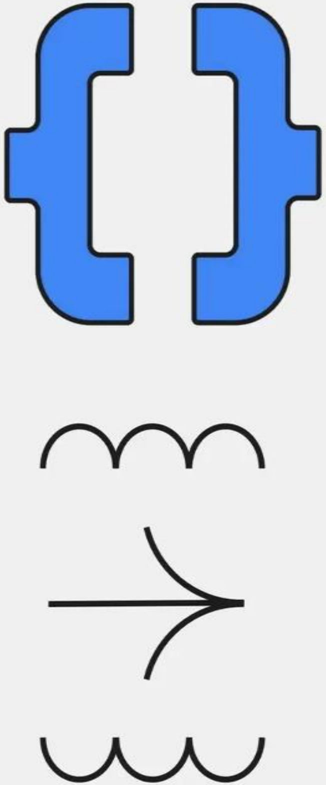
Agenda

- Welcome & The Agentic Future
- Introducing Google ADK & A2A: Solving Key Challenges
- Google ADK: Build Intelligent Agents
    - Core Concepts & Features
    - Quick Demo: Anatomy of an ADK Agent
- A2A Protocol: Enable Agent Collaboration
    - Core Concepts & How it Works
    - Quick Demo: A2A Interaction Snippet
- Synergy: ADK with A2A & Real-World Use Cases
- Getting Started & Your Next Steps



The Rise of AI Agents

What are AI Agents? More Than Just Code

Autonomous Entities: Operate independently to achieve goals.

Perceive: Understand their environment through data, sensors, or user input.

Reason: Process information, make decisions, and plan actions using AI models (especially LLMs).

Act: Execute actions, interact with systems, tools, or other agents.

Why the Surge Now? The LLM Revolution

Advanced large language models provide unprecedented reasoning, language understanding, and generation capabilities, forming the "brains" of modern agents.

The Vision: Collaborative Intelligence

Moving beyond single, monolithic AI systems to dynamic networks of specialized agents working together, much like human teams.

Key Challenges in Agent Development

Building Robust Agents:

Crafting reliable agent logic, ensuring consistent behavior, and managing complex internal states can be difficult.

Orchestration:

Coordinating multiple agents to perform a complex task requires clear workflows, data passing, and error handling across agents.

Interoperability:

How do agents built with different tools, by different teams, or even different companies, talk to each other effectively? Lack of standards creates silos.

Control & Safety:

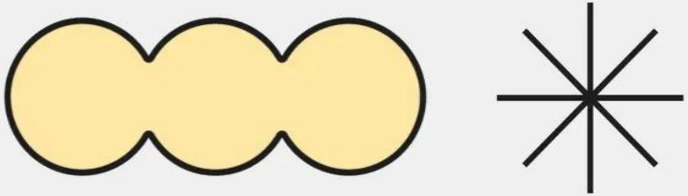
Ensuring agents operate within defined boundaries, use tools responsibly, and can be monitored and debugged is crucial for trust and adoption.

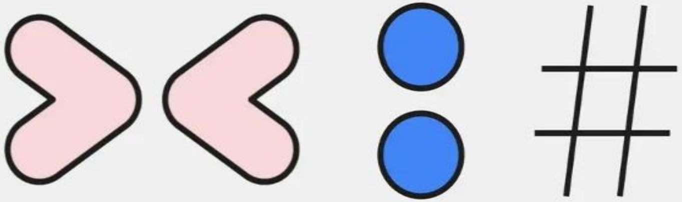
Google's Solution: A Powerful Duo

Google Agent Development Kit (ADK): Your Toolkit to BUILD

An open-source Python framework designed to simplify the creation, testing, and management of AI agents and multi-agent systems.

Purpose: Provides structure for agent logic, easy tool integration, robust orchestration capabilities, and pathways for deployment. Empowers developers with control.





Agent-to-Agent (A2A) Protocol: Your Bridge to CONNECT

An open, standardized communication protocol that defines how agents interact.

Purpose: Enables diverse AI agents – whether built with ADK, other frameworks, or custom code – to discover each other's capabilities, exchange information securely, and coordinate actions. Fosters an open ecosystem.

What is ADK?
Your Agent Building Toolkit

Open-Source & Python-Native:
Built for Python developers, offering a familiar and flexible environment.

Core Philosophy: The "Agent Triangle"

- **Instructions (Goal):** Clearly define what your agent should do, its personality, and its objectives using natural language.
- **Tools (Capabilities):** Equip agents with functions (Python code, APIs) to interact with the external world, fetch data, or perform specific actions.
- **Model (Brain):** Leverages the power of LLMs (like Gemini or your model of choice) for reasoning, planning, and language understanding to interpret instructions and use tools.

Goal: To streamline development from simple, single-purpose agents to complex, orchestrated multi-agent systems with built-in best practices.

ADK Key Features

Agent Types for Different Needs:

- LLMAgent: For agents driven by LLM reasoning.
- SequentialAgent, ParallelAgent, LoopAgent: For orchestrating workflows of multiple agents.
- CustomAgent: For more specialized agent behaviors.

Powerful Tool Integration:

- Use built-in tools like Google Search, Code Execution with minimal setup.
- Easily create custom Python tools to give agents unique skills.
- LongRunningFunctionTool handles tools that perform complex, time-consuming tasks and provide progress updates.

Flexible Orchestration:

- Design how multiple agents collaborate: assign a "planner" agent to delegate tasks to specialized "worker" agents.

Memory & Context Management:

- Enables agents to remember previous interactions and maintain context for coherent conversations and actions.

Observability & Debugging:

- Built-in logging helps you trace agent behavior, tool usage, and LLM interactions, making debugging easier.

Deployment Pathways:

- Develop locally, containerize with Docker, and deploy to cloud environments like Google Cloud Run or Vertex AI Agent Engine.

```python
from adk.agents import LLMAgent
from adk.tools import tool # Use the @tool decorator

# 1. Define a Tool: A simple Python function becomes a tool
@tool
def get_weather(city: str) -> str:
  """Gets the current weather for a specified city."""
  # In a real scenario, this would call a weather API
  if city.lower() == "london":
    return "It's likely cloudy with a chance of rain in London."
  return f"Weather data for {city} is not available with this mock tool."

# 2. Define the Agent: Combine instructions, an LLM, and tools
weather_assistant = LLMAgent(
    llm="gemini-1.5-flash", # Specify the LLM you want to use
    instructions="You are a helpful weather assistant. Use the get_weather tool to answer questions about weather.",
    tools=[get_weather] # Register the tool with the agent
)

# 3. Run the Agent (Conceptual - how you'd invoke it)
# query = "What's the weather like in London?"
# response = weather_assistant.run(query)
# print(f"User: {query}\nAgent: {response}")
```

Key Takeaways from the Code:

- @tool decorator: Simplifies making Python functions available to the agent.
- instructions: Shapes the agent's behavior and how it uses tools.
- tools list: Makes specific capabilities accessible to the LLM.
- ADK handles the underlying complexity of when and how the LLM calls the tool.

Quick Demo/Walkthrough

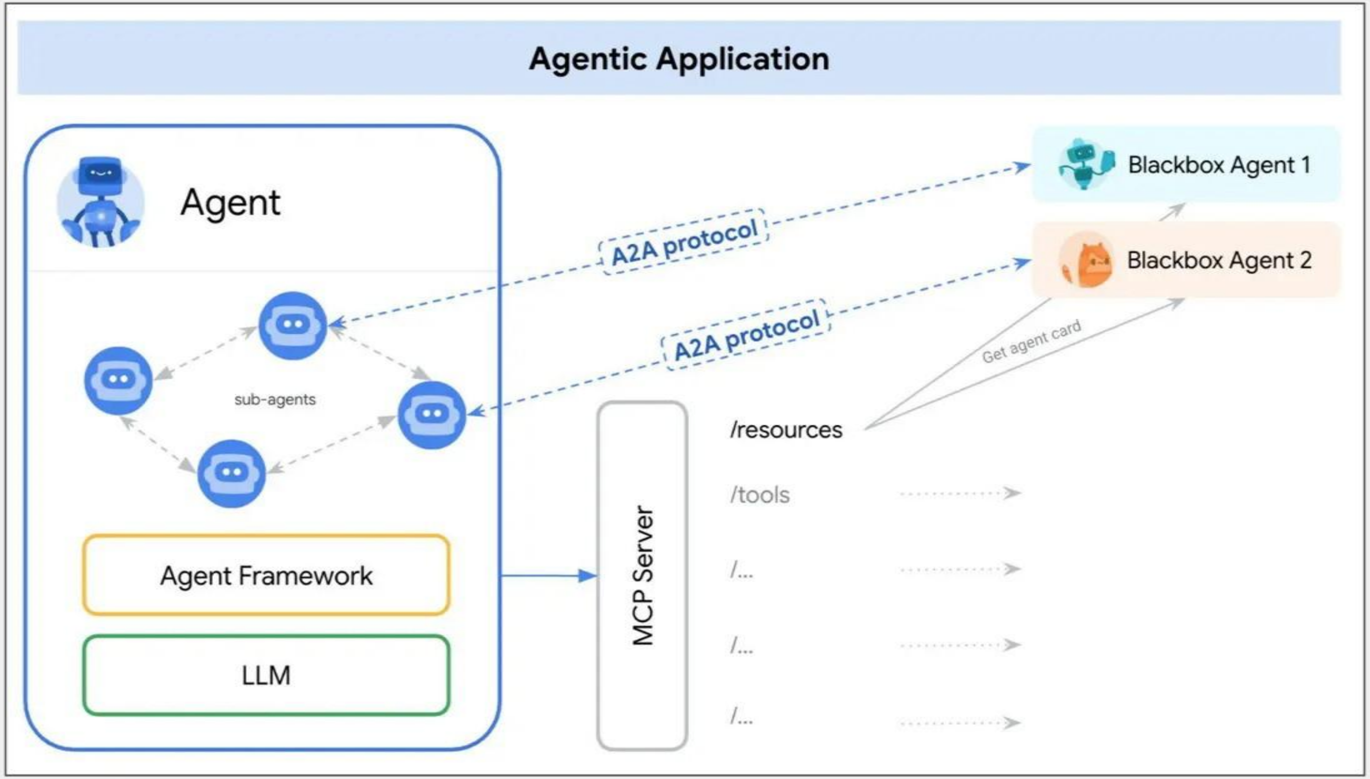
"Let's look at a very simple ADK agent in action. Here's a Python script similar to what we just saw, defining a weather_assistant."

"When I run this and ask, 'What's the weather like in London?', the ADK framework, powered by the Gemini model, understands it needs to use the get_weather tool." (Run the script with the query).

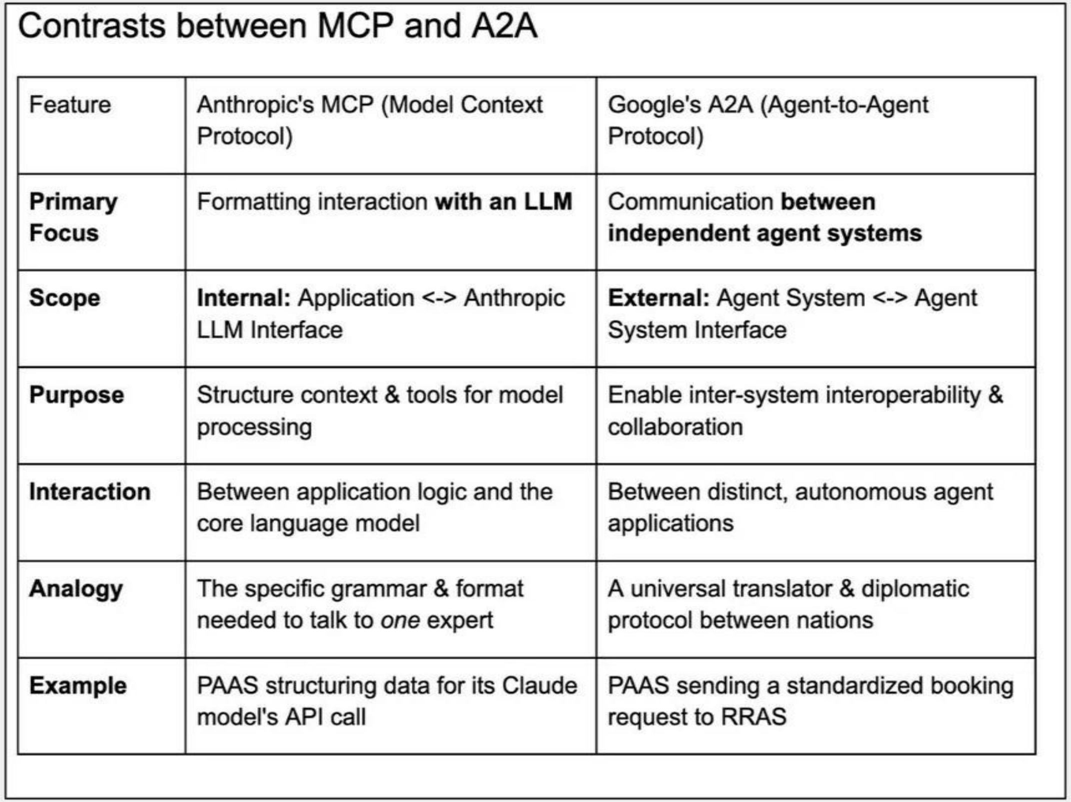
"You can see the output where the agent has (conceptually) called the tool and formulated a response. If we look at the logs ADK provides (show snippet if possible), we could trace this tool invocation."

A2A protocol

A2A vs MCP


Agentic Application
Agent
sub-agents
Agent Framework
LLM
A2A protocol
A2A protocol
Blackbox Agent 1
Blackbox Agent 2
Get agent card
MCP Server
/resources
/tools
/...
/...
/...

A2A vs MCP



Contrasts between MCP and A2A

Feature	Anthropic's MCP (Model Context Protocol)	Google's A2A (Agent-to-Agent Protocol)
Primary Focus	Formatting interaction **with an LLM**	Communication **between independent agent systems**
Scope	**Internal:** Application <-> Anthropic LLM Interface	**External:** Agent System <-> Agent System Interface
Purpose	Structure context & tools for model processing	Enable inter-system interoperability & collaboration
Interaction	Between application logic and the core language model	Between distinct, autonomous agent applications
Analogy	The specific grammar & format needed to talk to *one* expert	A universal translator & diplomatic protocol between nations
Example	PAAS structuring data for its Claude model's API call	PAAS sending a standardized booking request to RRAS

How A2A Works: A Simplified Interaction Flow

Discovery (Finding an Agent):

- The Client Agent finds the Agent Card of a Remote Agent (e.g., from a registry, or a known URL).

Capability Assessment (Can it do the job?):

- The Client Agent inspects the Agent Card to understand the Remote Agent's capabilities and authentication requirements.

Task Request (Sending the work):

- The Client Agent constructs a JSON-RPC request (the "task") according to A2A specifications.
- It sends this request via HTTP POST to the Remote Agent's designated A2A endpoint (e.g., /tasks/send).
- Request includes: task ID, method (capability), parameters, and any necessary input data.

A2A Core Concepts

1. Agent Card (agent.json): The Agent's Digital Passport

- A JSON file that an agent publishes to describe itself. Contains:
  - Identity: Unique ID, name, description.
  - Capabilities: List of tasks/skills the agent can perform (e.g., "summarize_text", "translate_document").
  - Endpoints: URLs for key A2A operations (e.g., /tasks/send to submit a task).
  - Authentication: Specifies required security schemes (e.g., OAuth 2.0 Bearer Token, API Key).

2. Task Lifecycle Management:

- Tasks progress through well-defined states: submitted -> working -> (optional input_required if more info needed) -> completed / failed / cancelled. This allows for tracking and robust error handling.

3. Standardized Message Structures:

- Primarily uses **JSON-RPC 2.0** for request and response payloads, ensuring clarity and consistency.
- Supports various content types within messages for multimodal interactions (text, files initially; extensible to audio/video).

4. Client & Remote Agent Roles:

- **Client Agent:** The agent that *initiates* a task request to another agent.
- **Remote Agent (or Server Agent):** The agent that *receives* and *processes* the task request. (Roles can be dynamic).

How A2A Works:
A Simplified Interaction Flow

Authentication & Authorization (Security check):

- The Remote Agent verifies the Client Agent's credentials (e.g., checks the Authorization header).

Task Processing (Doing the work):

- If authenticated, the Remote Agent accepts the task and begins processing it using its internal logic and tools.

Response & Updates (Reporting back):

- Short tasks: Remote Agent sends a JSON-RPC response with the task status (completed or failed) and any results (artifacts).
- Long-running tasks: Remote Agent might immediately acknowledge the task, then stream status updates (working, progress indicators) via Server-Sent Events (SSE), finally sending the completion status.

A2A Interaction Quick Demo / Walkthrough

"First, let's look at an Agent Card." (Show a sample agent.json file, highlighting id, capabilities, endpoints, authentication). "This tells other agents what 'MyEchoAgent' can do and how to talk to it."

"Now, as a Client Agent, I want to send a task. I'll use curl to send a JSON payload to its /tasks/send endpoint."

(Show curl command: curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer mytoken" -d '{"jsonrpc":"2.0","id":"task001","method":"echo_message","params":{"text":"Hello A2A!"}}' http://my-echo-agent.example.com/tasks/send)

"And here's the kind of JSON response we'd get back from MyEchoAgent, confirming the task is complete and echoing our message." (Show sample JSON response: {"jsonrpc":"2.0","id":"task001","result":{"status":"completed","artifacts":[{"text":"Agent responded: Hello A2A!"}]}})

ADK + A2A = The Best of Both Worlds

Build Powerful Agents with ADK, Connect Them with A2A:

- Use ADK's strengths to develop sophisticated individual AI agents, define their complex internal logic, integrate tools deeply, and orchestrate multi-step workflows *within your own system*.
- Then, leverage A2A as the "interface to the outside world" for your ADK agents or systems:
  - **Expose Services:** An ADK-built agent (or a system of ADK agents) can publish an Agent Card and expose A2A-compliant endpoints, allowing *other* A2A-compatible agents (from any source) to consume its services.
  - **Consume Services:** Your ADK agents can act as A2A clients to discover and delegate tasks to other A2A-enabled agents, whether they are also ADK-based, built with different frameworks, or provided by third parties.

ADK + A2A = The Best of Both Worlds

Enables Modular, Scalable, and Interoperable Systems:

- This combination allows you to build highly specialized agents (using ADK's fine-grained control) and then connect them into larger, distributed applications using A2A's standardized communication, promoting a more flexible, decoupled, and future-proof architecture.

Questions?