"]}
    }

    # 1. Save state
    manager.save_agent_state(test_agent_id, initial_state)

    # 2. Load state
    loaded_state = manager.load_agent_state(test_agent_id)

    # 3. Assertions
    assert loaded_state is not None, "Loaded state should not be None."
    assert loaded_state == initial_state, "Loaded state must exactly match
initial state."
    assert loaded_state["user_id"] == "test_user_001"
    assert loaded_state["conversation_history"][0]["content"] == "Hello
Firestore"
    assert loaded_state["nested_data"]["sub_key"] == 123

    # Test updating state: crucial for long-running agents
    updated_state = initial_state.copy()
    updated_state["workflow_status"] = "completed_db_check"
    updated_state["new_field"] = "added_during_update"
    manager.save_agent_state(test_agent_id, updated_state)
    reloaded_updated_state = manager.load_agent_state(test_agent_id)

    assert reloaded_updated_state == updated_state, "Updated state must be
correctly saved and loaded."
    assert reloaded_updated_state["new_field"] == "added_during_update"

def test_load_non_existent_state(firestore_client_fixture):
    """
    Tests loading a state for an agent ID that does not exist.
    It should gracefully return None, indicating no state found.
    """
    manager = FirestoreStateManager(firestore_client_fixture,
collection_name="agent_states")
    non_existent_id = "non_existent_agent_id_123"
    loaded_state = manager.load_agent_state(non_existent_id)
    assert loaded_state is None, "Loading non-existent state should return
None."
```

Explanation:

- **Firestore Emulator:** The `os.environ` lines are critical. They tell the `google-cloud-firestore` client library to connect to the local emulator, ensuring tests are isolated and fast. `GCLOUD_PROJECT` is a dummy ID required by the client.
- `firestore_client_fixture`: This `pytest` fixture provides a configured `firestore.Client` instance. `scope="module"` means it runs once for all tests in this file. The `yield` statement ensures that cleanup (deleting test data from the emulator) happens after all tests in the module are complete, maintaining a clean slate for subsequent test runs.
- **Save and Load:** The `test_save_and_load_agent_state_integration` function saves a sample `initial_state` for a unique `agent_id`, then immediately loads it back. It asserts that the loaded state is identical to the original, confirming both saving and loading work. It also includes an update scenario, which is vital for long-running agents that continuously modify their state.
- **Edge Case: Non-existent State:** `test_load_non_existent_state` verifies that the system gracefully handles requests to load state for an `agent_id` that has no stored data, returning `None`.

3. End-to-End Test: Pause and Resume Workflow

This E2E test simulates an agent interaction, saves its state, then simulates a restart and verifies the agent can pick up exactly where it left off. This requires a simplified model of our ADK agent (`MyADKAgent`) that interacts with our `StateManager`.

Let's assume `my_adk_agent/agent.py` contains our main `MyADKAgent` class. This class will manage its internal state using our `StateManager`.

```
# my_adk_agent/agent.py (Illustrative snippet, adapt to your ADK agent
structure)
from .persistence import StateManager, FirestoreStateManager # Assuming
FirestoreStateManager is used
# from adk.agent import Agent as AdkAgent # Placeholder for ADK Agent class,
not directly used in this simplified model
# from adk.message import Message # Placeholder for ADK Message class

class MyADKAgent:
    """
    A simplified agent model for demonstrating state management and pause/
resume.
    In a real ADK setup, state management would integrate with ADK's lifecycle
hooks.
    This model abstracts the ADK specifics to focus on state persistence
testing.
```

```
    """
    def __init__(self, agent_id: str, state_manager: StateManager):
        self.agent_id = agent_id
        self.state_manager = state_manager
        # Load existing state or initialize new state
        self.state = self.state_manager.load_agent_state(self.agent_id) or sel
f._initial_state()
        print(f"Agent {self.agent_id} initialized. Current state:
{self.state.get('current_workflow_step')}")


    def _initial_state(self) -> dict:
        """Returns the default initial state for a new agent session."""
        return {
            "conversation_history": [],
            "current_workflow_step": "start",
            "context_data": {}
        }

    def _save_state(self):
        """Saves the current internal state of the agent using the state
manager."""
        self.state_manager.save_agent_state(self.agent_id, self.state)

    def handle_message(self, user_message: str) -> str:
        """
        Simulates agent processing of a user message and updates its internal
state.
        Saves state after each interaction.
        """
        self.state["conversation_history"].append({"role": "user", "content":
user_message})

        response = ""
        current_step = self.state.get("current_workflow_step")

        # 🧠 Important: This simplified logic demonstrates state transitions.
        # A real ADK agent would use LLM calls, tool execution, etc.
        if current_step == "start":
            response = "Welcome! What is your name?"
            self.state["current_workflow_step"] = "awaiting_name"
        elif current_step == "awaiting_name":
            self.state["context_data"]["user_name"] = user_message
            response = f"Nice to meet you, {user_message}. What task can I
help you with?"
            self.state["current_workflow_step"] = "awaiting_task"
        elif current_step == "awaiting_task":
            response = f"Understood. I will help with '{user_message}'. Is
that correct? (yes/no)"
            self.state["context_data"]["proposed_task"] = user_message
            self.state["current_workflow_step"] = "confirm_task"
        elif current_step == "confirm_task":
            if user_message.lower() == "yes":
                response = "Great! Starting your task now."
                self.state["current_workflow_step"] = "task_in_progress"
            else:
                response = "Okay, what task should I help with instead?"
                self.state["current_workflow_step"] = "awaiting_task"
        elif current_step == "task_in_progress":
            response = "I'm currently working on your task. What's next?"
        else:
            response = "I'm not sure how to proceed. Let's restart. What is
```

```
your name?"
            self.state["current_workflow_step"] = "awaiting_name"

        self.state["conversation_history"].append({"role": "agent",
"content": response})
        self._save_state() # Save state after every interaction to ensure
persistence
        return response

    def get_current_workflow_step(self) -> str:
        """Returns the current step in the agent's workflow."""
        return self.state.get("current_workflow_step", "unknown")

    def get_context_data(self) -> dict:
        """Returns the current context data maintained by the agent."""
        return self.state.get("context_data", {})
```

Now, create `tests/test_e2e_resume.py`:

```
# tests/test_e2e_resume.py
import pytest
import os
from google.cloud import firestore
from my_adk_agent.persistence import FirestoreStateManager
from my_adk_agent.agent import MyADKAgent

# Ensure Firestore emulator is configured for tests
os.environ["FIRESTORE_EMULATOR_HOST"] = "localhost:8080"
os.environ["GCLOUD_PROJECT"] = "test-project-id"

@pytest.fixture(scope="module")
def firestore_client_e2e_fixture():
    """Provides a Firestore client connected to the emulator for E2E tests."""
    client = firestore.Client()
    yield client
    # Clean up specific collection used by E2E tests
    print("\nCleaning up E2E Firestore emulator data...")
    docs = client.collection("e2e_agent_states").stream()
    for doc in docs:
        doc.reference.delete()
    print("E2E Firestore emulator data cleaned.")

def test_agent_pause_resume_workflow(firestore_client_e2e_fixture):
    """
    Tests an end-to-end scenario where an agent workflow is paused and then
resumed
    from the correct state and context. This simulates an agent being
restarted.
    """
    test_agent_id = "e2e_agent_id_001"
    state_manager = FirestoreStateManager(firestore_client_e2e_fixture, collec
tion_name="e2e_agent_states")

    # --- Part 1: Initial interaction and state saving (Simulate Agent A) ---
    print("\n--- Agent A: Initial conversation ---")
    agent_a = MyADKAgent(test_agent_id, state_manager)
    # Agent A starts, loads state (will be initial state as it's the first
run)
```

```
    # First interaction
    response1 = agent_a.handle_message("My name is Alice")
    print(f"Agent A Response 1: {response1}")
    assert "Nice to meet you, Alice" in response1, "Agent A should greet
Alice."
    assert agent_a.get_current_workflow_step() == "awaiting_task", "Agent A
should move to awaiting_task step."
    assert agent_a.get_context_data().get("user_name") == "Alice", "Agent A
should store user name."

    # Second interaction
    response2 = agent_a.handle_message("I need help with booking a flight")
    print(f"Agent A Response 2: {response2}")
    assert "booking a flight" in response2, "Agent A should confirm task."
    assert agent_a.get_current_workflow_step() == "confirm_task", "Agent A
should move to confirm_task step."
    assert agent_a.get_context_data().get("proposed_task") == "booking a
flight", "Agent A should store proposed task."

    # At this point, agent_a's state is saved to Firestore by `handle_message`
    print(f"Agent A current state: {agent_a.get_current_workflow_step()},
context: {agent_a.get_context_data()}")

    # --- Part 2: Simulate agent shutdown and restart (Simulate Agent B) ---
    print("\n--- Agent B: Resuming conversation ---")
    # Simulate a new instance of the agent, loading state from persistence
    agent_b = MyADKAgent(test_agent_id, state_manager)
    # Agent B initializes, and crucially, loads the state saved by Agent A

    # Verify agent_b has correctly loaded the state from agent_a
    assert agent_b.get_current_workflow_step() == "confirm_task", "Agent B
should resume at 'confirm_task' step."
    assert agent_b.get_context_data().get("user_name") == "Alice", "Agent B
should have Alice's name in context."
    assert agent_b.get_context_data().get("proposed_task") == "booking a
flight", "Agent B should have proposed task in context."
    print(f"Agent B loaded workflow step:
{agent_b.get_current_workflow_step()}")
    print(f"Agent B loaded context data: {agent_b.get_context_data()}")

    # Continue the conversation from where Agent A left off
    response3 = agent_b.handle_message("yes")
    print(f"Agent B Response 3: {response3}")
    assert "Starting your task now" in response3, "Agent B should proceed to
task start confirmation."
    assert agent_b.get_current_workflow_step() == "task_in_progress",
"Agent B should move to 'task_in_progress' step."
    assert agent_b.get_context_data().get("proposed_task") == "booking a
flight", "Agent B should retain proposed task after confirmation."

    print("\nE2E Pause/Resume Test Passed!")
```

Explanation:

- **Simplified Agent Model:** `MyADKAgent` is a simplified Python class that models the core state-dependent behavior of our ADK agent. It's not a full ADK agent, but it demonstrates how an agent would interact with our `StateManager` to save and load its state. This abstraction allows us to test the persistence logic without needing to mock complex ADK internals.
- **Two Agent Instances:** The test creates `agent_a` to simulate the initial conversation. After `agent_a` completes a few steps and saves its state, `agent_b` is instantiated with the same `agent_id`. This simulates a new process or a restarted agent picking up the conversation.
- **State Verification:** Before `agent_b` continues, crucial assertions are made to ensure it has loaded the correct `current_workflow_step` and `context_data` from `agent_a`'s last saved state. This is the heart of the pause/resume functionality.
- **Resumption:** `agent_b` then sends the next logical message ("yes"), and we assert that the conversation proceeds as expected, demonstrating successful resumption from a previously saved state.

Testing & Verification

To run all your tests, ensure the Firestore emulator is running in a separate terminal. Then, navigate to your project's root directory in the terminal and execute:

```
pytest
```

Expected Output:

You should see output similar to this, indicating all tests passed:

```
============================= test session starts
==============================
platform linux -- Python 3.12.x, pytest-X.X.X, pluggy-X.X.X
rootdir: /path/to/your/project
plugins: mock-X.X.X
collected 6 items

tests/test_persistence_units.py::test_state_serialization_deserialization
PASSED [ 16%]
tests/test_persistence_units.py::test_empty_state_serialization PASSED [ 33%]
tests/test_persistence_units.py::test_invalid_json_deserialization PASSED
[ 50%]
tests/
test_persistence_integration.py::test_save_and_load_agent_state_integration
```

```
PASSED [ 66%]
tests/test_persistence_integration.py::test_load_non_existent_state PASSED
[ 83%]
tests/test_e2e_resume.py::test_agent_pause_resume_workflow PASSED [100%]

============================== 6 passed in X.XXs
==============================
```

If any test fails, `pytest` will provide detailed traceback information, pointing you to the exact line where an assertion failed, helping you quickly identify the root cause.

Quick Debugging Checks

- **Firestore Emulator Status:** Is the Firestore emulator running on `localhost:8080`? If not, `pytest` will report connection errors for integration and E2E tests.
- **Environment Variables:** Verify that `FIRESTORE_EMULATOR_HOST` and `GCLOUD_PROJECT` are correctly set in your test files.
- **Serialization Logic:** If `test_persistence_units.py` fails, carefully inspect `my_adk_agent/persistence.py` for issues in `serialize_state` or `deserialize_state`, especially concerning complex data types or custom objects that might not convert cleanly to/from JSON.
- **State Mismatch in E2E:** For integration and E2E tests, add `print()` statements to display the `initial_state` and `loaded_state` (or `agent_a.state` and `agent_b.state`) at critical points. This allows for direct visual comparison to pinpoint where data loss or corruption occurs.

Production Considerations

Robust testing is a cornerstone of production readiness, especially for stateful AI agents.

- **CI/CD Integration:** Integrate these tests into your Continuous Integration/Continuous Deployment (CI/CD) pipeline. Every code change should trigger the full test suite, including emulator-based integration tests, to catch regressions early before deployment. This ensures that new features or bug fixes don't inadvertently break state persistence.

- **Performance Testing:** For high-throughput agents handling thousands of concurrent users, the performance of your state persistence layer is critical. Conduct performance tests on `save_agent_state` and `load_agent_state` under simulated load. Even a few milliseconds of delay per call can add up to significant latency for users or bottlenecks for the system.
- **Test Data Management:** For complex E2E tests, consider using factory libraries (e.g., `Faker`) to generate realistic, yet randomized, test data. Ensure your test data is always cleaned up between test runs to maintain test isolation and prevent test pollution.
- **Reliability vs. Speed:** Unit tests are fast, providing quick feedback. Integration and E2E tests, involving external dependencies (even emulated ones), will inherently be slower. Balance the depth of testing with the need for quick feedback in your CI pipeline. You might run a subset of fast tests on every commit and the full, slower suite less frequently (e.g., nightly builds or before major deployments).

Common Issues & Solutions

1. Test Flakiness Due to External Dependencies

Issue: Integration or E2E tests intermittently fail without clear cause, especially when interacting with real external services (e.g., a live database, a cloud API). This can be due to network latency, transient service issues, or rate limits.

Solution:

- **Why it happens:** Live external services introduce non-determinism. Network conditions vary, and cloud services can have temporary hiccups.
- **Use Emulators:** As demonstrated, prioritize local emulators (Firestore, Pub/Sub, etc.) for integration tests. They provide a consistent, fast, and isolated environment, eliminating external variability.
- **Mock External APIs:** For services without emulators, use `unittest.mock` or `pytest-mock` to replace actual API calls with controlled, predictable responses.
- **Retries (Limited Use):** In very specific E2E scenarios hitting staging environments (not local tests), implementing simple retry logic with exponential backoff for network-related failures might be necessary. Avoid this for local development tests.

2. Incomplete State Capture in Tests

Issue: An E2E test passes, but later in production, the agent loses some context or data upon resume. This often means the test's `initial_state` was too simplistic or didn't cover all aspects of the agent's real, evolving state.

Solution:

- **Why it happens:** As agents evolve, new fields or complex data structures are added to their internal state, but the persistence logic or test cases aren't updated to reflect this.
- **Comprehensive State Models:** Ensure your test `initial_state` objects are as close as possible to the full complexity of your agent's actual state, including nested dictionaries, lists, and various data types.
- **Deep Assertions:** Beyond `assert loaded_state == initial_state`, add specific assertions for critical fields, especially those that are deeply nested or have specific type requirements. This forces a more thorough check.
- **Review Persistence Logic:** Regularly review your agent's `_save_state` and `load_agent_state` methods to ensure all relevant data is being persisted and restored. Consider an explicit schema or Pydantic models for your agent state to prevent accidental omissions.

3. Difficulty Testing Asynchronous Agent Actions

Issue: If your ADK agent performs asynchronous operations (e.g., calling external APIs, waiting for user input, processing long-running tasks), it can be challenging to test the state at specific intermediate points during these operations.

Solution:

- **Why it happens:** Asynchronous operations execute concurrently, making it hard to predict the exact state at a given moment without explicit synchronization.
- **Mock Asynchronous Calls:** Use `unittest.mock.AsyncMock` (Python 3.8+) or `pytest-asyncio` to mock asynchronous functions. This allows you to control their return values and side effects, making the asynchronous behavior deterministic for tests.
- **Introduce Checkpoints:** Design your agent with explicit "save points" or state transitions that occur after asynchronous operations complete. Your tests can then assert the state at these known, stable checkpoints.

- **Event-Driven Testing:** For truly complex asynchronous flows, consider an event-driven testing approach where tests assert that specific events are emitted, rather than trying to inspect internal state at every microsecond.

Summary & Next Step

In this chapter, we've established a critical foundation for ensuring the reliability of our long-running ADK agent: a robust testing suite. We successfully implemented:

- **Unit tests** for state serialization and deserialization, verifying the integrity of data conversion.
- **Integration tests** for our Firestore persistence layer, confirming seamless interaction with the database emulator for saving and loading agent states.
- **End-to-end tests** that simulate the entire pause-and-resume workflow, providing high confidence that the agent can pick up a conversation exactly where it left off, even after a restart.

This comprehensive testing approach provides confidence that our agent's ability to maintain context and state across sessions is solid and production-ready. The agent is now not only functional but also verifiable for its core persistence features, which is essential for user trust and system stability.

Our agent is now capable of persisting its state and has a strong testing framework in place. The next crucial step is to prepare it for actual deployment. In the next chapter, we will focus on **Containerization with Docker** to package our agent for scalable and portable deployment to Google Cloud.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- pytest Official Documentation
- Google Cloud Firestore Emulator Documentation
- unittest.mock — mock object library
- Python 3.12.x Documentation
- Google Cloud Client Library for Python (Firestore)

CHAPTER 08

Deploying and Monitoring Your Production ADK Agent on Google Cloud

This chapter marks a critical transition: moving your sophisticated, context-aware ADK agent from a local development environment to a production-grade cloud platform. We'll focus on deploying the containerized agent built in the previous chapter to Google Cloud Run, a fully managed serverless platform. Beyond deployment, we'll establish essential operational capabilities, including secure secret management, robust logging, and foundational monitoring.

By the end of this chapter, you will have a live, accessible ADK agent running on Google Cloud, capable of persisting its state and conversational context, ready to serve users reliably. This milestone is about making your agent resilient, scalable, and observable in a real-world environment.

Project Overview: From Prototype to Production

In previous chapters, you designed and built an ADK agent capable of maintaining long-term conversational context by integrating with Google Cloud Firestore. You also containerized this agent using Docker. This chapter's objective is to take that robust, containerized agent and deploy it to a production-ready environment on Google Cloud. We'll prioritize automation, security, and observability to ensure the agent is not just functional but also maintainable and reliable in the wild.

Tech Stack for Production Deployment

To achieve a production-grade deployment, we'll leverage several key Google Cloud services and Python libraries:

- **Python (3.12):** The core language for our agent and FastAPI application.
- **Google ADK (version unknown, checked 2026-05-23):** The Agent Development Kit for building the core agent logic.
- **FastAPI (0.111.0):** A modern, fast (high-performance) web framework for Python, used to expose our agent via an HTTP API.
- **Uvicorn (0.30.1):** An ASGI server that runs the FastAPI application.

- **Docker:** For containerizing our agent, providing a portable and consistent deployment unit.
- **Google Cloud Run:** A fully managed serverless platform for deploying containerized applications. It handles infrastructure, scaling, and provides a public endpoint.
- **Google Cloud Firestore:** Our chosen NoSQL database for persisting agent state and conversational context across sessions.
- **Google Secret Manager:** A service for securely storing and managing sensitive configuration data (e.g., API keys).
- **Google Cloud Logging:** Centralized logging for all application and infrastructure logs, automatically integrated with Cloud Run.
- **Google Cloud IAM (Identity and Access Management):** For managing permissions and ensuring the principle of least privilege for our deployed services.
- **Google Artifact Registry:** A universal package manager for storing Docker images and other build artifacts.

Build Plan: Deploying Your Agent to the Cloud

Our deployment process is broken down into a series of logical steps, ensuring each component is correctly configured and secured.

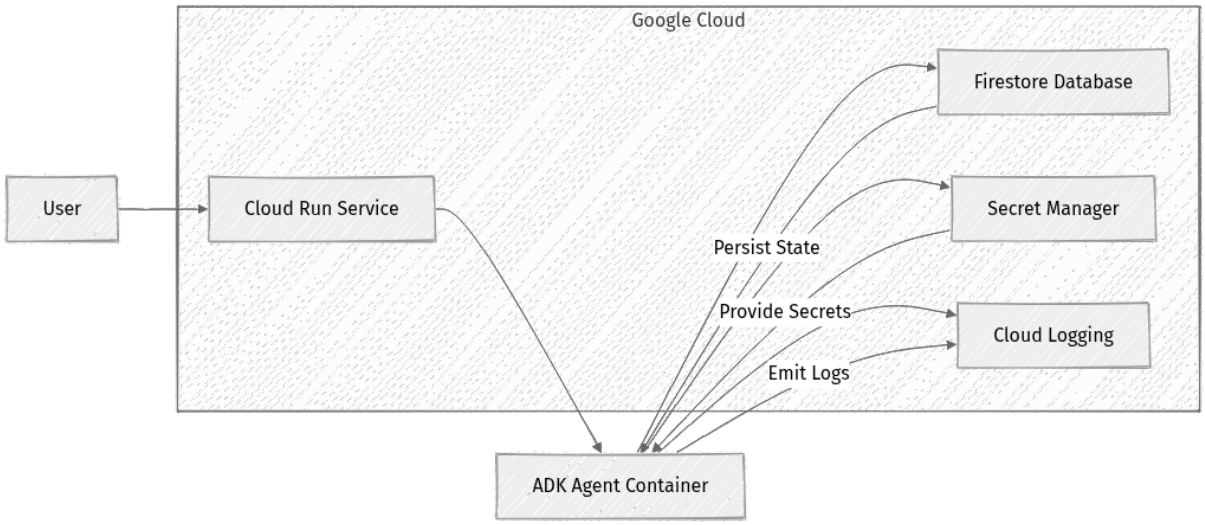
1. **Enable Google Cloud APIs:** Activate necessary services in your project.
2. **Create a Dedicated Service Account:** Establish a runtime identity for your agent with precise permissions.
3. **Refine Dockerfile for Cloud Run:** Optimize the container image for a serverless environment.
4. **Build and Push Docker Image:** Store your container in Google Artifact Registry.
5. **Secure Sensitive Configuration:** Manage any external API keys or secrets using Secret Manager.
6. **Deploy to Cloud Run:** Launch your containerized agent as a managed service.
7. **Verify and Monitor:** Test the deployed agent and inspect its logs and metrics.

Architecture: Production Deployment Strategy

Deploying an AI agent to production requires careful consideration of scalability, security, and observability. Google Cloud Run offers an excellent balance of ease of deployment, auto-scaling, and integration with other Google Cloud services, making it an ideal choice for our ADK agent.

Our deployment strategy centers on Cloud Run hosting the ADK agent. The agent will interact with Firestore for state persistence and potentially other external tools. Critical configurations like API keys will be managed securely using Google Secret Manager. All agent activity, including requests, responses, and internal processing, will be automatically captured by Google Cloud Logging.

Here's a high-level view of the architecture:



Why Cloud Run?

- **Serverless:** No infrastructure to provision or manage. You only pay for the compute resources consumed.
- **Auto-scaling:** Scales automatically from zero instances to handle peak loads, then scales back down, optimizing cost efficiency.
- **Container-based:** Deploy any application packaged as a Docker container, making it highly portable and environment-agnostic.
- **Integrated:** Seamlessly connects with other Google Cloud services like Firestore, Secret Manager, and Cloud Logging out of the box.

Step-by-Step Implementation

Before we begin, ensure you have the Google Cloud SDK installed and authenticated. The `gcloud` commands below assume you've already logged in and set your project.

```
gcloud auth login
gcloud config set project YOUR_GOOGLE_CLOUD_PROJECT_ID
```

1. Enable Required Google Cloud APIs

First, we need to ensure the necessary APIs are enabled in your Google Cloud project for Cloud Run, Artifact Registry, Secret Manager, and Logging.

```
gcloud services enable run.googleapis.com \
    artifactregistry.googleapis.com \
    secretmanager.googleapis.com \
    cloudbuild.googleapis.com \
    logging.googleapis.com
```

2. Create a Dedicated Service Account

It's best practice to use a dedicated Google Cloud Service Account for your Cloud Run service with the principle of least privilege. This service account will be the runtime identity of your agent, not for deployment itself.

File: `08_deployment_setup.sh`

Create this script in your project root and make it executable.

```
#!/bin/bash

# Get the current project ID
PROJECT_ID=$(gcloud config get-value project)
SERVICE_ACCOUNT_NAME="adk-agent-runner"
SERVICE_ACCOUNT_EMAIL="${SERVICE_ACCOUNT_NAME}@${PROJECT_ID}.iam.gserviceaccou
nt.com"

echo "Creating service account: ${SERVICE_ACCOUNT_EMAIL}"
gcloud iam service-accounts create "${SERVICE_ACCOUNT_NAME}" \
    --display-name "ADK Agent Cloud Run Service Account"

echo "Granting permissions to service account for runtime operations..."

# Grant Firestore Data Editor role for state persistence
# Firestore uses datastore roles for data access. This role allows read,
write, and delete.
gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
    --member "serviceAccount:${SERVICE_ACCOUNT_EMAIL}" \
    --role "roles/datastore.user"

# Grant Secret Manager Secret Accessor role to retrieve secrets
```

```
# This role allows the service account to access the secret payload.
gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
    --member "serviceAccount:${SERVICE_ACCOUNT_EMAIL}" \
    --role "roles/secretmanager.secretAccessor"

# Grant Cloud Logging Writer role
# While Cloud Run automatically captures stdout/stderr, this role allows
explicit logging if needed.
gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
    --member "serviceAccount:${SERVICE_ACCOUNT_EMAIL}" \
    --role "roles/logging.logWriter"

echo "Service account setup complete."
echo "Service Account Email: ${SERVICE_ACCOUNT_EMAIL}"
```

Run this script from your terminal:

```
bash 08_deployment_setup.sh
```

📌 **Key Idea:** The `adk-agent-runner` service account is strictly for the runtime identity of your Cloud Run service. It needs permissions to interact with other Google Cloud services (like Firestore, Secret Manager, Logging), but not to deploy or manage the Cloud Run service itself.

3. Update Dockerfile for Production Readiness

Ensure your `Dockerfile` (adapted from Chapter 7) is optimized for production. This includes using a multi-stage build to minimize image size and setting up the entrypoint for Uvicorn.

File: `requirements.txt`

Verify or create this file in your project root with the following dependencies. These versions were checked as stable on 2026-05-23.

```
adk
google-cloud-firestore
fastapi==0.111.0
uvicorn==0.30.1
pydantic==2.7.1
```

File: `Dockerfile`

Update your `Dockerfile` to use a multi-stage build.

```
# Stage 1: Build dependencies
FROM python:3.12-slim-bookworm as builder

WORKDIR /app
```

```
# Install ADK, FastAPI, Uvicorn, and other dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Final image for runtime
FROM python:3.12-slim-bookworm

WORKDIR /app

# Copy only necessary installed packages from the builder stage
COPY --from=builder /usr/local/lib/python3.12/site-packages /usr/local/lib/
python3.12/site-packages
# Copy your application code
COPY . .

# Environment variables for production best practices
ENV PYTHONUNBUFFERED=1
ENV ADK_ENV=production

# Cloud Run injects a PORT environment variable. We expose 8080 as a common
default.
EXPOSE 8080

# Command to run the agent using Uvicorn, listening on all interfaces
(0.0.0.0)
# and the port specified by Cloud Run (which defaults to 8080 if not set).
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Explanation:

- **`FROM python:3.12-slim-bookworm as builder`**: Starts the first stage with a slim Python image. `3.12-slim-bookworm` is a lightweight base image for Python 3.12.
- **`RUN pip install --no-cache-dir -r requirements.txt`**: Installs Python dependencies. `--no-cache-dir` prevents pip from storing cache, further reducing image size.
- **`FROM python:3.12-slim-bookworm`**: Starts the second, final stage with another clean, slim Python image.
- **`COPY --from=builder ...`**: This is the core of the multi-stage build. It copies only the installed Python packages from the `builder` stage, avoiding unnecessary build tools or temporary files.
- **`COPY . .`**: Copies your application code into the `/app` directory.
- **`ENV PYTHONUNBUFFERED=1`**: Ensures Python output is sent directly to `stdout`/`stderr`, which Cloud Logging can capture immediately without buffering delays.
- **`EXPOSE 8080`**: Declares that the container listens on port 8080. Cloud Run will use its `PORT` environment variable, which often defaults to 8080, to route traffic.

- `CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8080"]`: This is the entrypoint. It starts the Uvicorn ASGI server, which hosts your FastAPI application (where `app` is the FastAPI instance in `main.py`). It binds to `0.0.0.0` to listen on all available network interfaces.

File: `main.py`

Your `main.py` needs to expose an HTTP endpoint for Cloud Run to interact with, typically `/agent`. This endpoint will receive user messages and session IDs, then route them to your ADK agent's `handle_message` method, leveraging your persistent Firestore state manager.

```
# main.py
import os
import datetime
import logging # Use standard logging for better production practices
from fastapi import FastAPI, Request, HTTPException
from pydantic import BaseModel
from adk.agent import Agent
from adk.state_manager import InMemoryStateManager # ADK's internal, short-
term state manager
from adk.message import Message
from adk.state_manager.firestore import FirestoreStateManager # Our long-term
persistent state manager

# Configure basic logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__name__)

app = FastAPI()

# Pydantic model for incoming requests to ensure data validation
class AgentRequest(BaseModel):
    message: str
    session_id: str = "default_session"

# Initialize Firestore State Manager for long-term persistence
# This is *our* external state store, designed for cross-session context.
# It uses the GOOGLE_CLOUD_PROJECT environment variable, which Cloud Run will
provide.
PROJECT_ID = os.getenv("GOOGLE_CLOUD_PROJECT", os.getenv("GCP_PROJECT"))
if not PROJECT_ID:
    logger.error("GOOGLE_CLOUD_PROJECT environment variable not set. Cannot
initialize FirestoreStateManager.")
    raise ValueError("GOOGLE_CLOUD_PROJECT environment variable not set.")

persistent_state_manager = FirestoreStateManager(project_id=PROJECT_ID, collec
tion_name="adk_agent_sessions")
logger.info(f"FirestoreStateManager initialized for project: {PROJECT_ID},
collection: adk_agent_sessions")

# Define our custom agent that leverages the persistent state manager
class PersistentADKAgent(Agent):
    def __init__(self, name: str, persistent_state_manager: FirestoreStateMana
ger):
```

```
        # ADK's internal state_manager is typically for short-term
conversational context
        # within a single interaction or a very short-lived session.
        # For our long-running, pause-resume agent, we manage external
persistence separately.
        super().__init__(name, state_manager=InMemoryStateManager())
        self.persistent_state_manager = persistent_state_manager
        logger.info(f"PersistentADKAgent '{name}' initialized.")

    async def handle_message(self, message: Message, session_id: str = "defaul
t_session"):
        logger.info(f"Agent received message for session {session_id}: {messag
e.content}")

        # Load persistent context for this session from Firestore
        # This is where we implement the "pause/resume" and long-term memory.
        try:
            stored_context = await self.persistent_state_manager.load_state(se
ssion_id)
            if stored_context:
                logger.info(f"Loaded persistent context for {session_id}.")
                last_message_content = stored_context.get('last_message', 'non
e')
                last_timestamp = stored_context.get('timestamp', 'unknown')
                response_content = (
                    f"You said: '{message.content}'. "
                    f"Previously, on {last_timestamp}, you said: '{last_messag
e_content}'."
                )
            else:
                response_content = f"You said: '{message.content}'. This is a
new session."
                logger.info(f"No persistent context found for {session_id}.
Starting new session.")

            # Simulate agent processing and response
            # In a real agent, ADK's internal state_manager would be used here
            # for tool calls, intermediate thoughts, etc., within this single
interaction.
            response = Message(content=response_content, sender="agent")

            # Save updated persistent context to Firestore for the next
interaction
            await self.persistent_state_manager.save_state(
                session_id,
                {"last_message": message.content, "timestamp": datetime.dateti
me.utcnow().isoformat()}
            )
            logger.info(f"Saved updated persistent context for {session_id}.")
            return response
        except Exception as e:
            logger.error(f"Error in handle_message for session {session_id}:
{e}", exc_info=True)
            # Re-raise to be caught by the FastAPI exception handler
            raise

# Instantiate our agent globally
adk_agent = PersistentADKAgent(name="MyPersistentADKAgent", persistent_state_m
anager=persistent_state_manager)

@app.post('/agent')
async def agent_endpoint(request_data: AgentRequest):
```

```
    """
    Handles incoming messages for the ADK agent.
    Receives a message and session_id, processes it with the agent,
    and returns the agent's response.
    """
    user_message_content = request_data.message
    session_id = request_data.session_id

    user_message = Message(content=user_message_content, sender="user")

    try:
        response_message = await adk_agent.handle_message(user_message, sessio
n_id)
        return {"response": response_message.content}
    except Exception as e:
        logger.error(f"Error handling agent request for session {session_id}:
{e}", exc_info=True)
        # Raise HTTPException to return a proper HTTP 500 error to the client
        raise HTTPException(status_code=500, detail=f"Internal server error
processing request: {e}")

# Note: The `if __name__ == '__main__':` block is not needed when using
Uvicorn directly
# via the CMD command in the Dockerfile, as Uvicorn manages the server
startup.
```

Key Changes and Decisions in `main.py`:

- **FastAPI Integration:** The `@app.post('/agent')` decorator defines an HTTP POST endpoint. Cloud Run will route incoming requests to this.
- **Pydantic for Validation:** `AgentRequest` ensures that incoming JSON payloads have `message` (string) and `session_id` (string, with a default).
- **Persistent State Manager:** The `FirestoreStateManager` is initialized globally. It relies on the `GOOGLE_CLOUD_PROJECT` environment variable, which Cloud Run will inject.
- **`PersistentADKAgent`:** This custom agent class wraps the ADK `Agent` and explicitly uses our `FirestoreStateManager` for `load_state` and `save_state` operations, thereby providing long-term memory and pause/resume capabilities.
- **Structured Logging:** Replaced `print()` statements with Python's standard `logging` module. This is crucial for production, as Cloud Logging can better parse and filter structured log data.
- **Error Handling:** Added `try...except` blocks to catch potential issues during agent processing or Firestore interactions, returning a proper HTTP 500 response.

4. Build and Push Docker Image to Artifact Registry

First, configure Docker to authenticate with Google Cloud's Artifact Registry. Replace `us-central1` with your chosen region.

```
gcloud auth configure-docker us-central1-docker.pkg.dev
```

Next, build your Docker image and push it to Artifact Registry.

```
PROJECT_ID=$(gcloud config get-value project)

# Create an Artifact Registry repository if you haven't already
# The repository name 'adk-agent-repo' is a suggestion.
gcloud artifacts repositories create adk-agent-repo --repository-
format=docker \
    --location=us-central1 --description="Docker repository for ADK agents" \
    --async # Run in background to not block terminal

# Define the full image name
IMAGE_NAME="us-central1-docker.pkg.dev/${PROJECT_ID}/adk-agent-repo/adk-
persistent-agent:latest"

echo "Building Docker image: ${IMAGE_NAME}"
# Build the Docker image from the current directory
docker build -t "${IMAGE_NAME}" .

echo "Pushing Docker image to Artifact Registry..."
# Push the Docker image to Artifact Registry
docker push "${IMAGE_NAME}"
echo "Docker image pushed successfully."
```

Explanation:

- `gcloud artifacts repositories create`: Sets up a Docker repository in Artifact Registry to store your container images. This is a one-time setup.
- `docker build -t "${IMAGE_NAME}" .`: Builds the Docker image based on your `Dockerfile` in the current directory and tags it with the specified name.
- `docker push "${IMAGE_NAME}"`: Uploads the built image to the Artifact Registry, making it accessible for Cloud Run deployments.

5. Secure Secrets with Secret Manager

If your agent requires any sensitive configuration (e.g., an external API key for a third-party tool) that isn't handled by Google Cloud service accounts, use Secret Manager. For this project, the ADK often leverages service account credentials for Google Cloud services directly, reducing the need for explicit API keys for Google services.

Let's assume you have a hypothetical `EXTERNAL_TOOL_API_KEY` for a non-Google service.

```
PROJECT_ID=$(gcloud config get-value project)
SERVICE_ACCOUNT_EMAIL="adk-agent-runner@$
{PROJECT_ID}.iam.gserviceaccount.com" # From step 2

echo "Creating secret 'EXTERNAL_TOOL_API_KEY' in Secret Manager..."
# Create a secret in Secret Manager with a dummy value
echo "my-super-secret-key-value-123" | gcloud secret-manager secrets create EX
TERNAL_TOOL_API_KEY \
    --project="${PROJECT_ID}" --data-file=-

echo "Granting service account access to secret..."
# Grant the service account access to this specific secret
gcloud secret-manager secrets add-iam-policy-binding EXTERNAL_TOOL_API_KEY \
    --project="${PROJECT_ID}" \
    --member="serviceAccount:${SERVICE_ACCOUNT_EMAIL}" \
    --role="roles/secretmanager.secretAccessor"

echo "Secret setup complete."
```

Explanation:

- `gcloud secret-manager secrets create`: Creates a new secret named `EXTERNAL_TOOL_API_KEY`. The `--data-file=-` reads the secret value from standard input.
- `gcloud secret-manager secrets add-iam-policy-binding`: Grants your `adk-agent-runner` service account the `roles/secretmanager.secretAccessor` role for this specific secret. This ensures the principle of least privilege.

6. Deploy to Cloud Run

Now, deploy your containerized agent to Cloud Run. We'll link the dedicated service account, set environment variables, and inject the secret.

```
PROJECT_ID=$(gcloud config get-value project)
SERVICE_ACCOUNT_EMAIL="adk-agent-runner@$
{PROJECT_ID}.iam.gserviceaccount.com" # From step 2
IMAGE_NAME="us-central1-docker.pkg.dev/${PROJECT_ID}/adk-agent-repo/adk-
persistent-agent:latest" # From step 4

echo "Deploying 'adk-persistent-agent' to Cloud Run..."
gcloud run deploy adk-persistent-agent \
    --image "${IMAGE_NAME}" \
    --platform managed \
    --region us-central1 \
    --allow-unauthenticated \
    --service-account "${SERVICE_ACCOUNT_EMAIL}" \
    --set-env-vars GOOGLE_CLOUD_PROJECT="${PROJECT_ID}" \
    --update-secrets EXTERNAL_TOOL_API_KEY=EXTERNAL_TOOL_API_KEY:latest \
```

```
    --memory 512Mi \
    --cpu 1 \
    --timeout 300s \
    --min-instances 0 \
    --max-instances 10

echo "Deployment initiated. Cloud Run will provide a service URL upon
completion."
```

Explanation of parameters:

- `adk-persistent-agent`: The chosen name for your Cloud Run service.
- `--image "${IMAGE_NAME}"`: Specifies the Docker image to deploy from Artifact Registry.
- `--platform managed`: Indicates you're using the fully managed Cloud Run service, Google handles the underlying infrastructure.
- `--region us-central1`: The Google Cloud region for deployment. Choose one close to your users.
- `--allow-unauthenticated`: Makes the service publicly accessible. For internal applications, you'd omit this and manage access with IAM.
- `--service-account "${SERVICE_ACCOUNT_EMAIL}"`: Assigns the dedicated service account created earlier. This account's permissions dictate what your agent can do on Google Cloud (e.g., access Firestore, Secret Manager).
- `--set-env-vars GOOGLE_CLOUD_PROJECT="${PROJECT_ID}"`: Passes your project ID as an environment variable to the container. Your `FirestoreStateManager` uses this for initialization.
- `--update-secrets EXTERNAL_TOOL_API_KEY=EXTERNAL_TOOL_API_KEY:latest`: Injects the `EXTERNAL_TOOL_API_KEY` secret as an environment variable named `EXTERNAL_TOOL_API_KEY` into your container. The `:latest` ensures it always uses the most recent version of the secret.
- `--memory 512Mi`: Sets the memory limit for each instance. Adjust based on your agent's resource needs.
- `--cpu 1`: Allocates 1 CPU core per instance.
- `--timeout 300s`: Maximum request processing time. ADK agents with complex tool use or long LLM calls might need longer.
- `--min-instances 0`: Allows the service to scale down to zero instances when idle, significantly saving costs.

- `--max-instances 10`: Sets the maximum number of instances Cloud Run can spin up to handle traffic. Adjust based on expected load.

After deployment, Cloud Run will provide a URL for your service in the terminal output. Note this URL.

7. Configure Logging and Monitoring

Cloud Run automatically integrates with Cloud Logging. All `print()` statements and logs from your Python application (especially those using the standard `logging` module, as we've updated `main.py`) will appear in Cloud Logging.

Viewing Logs:

1. Navigate to the Google Cloud Console.
2. Go to **Logging** > **Logs Explorer**.
3. Filter by `resource.type="cloud_run_revision"` and `resource.labels.service_name="adk-persistent-agent"`.

You'll see your agent's `INFO` and `ERROR` log messages, FastAPI server logs, and any Python tracebacks.

Basic Monitoring:

Cloud Run also provides basic monitoring metrics (request count, latency, error rates) directly on the Cloud Run service details page in the Google Cloud Console. For advanced monitoring and custom alerts, you can use Cloud Monitoring (formerly Stackdriver Monitoring) to create dashboards and trigger notifications based on these metrics or even specific log patterns.

Testing & Verification

Now that your agent is deployed, let's verify its functionality and context persistence.

1. **Access the Agent:** Use `curl` or a tool like Postman to send `POST` requests to your Cloud Run service URL (e.g., `<https://adk-persistent-agent-xxxxxxx-uc.a.run.app/agent>`). Remember to replace `YOUR_CLOUD_RUN_URL` with the actual URL provided by Cloud Run after deployment.

```
# Replace with your actual Cloud Run URL
AGENT_URL="YOUR_CLOUD_RUN_URL/agent"

echo "--- First interaction (new session for user123) ---"
curl -X POST -H "Content-Type: application/json" \
```

```
        -d '{"message": "Hello ADK!", "session_id": "user123"}' \
        "${AGENT_URL}"

    echo ""
    echo "--- Second interaction (same session for user123) ---"
    curl -X POST -H "Content-Type: application/json" \
        -d '{"message": "How are you doing?", "session_id": "user123"}' \
        "${AGENT_URL}"

    echo ""
    echo "--- First interaction (new session for user456) ---"
    curl -X POST -H "Content-Type: application/json" \
        -d '{"message": "Who are you?", "session_id": "user456"}' \
        "${AGENT_URL}"
```

1. **Verify Context Persistence:**
    - **Observe Responses:** Check the responses from the `curl` commands. For `user123`, the second message should correctly acknowledge the content of the first message, demonstrating that the agent loaded the previous context. The response should resemble: `"You said: 'How are you doing?'. Previously, on [timestamp], you said: 'Hello ADK!'."`
    - **Inspect Firestore:** Go to your Firestore console in Google Cloud. You should see a collection named `adk_agent_sessions` (or whatever you configured in `main.py`). Within this collection, there should be documents corresponding to `user123` and `user456`, each containing their respective `last_message` and `timestamp` fields. This confirms state is being written and read from the external store.
2. **Check Cloud Run Logs:**
    - Go to **Logging** > **Logs Explorer** and filter for your `adk-persistent-agent` service.
    - You should see log entries for each request, including the agent's internal `logger.info` statements (e.g., "Agent received message...", "Loaded persistent context...", "Saved updated persistent context..."). This confirms your agent is running, processing requests, and interacting with Firestore as expected. Look for any `logger.error` messages if something isn't working.

Operations and Production Readiness

Deploying is just the first step. Operating a production AI agent requires ongoing attention to observability, security, scalability, and cost.

Observability

- **Structured Logging:** As implemented in `main.py`, using Python's `logging` module is a good start. For even richer insights, consider formatting your logs as JSON. Cloud Logging can automatically parse JSON logs, allowing you to filter and query specific fields (e.g., `jsonPayload.session_id`, `jsonPayload.tool_call_details`, `jsonPayload.error_type`).
- **Metrics & Alerts:** Use Cloud Monitoring to create custom metrics from your logs (e.g., count of agent errors, latency of agent responses, number of tool calls). Set up alerts to notify your team via email, SMS, or PagerDuty if these metrics cross predefined thresholds.
- **Request Tracing:** For complex agents interacting with many services (databases, other APIs, multiple LLMs), integrate OpenTelemetry or OpenCensus. This enables distributed tracing, giving you end-to-end visibility into request flows across all services involved in an agent's response.

Security

- **IAM Least Privilege:** Regularly review and refine the permissions of your `adk-agent-runner` service account. Only grant the roles absolutely necessary for the agent's operation. Avoid broad roles like "Editor" or "Owner."
- **Secret Manager Policies:** Ensure only the Cloud Run service account can access the secrets it needs. Avoid granting broad "Secret Manager Admin" roles, which could expose all secrets.
- **Network Access:** If your agent needs to access private resources (e.g., a database in a private VPC network), configure a Serverless VPC Access connector for your Cloud Run service.
- **Input Validation & Guardrails:** Always validate and sanitize user inputs to prevent prompt injection attacks or unexpected behavior. While ADK provides some built-in safety features, custom tools or complex prompts might require additional, explicit checks within your agent's logic.

Scalability and Cost Management

- **Cold Starts:** When `min-instances` is 0 (as configured), Cloud Run scales down to zero instances when idle. The very first request after a period of inactivity will incur a "cold start" delay as a new instance spins up. For latency-sensitive applications with frequent but sporadic traffic, you might set `--min-instances` to 1 or higher to keep an instance warm, but this increases continuous cost.
- **Concurrency:** Cloud Run instances can handle multiple concurrent requests. The `--concurrency` setting (default is 80) determines how many requests a single instance can process simultaneously. Tune this based on your agent's average request processing time and resource usage to optimize performance and cost.
- **Cost Monitoring:** Regularly review your Google Cloud billing to understand Cloud Run, Firestore, and Secret Manager costs. Set budget alerts in the Google Cloud Console to avoid unexpected expenditures.

Common Issues & Solutions

1. **Deployment Fails with Permission Errors:**
    - **Issue:** The `gcloud run deploy` command fails, often due to the user or the Cloud Build service account (if using CI/CD) lacking sufficient permissions (e.g., `roles/run.admin` or `roles/artifactregistry.writer`).
    - **Solution:** Ensure the identity performing the deployment (your `gcloud` user) has the necessary permissions. The `adk-agent-runner` service account is for runtime, not deployment.
2. **Agent Not Responding / HTTP 500 Errors:**
    - **Issue:** The deployed agent returns HTTP 500 errors or doesn't respond as expected.
    - **Solution:**
        - **Check Cloud Run Logs:** This is your first and most important debugging step. Look for Python tracebacks, `ModuleNotFoundError` (ensure `requirements.txt` is complete and `pip install` ran correctly in the Dockerfile), or issues connecting to Firestore.
        - **Environment Variables:** Verify all required environment variables (e.g., `GOOGLE_CLOUD_PROJECT`) are correctly set in the Cloud Run service configuration. You can check this with `gcloud run services describe adk-persistent-agent --platform managed --region us-central1`.
        - **Service Account Permissions:** Double-check that the `adk-agent-runner` service account has `roles/datastore.user` (for Firestore) and `roles/secretmanager.secretAccessor` (if using Secret Manager).
        - **Port Listening:** Confirm your `main.py` FastAPI app is correctly exposed via Uvicorn, listening on `0.0.0.0` and the port provided by Cloud Run (default 8080).

3. **Context Loss / State Not Persisting:**
    - **Issue:** The agent doesn't remember previous conversational context across sessions or requests, despite the setup.
    - **Solution:**
        - **Firestore Connectivity:** Check Cloud Run logs for any errors connecting to Firestore. Verify `GOOGLE_CLOUD_PROJECT` is correctly passed and that the service account has `roles/datastore.user`.
        - **Firestore Data:** Manually inspect your Firestore collection (`adk_agent_sessions`) in the Google Cloud Console to see if state is actually being written and read for the correct `session_id`.
        - **Agent Logic:** Double-check your `PersistentADKAgent`'s `handle_message` method to ensure it's correctly calling `self.persistent_state_manager.load_state()` and `self.persistent_state_manager.save_state()` with the appropriate `session_id`.
4. **Slow Responses (Cold Starts):**
    - **Issue:** Initial requests to the agent are slow, especially after periods of inactivity (e.g., the first request of the day).
    - **Solution:** This is a known characteristic of serverless services configured with `min-instances=0`. If the cold start delay is unacceptable for your application's latency requirements, set `--min-instances` to `1` or higher on your Cloud Run service. Be aware that this will incur continuous costs, even when the service is idle.

Summary & Next Step

Congratulations! You have successfully deployed your long-running, context-aware ADK agent to Google Cloud Run, secured its secrets with Secret Manager, and established fundamental logging and monitoring. This agent is now running in a scalable, reliable, and observable production-like environment. You've moved beyond a local prototype to a robust system ready for real users.

The project is now fully functional and production-ready. From here, you might consider:

- **CI/CD Integration:** Automate your build, test, and deployment process using Cloud Build, GitHub Actions, or GitLab CI to streamline future updates.
- **Advanced Monitoring:** Set up custom dashboards, detailed alerts, and potentially integrate with APM (Application Performance Monitoring) tools for deeper insights into agent performance and health.
- **User Interface:** Build a frontend application (web or mobile) to interact with your deployed agent API, creating a complete user experience.
- **A/B Testing:** Implement mechanisms to test different agent versions, prompt strategies, or tool configurations in production to optimize performance and user satisfaction.

This concludes our journey of building a long-running AI agent with Google ADK that can maintain context and state across sessions. You now have a solid foundation for developing and operating sophisticated AI applications.

References

- Google Cloud Run Documentation
- Google Cloud Artifact Registry Documentation
- Google Cloud Secret Manager Documentation
- Google Cloud IAM Documentation
- Google Cloud Firestore Documentation
- FastAPI Documentation
- Uvicorn Documentation
- awesome-adk-agents: Curated collection of ... - GitHub
- raphaelmansuy/adk_training: Google ADK Training Hub - GitHub

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Building Persistent AI Agents with Google ADK: Pause, Resume, Recover

Building Persistent AI Agents with Google ADK: Pause, Resume, Recover

Imagine an AI agent assisting a customer, gathering information, and then needing to pause its work—perhaps the customer needs to find a document, or the agent needs to wait for an external system. If that agent loses all memory of the conversation and its current task when it pauses, it's not truly helpful. This guide addresses that critical challenge: building AI agents that can maintain context and state across sessions, allowing for seamless pause, resume, and recovery from interruptions without losing valuable information.

Why Persistent Agents Matter

In real-world applications, AI agents are increasingly tasked with complex, multi-step workflows. These can range from orchestrating business processes and managing customer support interactions to handling long-running data analysis tasks. A stateless agent, while simpler to build initially, quickly hits limitations:

- **Context Loss:** Every interaction starts from scratch, leading to repetitive questions and user frustration.
- **Workflow Interruption:** If a process requires human input or external system delays, the agent cannot gracefully wait and resume.
- **Lack of Reliability:** System crashes or redeployments mean losing all in-progress work.

By implementing persistence, we empower agents to be more reliable, user-friendly, and capable of tackling more sophisticated, long-duration tasks. This guide focuses on Google's Agent Development Kit (ADK) to build such robust agents, leveraging Google Cloud services for durable state management and scalable deployment.

Project Goal: A Resilient ADK Agent System

Our objective is to construct a production-minded AI agent system using Google ADK that can:

1. **Engage in multi-turn conversations** while retaining context.
2. **Persist its internal state and conversational memory** to an external, durable store.
3. **Gracefully pause and resume** its operation, reloading its exact state and context.
4. **Recover from unexpected interruptions** (e.g., application restarts, network issues) without losing progress.
5. **Be deployable and observable** in a cloud environment.

By the end of this guide, you will have built a functional, containerized ADK agent deployed on Google Cloud, demonstrating robust state management and a clear path to production readiness.

Core Technology Stack

We'll use a pragmatic and powerful set of technologies:

- **Python:** The primary language for agent development, known for its rich ecosystem and ease of use with AI frameworks.
- **Google Agent Development Kit (ADK):** Google's framework for building sophisticated, multi-turn AI agents. It provides abstractions for tools, memory, and orchestrating agent behavior.
- **Google Cloud Platform (GCP):** For hosting our agent and providing durable services like:
    - **Firestore:** A NoSQL document database for persisting agent state and conversational history. Its flexible data model and real-time capabilities make it an excellent choice for dynamic agent memory.
    - **Cloud Run:** A fully managed compute platform for deploying containerized applications. It scales automatically and handles infrastructure, letting us focus on the agent logic.
    - **Cloud Logging & Monitoring:** For observing agent behavior and diagnosing issues in production.

Architectural Blueprint

A resilient agent system requires careful architectural planning. We will focus on these key principles:

- **Decoupled State:** The agent's core logic will be separated from its state persistence mechanism. This allows us to swap out storage solutions without rewriting the agent.
- **External Durable Storage:** Instead of relying on in-memory state, we'll use a persistent database (Firestore) to ensure state survives restarts and can be accessed by different instances.
- **Resumable Workflows:** Agent workflows will be designed to be idempotent or resumable from any logical checkpoint, minimizing data loss if an operation is interrupted.
- **Containerization:** Packaging the agent in a Docker container ensures portability and consistency across development, testing, and production environments.
- **Observability:** Integrating logging and monitoring from the start to understand agent behavior, track state changes, and quickly identify and resolve issues.

Prerequisites

To follow along with this guide, you should have:

- **Python 3.x:** A working Python environment. The exact latest stable version should be confirmed from official Python documentation as of 2026-05-23.
- **Google Cloud Account:** An active Google Cloud account with billing enabled.
- **Google Cloud Project:** A new or existing Google Cloud project where you have owner or editor permissions.
- **Git:** Basic familiarity with Git for version control.
- **Docker Desktop:** For containerizing your application locally.

Setting Up Your Workspace

Before we dive into agent development, let's prepare our environment:

1. **Install Python:** If not already installed, download and install the latest stable version of Python 3.x from python.org.
2. **Create a Virtual Environment:**

```
    python3 -m venv .venv
    source .venv/bin/activate  # On Windows, use: .venv\Scripts\activate
```

1. **Install Google ADK:** The exact latest stable version of Google ADK should be confirmed from its official documentation as of 2026-05-23. Install it via pip:

```
    pip install google-generative-ai  # This is a common dependency for ADK-
like frameworks.
    pip install google-cloud-firestore
    # Placeholder for actual ADK package name, if different:
    # pip install google-adk # (Verify actual package name from official ADK
docs)
```

```
*Note: As of 2026-05-23, the specific official ADK package name for Google's
Agent Development Kit was not definitively found in public search results.
Please refer to Google's official AI documentation or SDK releases for the
precise package name and installation instructions.*
```

1. **Google Cloud SDK:** Install the `gcloud` CLI tool from the Google Cloud SDK documentation.
2. **Initialize gcloud:**

```
    gcloud init
    gcloud auth login
    gcloud config set project YOUR_PROJECT_ID
```

```
Replace `YOUR_PROJECT_ID` with your actual Google Cloud project ID.
```

1. **Enable APIs:** Ensure the necessary APIs are enabled for your project:
    - Cloud Firestore API
    - Cloud Run API
    - Cloud Build API (for container deployment)
    - Cloud Logging API
    - Cloud Monitoring API You can enable them via the Google Cloud Console or using `gcloud`:

```
    gcloud services enable firestore.googleapis.com run.googleapis.com cloudbu
ild.googleapis.com logging.googleapis.com monitoring.googleapis.com
```

What You'll Achieve

Upon completing this guide, you will:

- Understand the principles of state management and context persistence for AI agents.
- Be proficient in setting up a Python and Google Cloud environment for ADK development.
- Implement an ADK agent with external state storage using Firestore.
- Design and implement pause/resume capabilities for complex agent workflows.
- Learn to containerize your agent with Docker.
- Gain experience deploying, logging, and monitoring your agent on Google Cloud Run.
- Develop a mindset for building production-ready, resilient AI systems.

Learning Path

This guide is structured into incremental milestones, each building upon the last to construct a fully functional and resilient AI agent system.

Setting Up Your ADK Agent Development Environment

Configure your Python environment, Google Cloud project, and install the Google ADK to prepare for agent development.

Building a Basic, Stateless ADK Agent

Develop a foundational ADK agent capable of simple, stateless conversational interactions to understand core components.

Implementing Persistent Agent State with External Storage

Integrate an external Google Cloud database (e.g., Firestore) to store and retrieve agent state, enabling memory beyond a single session.

Designing for Context Preservation and Resume Capabilities

Implement mechanisms to serialize and deserialize conversational context and agent state, allowing the agent to pause, save, and resume its workflow.

Enhancing Agent Intelligence with Tools and Multi-Step Workflows

Extend the agent's capabilities by integrating external tools and constructing complex, multi-turn workflows that leverage its persistent context.

Containerizing Your ADK Agent for Portability and Scalability

Package your ADK agent application into a Docker container, making it portable and ready for cloud deployment.

Robust Testing for Long-Running Agent Workflows

Write unit, integration, and end-to-end tests to ensure the agent's state persistence, context retrieval, and pause/resume functionality work reliably.

Deploying and Monitoring Your Production ADK Agent on Google Cloud

Deploy the containerized ADK agent to a Google Cloud service like Cloud Run, configure logging, monitoring, and implement basic security practices.

References

- Google Cloud SDK Documentation
- Python Official Website
- Google Cloud Firestore Documentation
- Google Cloud Run Documentation
- awesome-adk-agents - GitHub
- raphaelmansuy/adk_training - GitHub

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.