Google Cloud

Building AI agents with Google's Agent Development Kit (ADK) as MCP client – a deep dive (full code)

Written by Arjun Prabhulal
Find more Google Cloud community content on Medium

Google Cloud Next '25 unveiled several groundbreaking announcements, and I had the privilege of attending the event in person. It was an inspiring experience filled with innovation, hands-on demos, product showcases, and insightful discussions.

While last year's spotlight was on genAI (chatbots) and Vertex AI, this year's theme was loud and clear – **agents, agents and more agents**. From Agent Development Kit (**ADK**) to Agent to Agent (**A2A**) Protocol and **Agentspace** along with Google Flagship LLM – Gemini 2.5 Pro Preview

In my **previous article**, I explored how to integrate the Gemini LLM using the **Model Context Protocol (MCP)** acting as **MCP client**, enabling structured, tool-augmented interactions with external APIs and systems. We also discussed the core Concept of MCP, a use case, and a demo.

In this article, our focus will be leveraging **existing MCP Servers** through **ADK agent** acting as **MCP client** using Gemini LLM , enabling tool invocation using capabilities provided by external MCP-powered tools. We will discuss the core concepts of **ADK** before diving into code implementation.

What is Agent Development Kit, or ADK?

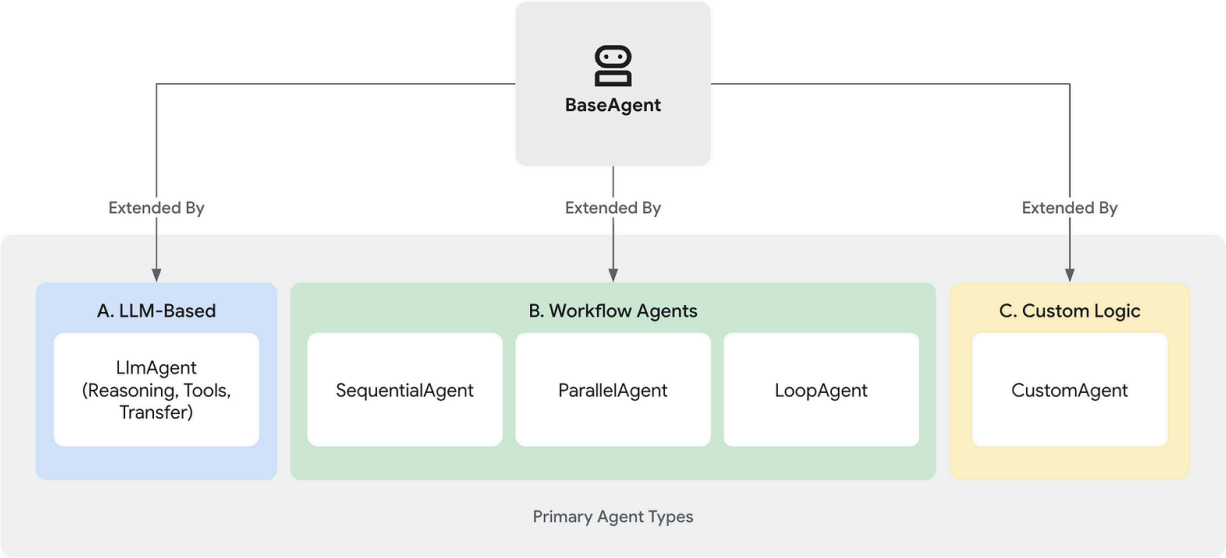
ADK is an open-source, code-first Python toolkit for building, evaluating, and deploying intelligent AI agents.

> ADK enables developers to create agentic workflows – from simple single-agent tasks to complex multi-agent orchestration – all within a modular and extensible framework.

What are agents in ADK ?

An **agent** is an autonomous, self-contained execution unit designed to achieve specific goals. Agents can do the following things:

1. Perform tasks.
2. Interact with users.
3. Leverage external tools.
4. Collaborate with other agents to complete complex workflows.



https://google.github.io/adk-docs/agents/

Core agent categories

ADK offers three primary agent types to support

- **LLM agents** *(e.g., LlmAgent, Agent)*: Use LLMs to understand, reason, plan, and act. They are ideal for dynamic, language-driven tasks.
- **Workflow agents** *(e.g., SequentialAgent, ParallelAgent, LoopAgent)*: Orchestrate other agents in predictable patterns without relying on an LLM for flow control. They are best for structured, repeatable processes.
- **Custom agents**: Built by extending BaseAgent to enable custom logic, specialized workflows, or unique tool integrations. They are perfect for advanced, tailor-made solutions.

In this article we will be using **LLM agents** with **MCPTools**

What is a tool in the context of ADK?

A **tool** represents a specific capability granted to an AI agent, allowing it to perform actions and interact with the external world beyond basic text generation and reasoning.

> A tool is usually a modular code component—such as a Python function, class method, or even another agent—designed to carry out a defined task.

How do agents use tools?

Google Cloud

Agents dynamically leverage tools through **function-calling mechanisms**, where the LLM reasons over context, selects and invokes the appropriate tool with generated inputs, observes the result, and integrates the output into its next action or response.

Tool types in ADK

ADK supports several types of tools

1. **Function tools**: Custom tools built specifically for our application's unique logic and workflows.
- **Functions or methods**: Standard synchronous Python functions (def) or class methods registered as tools.
- **Agents as tools**: Use specialized agents as callable tools within a parent agent for modular behavior.
- **Long-running function tools**: Tools designed for asynchronous or time-intensive operations.

2. Built-in tools: Predefined tools included in the framework for tasks like web search, code execution, or RAG.

3. Third-party tools: Easily integrate tools from popular ecosystems like LangChain or CrewAI.

Architecture

We will use the same architecture as in the earlier article and, in addition, we will use ADK.

ADK (Agent Development Kit) + MCP (Model Context Protocol) with Gemini 2.5 Pro LLM

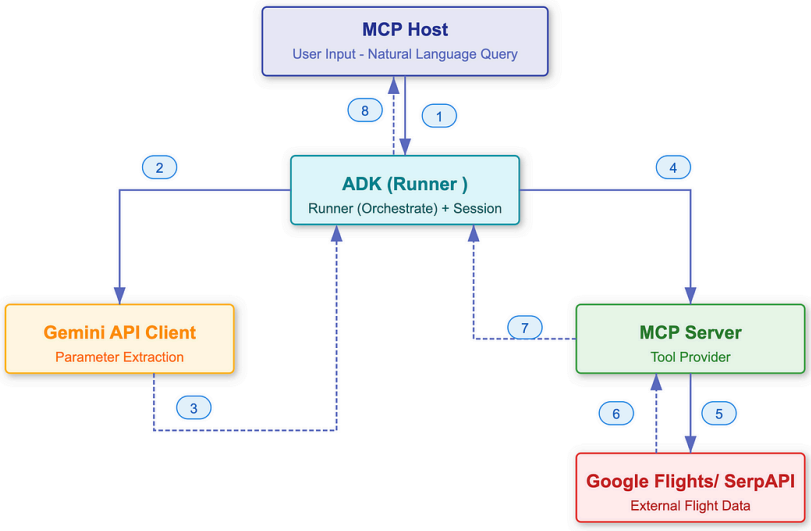


Image by Author

Google Cloud

Implementation

Let us dive into building this pipeline with ADK + MCP + Gemini AI by breaking down into key implementation steps

Pre-requisites

1. **Python 3.8+** installed

2. Google **Gemini** Generative AI access via API key

3. A valid **SerpAPI key** (used to fetch live flight data)

Step 1: Setup a virtual environment

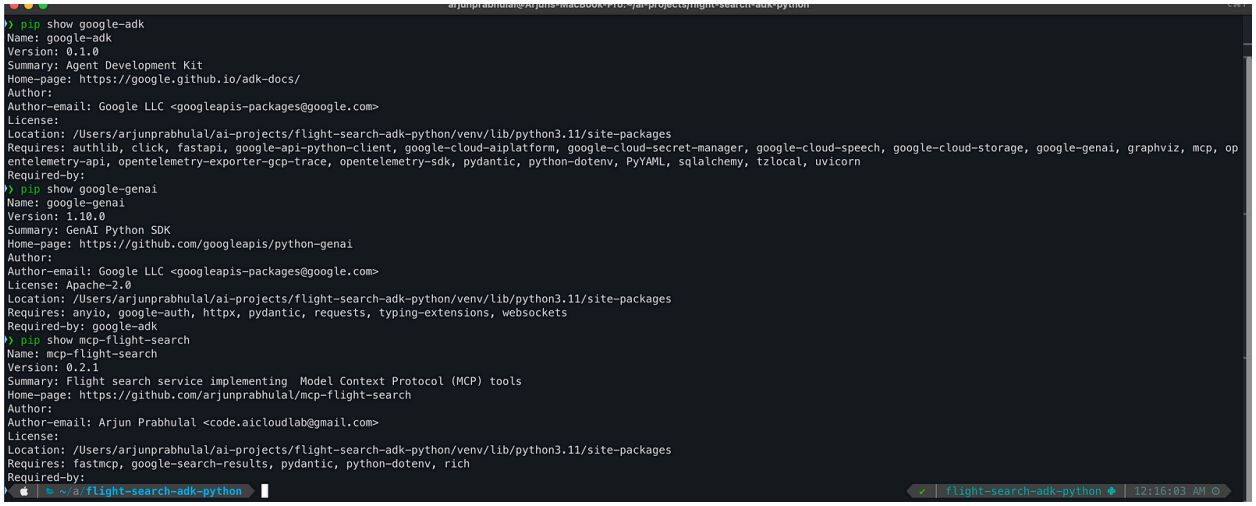
Install the dependencies using

```
# Setup virtual environment (mac or Unix )
python -m venv venv && source venv/bin/active

# Install agent development kit
pip install google-adk

# Install MCP server
pip install mcp-flight-search

# Install GenAI Python SDK
pip install google-genai
```

google-sdk : Google's Agent Development Kit for building agents.

google-genai : Google's library for interacting with Generative AI models (like Gemini).

mcp-flight-search : MCP server which uses SerpAPI to fetch flight using MCP library



Set environment variables :

Note the difference here. Instead of GEMINI_API_KEY, it will be GOOGLE_API_KEY in ADK

export GOOGLE_API_KEY="your-google-api-key"
export SERP_API_KEY="your-serpapi-key"

Step 2: Install the MCP server, mcp-flight-search

To enable Gemini to interact with real-world APIs, we'll use an MCP-compliant server

For this article, we'll use **mcp-flight-search**, a lightweight MCP server built using FastMCP which exposes a tool that searches real-time flight data using the SerpAPI.

Verify the installed MCP server package at https://pypi.org/project/mcp-flight-search/

Install from PyPI (already installed from step 1)
pip install mcp-flight-search

Step 3: Understanding ADK as an MCP client

from google.adk.agents.llm_agent import LlmAgent
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset, StdioServerParameters

LlmAgent is a core component in ADK, acting as the "thinking" part of our application. It leverages the power of a large language model (LLM) for reasoning, understanding natural language, making decisions, generating responses, and interacting with tools.

```
🔍 DEBUG: Creating LlmAgent...
🔍 DEBUG: name='flight_search_assistant' description='' parent_agent=None sub_agents=[] before_agent_callback=None after_agent_callback=None model='gemini-2.5-pro-exp-03-25'
instruction='Help user to search for flights using available tools based on prompt. If return date not specified, use an empty string for one-way trips.' global_instruction=''
tools=[<google.adk.tools.mcp_tool.mcp_tool.MCPTool object at 0x1196d9e90>, <google.adk.tools.mcp_tool.mcp_tool.MCPTool object at 0x1197b6750>] generate_content_config=None
disallow_transfer_to_parent=False disallow_transfer_to_peers=False include_contents='default' input_schema=None output_schema=None output_key=None planner=None code_executor=None
examples=None before_model_callback=None after_model_callback=None before_tool_callback=None after_tool_callback=None
✅ SUCCESS: LlmAgent created successfully
```

Runner is responsible for coordinating the interactions between various components during an agent's lifecycle.Runner uses in-memory implementations for artifact, session, and memory services, providing a lightweight and self-contained environment for agent execution.

```
✅ SUCCESS: Agent and exit stack retrieved
🔍 DEBUG: Creating Runner...
🔍 DEBUG: <google.adk.runners.Runner object at 0x11990bf90>
✅ SUCCESS: Runner created
```

InMemorySessionService is an implementation of ADK's SessionService interface that stores all session data – such as conversation history, state, and metadata – directly in the application's memory. This means that all session information is lost when the application stops or restarts.

When a user interacts with an AI agent, a Session object is created to track the conversation.

```
🔍 DEBUG: Starting main execution logic...
🔍 DEBUG: <google.adk.sessions.in_memory_session_service.InMemorySessionService object at 0x11d20b0d0>
🔍 DEBUG: InMemorySessionService created
🔍 DEBUG: id='d8a5b315-ef35-4648-bbd3-0ede4812c02c' app_name='flight_search_app' user_id='user_flights' state={} events=[] last_update_time=1744357211.956345
✅ SUCCESS: Session created
──────────────────────────── Data ────────────────────────────
{'session_id': 'd8a5b315-ef35-4648-bbd3-0ede4812c02c', 'user_id': 'user_flights'}
```

MCPToolSet is a class within ADK that enables our AI agents to connect with external MCP servers. These servers expose tools—such as APIs or services—that agents can utilize to perform specific tasks. By using MCPToolset, agents can discover, invoke, and manage these external tools seamlessly.

```
🔍 DEBUG: [<google.adk.tools.mcp_tool.mcp_tool.MCPTool object at 0x11cbbaf90>, <google.adk.tools.mcp_tool.mcp_tool.MCPTool object at 0x11d201f90>]
✅ SUCCESS: MCP server connection established
✅ SUCCESS: Fetched 2 tools from MCP server
```

Fetched MCP Tools

Name	Description	Schema
search_flights_tool	Search for flights using SerpAPI Google Flights. This MCP tool allows AI models to search for flight information by specifying departure and arrival airports and travel dates. Args: origin: Departure airport code (e.g., ATL, JFK) destination: Arrival airport code (e.g., LAX, ORD) outbound_date: Departure date (YYYY-MM-DD) return_date: Return date for round trips (YYYY-MM-DD) Returns: A list of available flights with details	Schema not readily available
server_status	Check if the Model Context Protocol server is running. This MCP tool provides a simple way to verify the server is operational. Returns: A status message indicating the server is online	Schema not readily available

StdioServerParameters is a configuration class used to specify how the agent should connect to an MCP server via standard input/output streams. This is particularly useful when the MCP server is a local process that communicates through the console.

How do MCPToolSet and StdioServerParameters work together?

When combined, MCPToolset and StdioServerParameters allow an ADK agent to do the following:

1. **Establish a connection**: Using StdioServerParameters, define the command and arguments needed to start the MCP server process.
2. **Discover available tools**: MCPToolset connects to the MCP server and retrieves a list of available tools that the agent can use.
3. **Integrate tools into the agent**: The discovered tools are adapted into a format compatible with ADK agents, allowing seamless invocation during agent execution.
4. **Manage the Connection Lifecycle**: MCPToolset handles the setup and teardown of the connection to the MCP server, ensuring resources are managed appropriately.

Step 4: Connecting to MCP server

StdioServerParameters defines the MCP configuration to list and listen async using MCPToolSet

```
# --- Step 1: Get tools from MCP server ---
async def get_tools_async():
  """Gets tools from the Flight Search MCP Server."""
  print("Attempting to connect to MCP Flight Search server...")
  server_params = StdioServerParameters(
      command="mcp-flight-search",
      args=["--connection_type", "stdio"],
      env={"SERP_API_KEY": os.getenv("SERP_API_KEY")},
  )

  tools, exit_stack = await MCPToolset.from_server(
      connection_params=server_params
  )
  print("MCP Toolset created successfully.")
  return tools, exit_stack
```

Step 5: Agent creation from ADK

As mentioned above , we are using Llm agent which is the thinking part of application.

```
# --- Step 2: Define ADK Agent Creation ---
async def get_agent_async():
```

```
    """Creates an ADK Agent equipped with tools from the MCP Server."""
    tools, exit_stack = await get_tools_async()
    print(f"Fetched {len(tools)} tools from MCP server.")

    # Create the LlmAgent matching the example structure
    root_agent = LlmAgent(
        model=os.getenv("GEMINI_MODEL", "gemini-2.5-pro-preview-03-25"),
        name='flight_search_assistant',
        instruction='Help user to search for flights using available tools based on prompt. If return
date not specified, use an empty string for one-way trips.',
        tools=tools,
    )

    return root_agent, exit_stack
```

Step 6: Integrating Agent creation, session management, and orchestration with Runner

```
async def async_main():
    # Create services
    session_service = InMemorySessionService()

    # Create a session
    session = session_service.create_session(
        state={}, app_name='flight_search_app', user_id='user_flights'
    )

    # Define the user prompt
    query = "Find flights from Atlanta to Las Vegas 2025-05-05"
    print(f"User Query: '{query}'")

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

    # Get agent and exit_stack
    root_agent, exit_stack = await get_agent_async()

    # Create Runner
    runner = Runner(
        app_name='flight_search_app',
        agent=root_agent,
        session_service=session_service,
    )

    print("Running agent...")
```

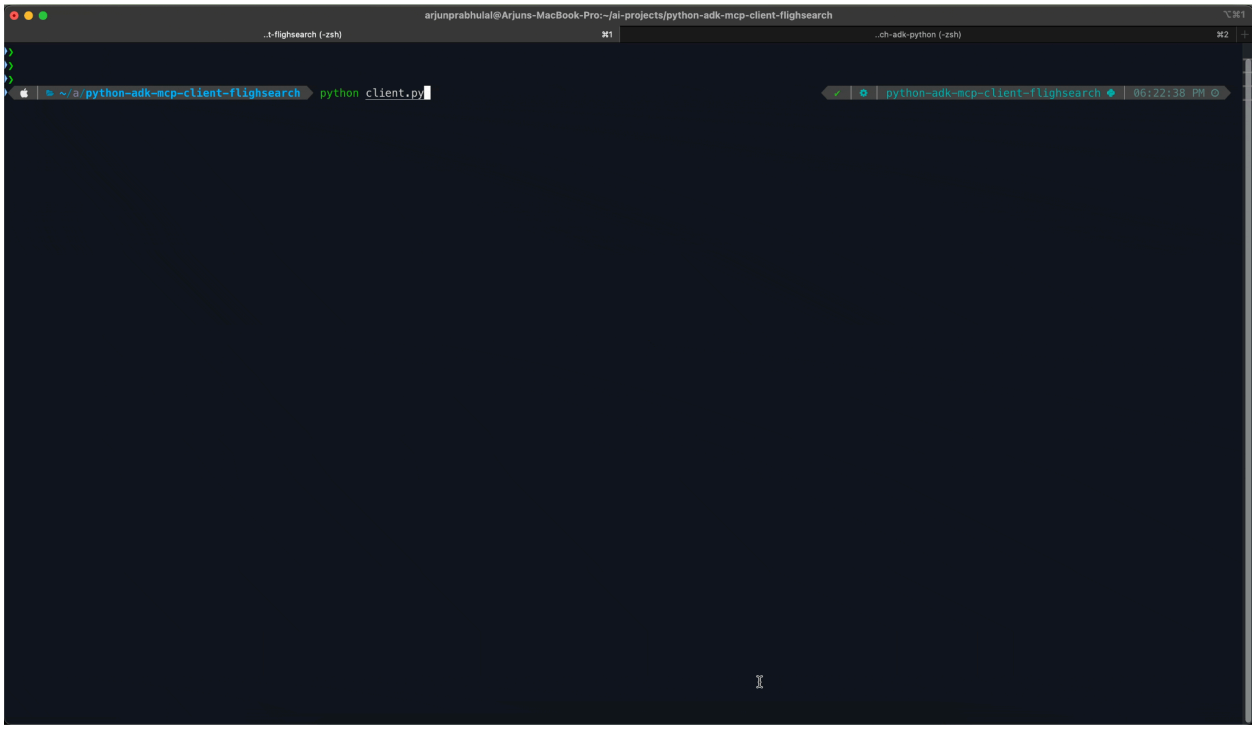
```
events_async = runner.run_async(
    session_id=session.id,
    user_id=session.user_id,
    new_message=content
)

# Process events
final_content = None
async for event in events_async:
    print(f"Event received: {event}")


# Always clean up resources
print("Closing MCP server connection...")
await exit_stack.aclose()
print("Cleanup complete.")
```

Step 7: Demo

User query in MCP Client :

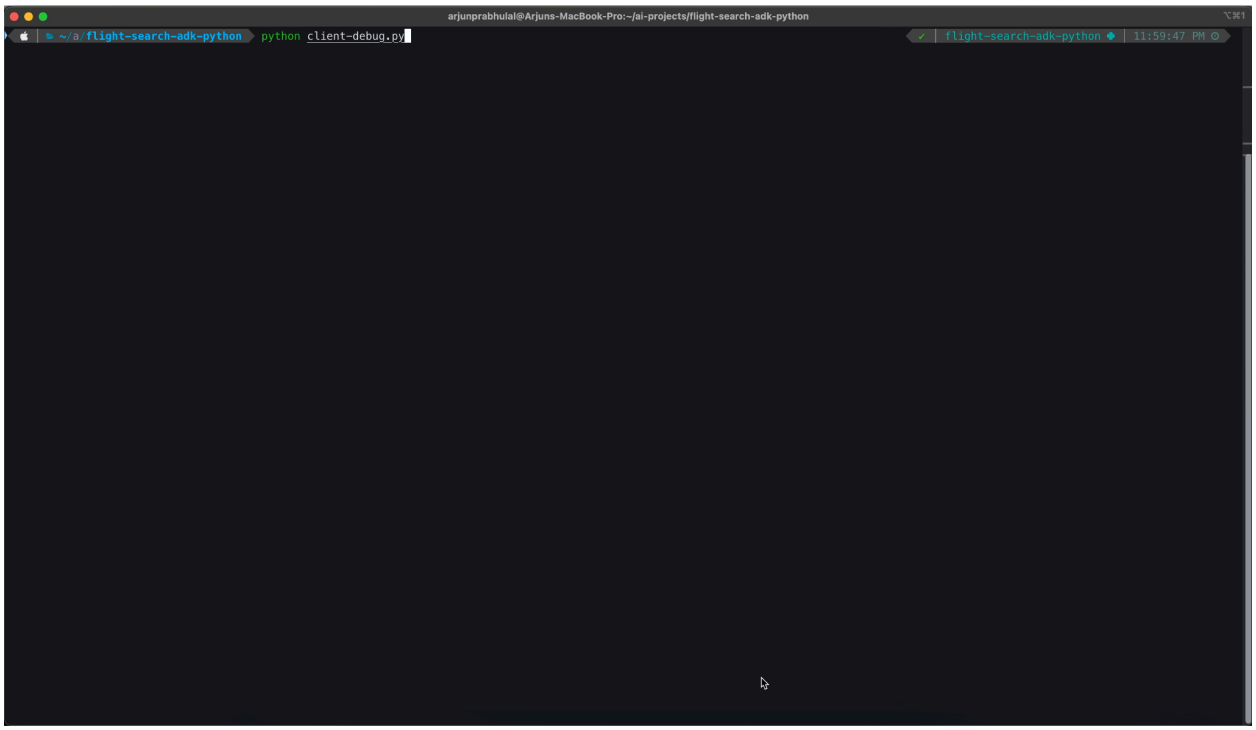
query = "Find flights from Atlanta to Las Vegas 2025-05-05"

Demo with standard logging



Google Cloud

Demo with debug logging



Key considerations for integrating MCP with ADK

1. **MCP versus ADK**

- **MCP is an** open protocol that standardizes how AI models interact with external tools and data sources.
- **ADK is a** Python-based framework for building and deploying AI agents.
- **MCPToolset** bridges MCP and ADK by enabling ADK agents to consume tools exposed by MCP servers.

To build an MCP server in Python, use the model-context-protocol library.

2. **Tool Types and Integration**

- **ADK tools**: Python objects (e.g., BaseTool, FunctionTool) designed for direct use within ADK agents.
- **MCP tools**: Capabilities exposed by MCP servers that **MCPToolset** adapts for use within ADK agents.
- **Third-party tools**: Libraries like **LangChain and CrewAI**, which offer tools that can be integrated into **ADK using wrappers like LangchainTool and CrewaiTool.**

3. **Asynchronous architecture**

- Both ADK and the MCP Python library are built on Python's Asyncio framework.

- Tool implementations and server handlers should be asynchronous (async def) to ensure non-blocking operations.

4. **Stateful sessions in MCP**

- MCP establishes persistent, stateful connections between clients and servers, unlike typical stateless REST APIs.
- This statefulness allows for context retention across interactions but requires careful session management.

5. **Deployment considerations**

- The persistent nature of MCP connections can pose challenges for scaling and deployment, especially for remote servers handling multiple users.
- Infrastructure considerations include load balancing and session affinity to maintain connection stability.

6. **Managing MCP connections in ADK**

- MCPToolset manages the lifecycle of MCP connections within ADK.
- Using an exit_stack ensures that connections are properly terminated when the agent's execution completes.

Troubleshooting:

1. By default, adk library expects GCP Project Vertex AI, location and VertexAI Configs. Make sure to use GOOGLE_API_KEY instead of GEMINI_API_KEY as the error message may not clearly indicate a missing env. value

Solution: Make sure to set variable as **GOOGLE_API_KEY.**

ValueError: Missing key inputs argument! To use the Google AI API,provide (`api_key`) arguments. To use the Google Cloud API, provide (`vertexai`, `project` & `location`) arguments.

2. Frequent 429 rate-limit error and 500 internal server.

Solution: Switch to Gemini 2 Flash as *"**The Gemini API "free tier" is offered through the API service with lower rate limits for testing purposes**."*

google.genai.errors.ClientError: 429 RESOURCE_EXHAUSTED. {'error': {'code': 429, 'message': 'You exceeded your current quota, please check your plan and billing details. For more information on this error, head to: https://ai.google.dev/gemini-api/docs/rate-limits.', 'status': 'RESOURCE_EXHAUSTED', 'details': [{'@type': 'type.googleapis.com/google.rpc.QuotaFailure', 'violations': [{'quotaMetric': 'generativelanguage.googleapis.com/generate_content_free_tier_requests', 'quotaId': 'GenerateRequestsPerDayPerProjectPerModel-FreeTier', 'quotaDimensions': {'model':

'gemini-2.0-pro-exp', 'location': 'global'}, 'quotaValue': '25'}]}, {'@type': 'type.googleapis.com/google.rpc.Help', 'links': [{'description': 'Learn more about Gemini API quotas', 'url': 'https://ai.google.dev/gemini-api/docs/rate-limits'}]}, {'@type': 'type.googleapis.com/google.rpc.RetryInfo', 'retryDelay': '27s'}]}}

An error occurred during execution: 500 INTERNAL. {'error': {'code': 500, 'message': 'An internal error has occurred. Please retry or report in https://developers.generativeai.google/guide/troubleshooting', 'status': 'INTERNAL'}}

Gemini API cost billing

> **Gemini 2.5 Pro Preview**: As per Google Docs, Gemini API free tier is offered through the API service with lower rate limits for testing purposes. The Gemini API paid tier comes with higher rate limits, additional features, and different data handling.

Preview models may change before becoming stable and generally available.

	Free Tier	Paid Tier, per 1M tokens in USD
Input price	Free of charge, use "gemini-2.5-pro-exp-03-25"	$1.25, prompts <= 200k tokens $2.50, prompts > 200k tokens
Output price (including thinking tokens)	Free of charge, use "gemini-2.5-pro-exp-03-25"	$10.00, prompts <= 200k tokens $15.00, prompts > 200k
Context caching price	Not available	Not available
Grounding with Google Search	Free of charge, up to 500 RPD	1,500 RPD (free), then $35 / 1,000 requests
Used to improve our products	Yes	No

Official documentation references

- **Agent Development Kit (ADK) Documentation**, a complete guide to the open-source AI agent framework integrated with Gemini and Google.
- **LLM Agents in ADK**, detailed documentation on implementing LLM agents. This includes defining identity, instructions, tools, and advanced configuration.
- **MCP Tools with ADK** is specific guidance on using ADK as an MCP client to connect to MCP servers.

GitHub repository:

You can access all the code used in this tutorial in my GitHub:

arjunprabhulal/adk-python-mcp-client
*Contribute to arjunprabhulal/adk-python-mcp-client development by creating an account on GitHub.*github.com

Conclusion

In this article, we explored how to use open source ADK to build an agent setup assistant using Model Context Protocol (MCP) with Google Gemini LLM as MCP Client.