Copyright 2025 Google LLC.

In [ ]:
```
# @title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

🤝 Agent2Agent (A2A) Communication with ADK

Welcome to the Kaggle 5-day Agents course!

This notebook teaches you how to build **multi-agent systems** where different agents can communicate and collaborate using the **Agent2Agent (A2A) Protocol**. You'll learn how to integrate with external agent services and consume remote agents as if they were local.

In this notebook, you'll:

- Understand the A2A protocol and when to use it vs sub-agents
- Learn common A2A architecture patterns (cross-framework, cross-language, cross-organization)
- Expose an ADK agent via A2A using `to_a2a()`
- Consume remote agents using `RemoteA2aAgent`
- Build a product catalog integration system

‼️ Please Read

> ❌ ℹ️ **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.
>
> ⏸️ **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads **"Waiting for the next available notebook"**. The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.
>
> ❌ **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. See FAQ on 429 errors for more information. (https://www.kaggle.com/code/kaggle5daysofai/day-0-troubleshooting-and-faqs)

For help: Ask questions on the Kaggle Discord (https://discord.com/invite/kaggle) server.

📚 Overview of Agent2Agent (A2A)

🧐 The Problem

As you build more complex AI systems, you'll find that:

- **A single agent can't do everything** - Specialized agents for different domains work better
- **You need agents to collaborate** - Customer support needs product data, order systems need inventory info
- **Different teams build different agents** - You want to integrate agents from external vendors
- **Agents may use different languages/frameworks** - You need a standard communication protocol

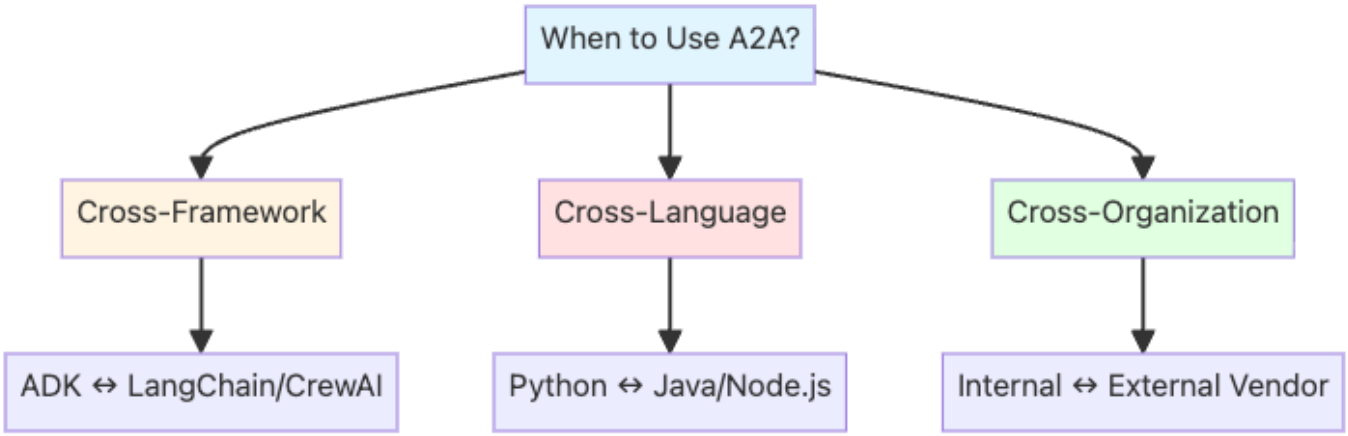
✅ The Solution: A2A Protocol

The Agent2Agent (A2A) Protocol (https://a2a-protocol.org/) is a **standard** that allows agents to:

- ✨ **Communicate over networks** - Agents can be on different machines
- ✨ **Use each other's capabilities** - One agent can call another agent like a tool
- ✨ **Work across frameworks** - Language/framework agnostic
- ✨ **Maintain formal contracts** - Agent cards describe capabilities

🏗️ Common A2A Architecture Patterns

The A2A protocol is particularly useful in three scenarios:



1. **Cross-Framework Integration**: ADK agent communicating with other agent frameworks
2. **Cross-Language Communication**: Python agent calling Java or Node.js agent
3. **Cross-Organization Boundaries**: Your internal agent integrating with external vendor services

📋 What This Notebook Demonstrates

We'll build a practical e-commerce integration:

1. **Product Catalog Agent** (exposed via A2A) - External vendor service that provides product information
2. **Customer Support Agent** (consumer) - Your internal agent that helps customers by querying product data

```
┌─────────────────────────┐          ┌─────────────────────────┐
│ Customer Support        │  ─A2A──▶ │ Product Catalog         │
│ Agent (Consumer)        │          │ Agent (Vendor)          │
│ Your Company            │          │ External Service        │
│ (localhost:8000)        │          │ (localhost:8001)        │
└─────────────────────────┘          └─────────────────────────┘
```

Why this justifies A2A:

- Product Catalog is maintained by an external vendor (you can't modify their code)
- Different organizations with separate systems
- Formal contract needed between services
- Product Catalog could be in a different language/framework

💡 A2A vs Local Sub-Agents: Decision Table

Factor	Use A2A	Use Local Sub-Agents
Agent Location	External service, different codebase	Same codebase, internal
Ownership	Different team/organization	Your team
Network	Agents on different machines	Same process/machine
Performance	Network latency acceptable	Need low latency
Language/Framework	Cross-language/framework needed	Same language
Contract	Formal API contract required	Internal interface
Example	External vendor product catalog	Internal order processing steps

📝 Tutorial Context

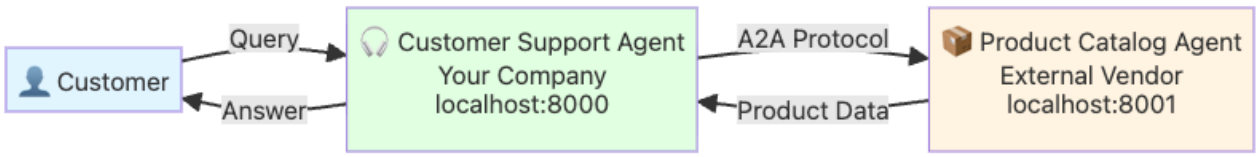
In this tutorial, we'll simulate both agents locally (both running on localhost) for learning purposes. In production:

- Product Catalog Agent would run on vendor's infrastructure (e.g., `https://vendor.com`)
- Customer Support Agent would run on your infrastructure
- They'd communicate over the internet using A2A protocol

This local simulation lets you learn A2A without needing to deploy services!

🔄 What We'll Build

Here's a high-level view of the system architecture you'll create in this tutorial:



How it works:

1. **Customer** asks a product question to your Customer Support Agent
2. **Support Agent** realizes it needs product information
3. **Support Agent** calls the **Product Catalog Agent** via A2A protocol
4. **Product Catalog Agent** (external vendor) returns product data
5. **Support Agent** formulates an answer and responds to the customer

---

🗺️ Tutorial Steps

In this tutorial, you'll complete **6 steps**:

1. **Create the Product Catalog Agent** - Build the vendor's agent with product lookup

2. **Expose via A2A** - Make it accessible using `to_a2a()`
3. **Start the Server** - Run the agent as a background service
4. **Create the Customer Support Agent** - Build the consumer agent
5. **Test Communication** - See A2A in action with real queries
6. **Understand the Flow** - Learn what happened behind the scenes

Let's get started! 🚀

📖 Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks in the documentation (https://www.kaggle.com/docs/notebooks).

Here's how to get started:

1. Verify Your Account (Required)

To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your Kaggle settings (https://www.kaggle.com/settings).

2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy and Edit` button in the top-right corner.

This creates a private copy of the notebook just for you.

3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶️ Run button next to any code cell to execute it.

Run the cells in order from top to bottom.

4. If You Get Stuck

To restart: Select `Factory reset` from the `Run` menu.

For help: Ask questions on the Kaggle Discord (https://discord.com/invite/kaggle) server.

⚙️ Setup

Before we go into today's concepts, follow the steps below to set up the environment.

Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the google-adk (https://google.github.io/adk-docs/) library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK, including A2A and its dependencies, in your own Python development environment outside of this course, you can do so by running:

```
pip install -q google-adk[a2a]
```

Configure your Gemini API Key

This notebook uses the Gemini API (https://ai.google.dev/gemini-api/), which requires an API key.

1. Get your API key

If you don't have one already, create an API key in Google AI Studio (https://aistudio.google.com/app/api-keys).

2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select `Add-ons` then `Secrets`.
2. Create a new secret with the label `GOOGLE_API_KEY`.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to `GOOGLE_API_KEY` is selected so that the secret is attached to the notebook.

3. Authenticate in the notebook

Run the cell below to access the `GOOGLE_API_KEY` you just saved and set it as an environment variable for the notebook to use:

In [ ]:
```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Setup and authentication complete.")
except Exception as e:
    print(
        f"🔑 Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

In [ ]:
```
import json
import requests
import subprocess
import time
import uuid

from google.adk.agents import LlmAgent
from google.adk.agents.remote_a2a_agent import (
    RemoteA2aAgent,
    AGENT_CARD_WELL_KNOWN_PATH,
)

from google.adk.a2a.utils.agent_to_a2a import to_a2a
from google.adk.models.google_llm import Gemini
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types

# Hide additional warnings in the notebook
import warnings

warnings.filterwarnings("ignore")

print("✅ ADK components imported successfully.")
```

Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

In [ ]:
```
retry_config = types.HttpRetryOptions(
    attempts=5,  # Maximum retry attempts
    exp_base=7,  # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],  # Retry on these HTTP errors
)
```

📦 Section 1: Create the Product Catalog Agent (To Be Exposed)

We'll create a **Product Catalog Agent** that provides product information from an external vendor's catalog. This agent will be **exposed via A2A** so other agents (like customer support) can use it.

Why expose this agent?

- In a real system, this would be maintained by an **external vendor** or third-party provider
- Your internal agents (customer support, sales, inventory) need product data
- The vendor **controls their own codebase** - you can't modify their implementation
- By exposing it via A2A, any authorized agent can consume it using the standard protocol

In [ ]:

```python
# Define a product catalog lookup tool
# In a real system, this would query the vendor's product database
def get_product_info(product_name: str) -> str:
    """Get product information for a given product.

    Args:
        product_name: Name of the product (e.g., "iPhone 15 Pro", "MacBook Pr
o")

    Returns:
        Product information as a string
    """
    # Mock product catalog - in production, this would query a real database
    product_catalog = {
        "iphone 15 pro": "iPhone 15 Pro, $999, Low Stock (8 units), 128GB, Tit
anium finish",
        "samsung galaxy s24": "Samsung Galaxy S24, $799, In Stock (31 units),
256GB, Phantom Black",
        "dell xps 15": 'Dell XPS 15, $1,299, In Stock (45 units), 15.6" displa
y, 16GB RAM, 512GB SSD',
        "macbook pro 14": 'MacBook Pro 14", $1,999, In Stock (22 units), M3 Pr
o chip, 18GB RAM, 512GB SSD',
        "sony wh-1000xm5": "Sony WH-1000XM5 Headphones, $399, In Stock (67 uni
ts), Noise-canceling, 30hr battery",
        "ipad air": 'iPad Air, $599, In Stock (28 units), 10.9" display, 64G
B',
        "lg ultrawide 34": 'LG UltraWide 34" Monitor, $499, Out of Stock, Expe
cted: Next week',
    }

    product_lower = product_name.lower().strip()

    if product_lower in product_catalog:
        return f"Product: {product_catalog[product_lower]}"
    else:
        available = ", ".join([p.title() for p in product_catalog.keys()])
        return f"Sorry, I don't have information for {product_name}. Available
products: {available}"


# Create the Product Catalog Agent
# This agent specializes in providing product information from the vendor's ca
talog
product_catalog_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="product_catalog_agent",
    description="External vendor's product catalog agent that provides product
information and availability.",
    instruction="""
    You are a product catalog specialist from an external vendor.
    When asked about products, use the get_product_info tool to fetch data fro
m the catalog.
    Provide clear, accurate product information including price, availability,
and specs.
    If asked about multiple products, look up each one.
    Be professional and helpful.
```

```
    """,
    tools=[get_product_info],  # Register the product lookup tool
)

print("✅ Product Catalog Agent created successfully!")
print("   Model: gemini-2.5-flash-lite")
print("   Tool: get_product_info()")
print("   Ready to be exposed via A2A...")
```

🌐 Section 2: Expose the Product Catalog Agent via A2A

Now we'll use ADK's `to_a2a()` function to make our Product Catalog Agent accessible to other agents.

What `to_a2a()` does:

- 🔧 Wraps your agent in an A2A-compatible server (FastAPI/Starlette)
- 📋 Auto-generates an **agent card** that includes:
  - Agent name, description, and version
  - Skills (your tools/functions become "skills" in A2A)
  - Protocol version and endpoints
  - Input/output modes
- 🌐 Serves the agent card at `/.well-known/agent-card.json` (standard A2A path)
- ✨ Handles all A2A protocol details (request/response formatting, task endpoints)

This is the **easiest way** to expose an ADK agent via A2A!

💡 **Key Concept: Agent Cards**

An **agent card** is a JSON document that serves as a "business card" for your agent. It describes:

- What the agent does (name, description, version)
- What capabilities it has (skills, tools, functions)
- How to communicate with it (URL, protocol version, endpoints)

Every A2A agent must publish its agent card at the standard path: `/.well-known/agent-card.json`

Think of it as the "contract" that tells other agents how to work with your agent.

📖 **Learn more:**

- Exposing Agents with ADK (https://google.github.io/adk-docs/a2a/quickstart-exposing/)
- A2A Protocol Specification (https://a2a-protocol.org/latest/specification/)

In [ ]:
```python
# Convert the product catalog agent to an A2A-compatible application
# This creates a FastAPI/Starlette app that:
#   1. Serves the agent at the A2A protocol endpoints
#   2. Provides an auto-generated agent card
#   3. Handles A2A communication protocol
product_catalog_a2a_app = to_a2a(
    product_catalog_agent, port=8001  # Port where this agent will be served
)

print("✅ Product Catalog Agent is now A2A-compatible!")
print("   Agent will be served at: http://localhost:8001")
print("   Agent card will be at: http://localhost:8001/.well-known/agent-card.
json")
print("   Ready to start the server...")
```

🚀 Section 3: Start the Product Catalog Agent Server

We'll start the Product Catalog Agent server in the **background** using `uvicorn`, so it can serve requests from other agents.

Why run in background?

- The server needs to keep running while we create and test the Customer Support Agent
- This simulates a real-world scenario where different agents run as separate services
- In production, the vendor would host this on their infrastructure

In [ ]:

```
# First, let's save the product catalog agent to a file that uvicorn can impor
t
product_catalog_agent_code = '''
import os
from google.adk.agents import LlmAgent
from google.adk.a2a.utils.agent_to_a2a import to_a2a
from google.adk.models.google_llm import Gemini
from google.genai import types

retry_config = types.HttpRetryOptions(
    attempts=5,  # Maximum retry attempts
    exp_base=7,  # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504],  # Retry on these HTTP errors
)

def get_product_info(product_name: str) -> str:
    """Get product information for a given product."""
    product_catalog = {
        "iphone 15 pro": "iPhone 15 Pro, $999, Low Stock (8 units), 128GB, Tit
anium finish",
        "samsung galaxy s24": "Samsung Galaxy S24, $799, In Stock (31 units),
256GB, Phantom Black",
        "dell xps 15": "Dell XPS 15, $1,299, In Stock (45 units), 15.6\\" disp
lay, 16GB RAM, 512GB SSD",
        "macbook pro 14": "MacBook Pro 14\\", $1,999, In Stock (22 units), M3
Pro chip, 18GB RAM, 512GB SSD",
        "sony wh-1000xm5": "Sony WH-1000XM5 Headphones, $399, In Stock (67 uni
ts), Noise-canceling, 30hr battery",
        "ipad air": "iPad Air, $599, In Stock (28 units), 10.9\\" display, 64G
B",
        "lg ultrawide 34": "LG UltraWide 34\\" Monitor, $499, Out of Stock, Ex
pected: Next week",
    }

    product_lower = product_name.lower().strip()

    if product_lower in product_catalog:
        return f"Product: {product_catalog[product_lower]}"
    else:
        available = ", ".join([p.title() for p in product_catalog.keys()])
        return f"Sorry, I don't have information for {product_name}. Available
products: {available}"

product_catalog_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="product_catalog_agent",
    description="External vendor's product catalog agent that provides product
information and availability.",
    instruction="""
    You are a product catalog specialist from an external vendor.
    When asked about products, use the get_product_info tool to fetch data fro
m the catalog.
    Provide clear, accurate product information including price, availability,
and specs.
    If asked about multiple products, look up each one.
```

```
    Be professional and helpful.
    """,
    tools=[get_product_info]
)

# Create the A2A app
app = to_a2a(product_catalog_agent, port=8001)
'''

# Write the product catalog agent to a temporary file
with open("/tmp/product_catalog_server.py", "w") as f:
    f.write(product_catalog_agent_code)

print("📝 Product Catalog agent code saved to /tmp/product_catalog_server.p
y")

# Start uvicorn server in background
# Note: We redirect output to avoid cluttering the notebook
server_process = subprocess.Popen(
    [
        "uvicorn",
        "product_catalog_server:app",  # Module:app format
        "--host",
        "localhost",
        "--port",
        "8001",
    ],
    cwd="/tmp",  # Run from /tmp where the file is
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    env={**os.environ},  # Pass environment variables (including GOOGLE_API_KE
Y)
)

print("🚀 Starting Product Catalog Agent server...")
print("   Waiting for server to be ready...")

# Wait for server to start (poll until it responds)
max_attempts = 30
for attempt in range(max_attempts):
    try:
        response = requests.get(
            "http://localhost:8001/.well-known/agent-card.json", timeout=1
        )
        if response.status_code == 200:
            print(f"\n✅ Product Catalog Agent server is running!")
            print(f"   Server URL: http://localhost:8001")
            print(f"   Agent card: http://localhost:8001/.well-known/agent-car
d.json")
            break
    except requests.exceptions.RequestException:
        time.sleep(5)
        print(".", end="", flush=True)
else:
    print("\n⚠️  Server may not be ready yet. Check manually if needed.")
```

```
# Store the process so we can stop it later
globals()["product_catalog_server_process"] = server_process
```

🔍 View the Auto-Generated Agent Card

The `to_a2a()` function automatically created an **agent card** that describes the Product Catalog Agent's capabilities. Let's take a look!

In [ ]:
```
# Fetch the agent card from the running server
try:
    response = requests.get(
        "http://localhost:8001/.well-known/agent-card.json", timeout=5
    )

    if response.status_code == 200:
        agent_card = response.json()
        print("📋 Product Catalog Agent Card:")
        print(json.dumps(agent_card, indent=2))

        print("\n✨ Key Information:")
        print(f"   Name: {agent_card.get('name')}")
        print(f"   Description: {agent_card.get('description')}")
        print(f"   URL: {agent_card.get('url')}")
        print(f"   Skills: {len(agent_card.get('skills', []))} capabilities ex
posed")
    else:
        print(f"❌ Failed to fetch agent card: {response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"❌ Error fetching agent card: {e}")
    print("   Make sure the Product Catalog Agent server is running (previous
cell)")
```

🎧 Section 4: Create the Customer Support Agent (Consumer)

Now we'll create a **Customer Support Agent** that consumes the Product Catalog Agent using A2A.

How it works:

1. We use `RemoteA2aAgent` to create a **client-side proxy** for the Product Catalog Agent
2. The Customer Support Agent can use the Product Catalog Agent like any other tool
3. ADK handles all the A2A protocol communication behind the scenes

This demonstrates the power of A2A: **agents can collaborate as if they were local!**

How RemoteA2aAgent works:

- It's a **client-side proxy** that reads the remote agent's card
- Translates sub-agent calls into A2A protocol requests (HTTP POST to `/tasks`)
- Handles all the protocol details so you just use it like a regular sub-agent

📖 **Learn more:**

- Consuming Remote Agents with ADK (https://google.github.io/adk-docs/a2a/quickstart-consuming/)
- What is A2A? (https://a2a-protocol.org/latest/topics/what-is-a2a/)

In [ ]:
```
# Create a RemoteA2aAgent that connects to our Product Catalog Agent
# This acts as a client-side proxy - the Customer Support Agent can use it lik
e a local agent
remote_product_catalog_agent = RemoteA2aAgent(
    name="product_catalog_agent",
    description="Remote product catalog agent from external vendor that provid
es product information.",
    # Point to the agent card URL - this is where the A2A protocol metadata li
ves
    agent_card=f"http://localhost:8001{AGENT_CARD_WELL_KNOWN_PATH}",
)

print("✅ Remote Product Catalog Agent proxy created!")
print(f"   Connected to: http://localhost:8001")
print(f"   Agent card: http://localhost:8001{AGENT_CARD_WELL_KNOWN_PATH}")
print("   The Customer Support Agent can now use this like a local sub-agen
t!")
```

In [ ]:
```
# Now create the Customer Support Agent that uses the remote Product Catalog Agent
customer_support_agent = LlmAgent(
    model=Gemini(model="gemini-2.5-flash-lite", retry_options=retry_config),
    name="customer_support_agent",
    description="A customer support assistant that helps customers with product inquiries and information.",
    instruction="""
    You are a friendly and professional customer support agent.

    When customers ask about products:
    1. Use the product_catalog_agent sub-agent to look up product information
    2. Provide clear answers about pricing, availability, and specifications
    3. If a product is out of stock, mention the expected availability
    4. Be helpful and professional!

    Always get product information from the product_catalog_agent before answering customer questions.
    """,
    sub_agents=[remote_product_catalog_agent],  # Add the remote agent as a sub-agent!
)

print("✅ Customer Support Agent created!")
print("   Model: gemini-2.5-flash-lite")
print("   Sub-agents: 1 (remote Product Catalog Agent via A2A)")
print("   Ready to help customers!")
```

🧪 Section 5: Test A2A Communication

Let's test the agent-to-agent communication! We'll ask the Customer Support Agent about products, and it will communicate with the Product Catalog Agent via A2A.

What happens behind the scenes:

1. Customer asks Support Agent a question about a product
2. Support Agent realizes it needs product info
3. Support Agent calls the `remote_product_catalog_agent` (RemoteA2aAgent)
4. ADK sends an A2A protocol request to `http://localhost:8001`
5. Product Catalog Agent processes the request and responds
6. Support Agent receives the response and continues
7. Customer gets the final answer

All of this happens **transparently** - the Support Agent doesn't need to know it's talking to a remote agent!

In [ ]:

```
async def test_a2a_communication(user_query: str):
    """
    Test the A2A communication between Customer Support Agent and Product Cata
log Agent.

    This function:
    1. Creates a new session for this conversation
    2. Sends the query to the Customer Support Agent
    3. Support Agent communicates with Product Catalog Agent via A2A
    4. Displays the response

    Args:
        user_query: The question to ask the Customer Support Agent
    """
    # Setup session management (required by ADK)
    session_service = InMemorySessionService()

    # Session identifiers
    app_name = "support_app"
    user_id = "demo_user"
    # Use unique session ID for each test to avoid conflicts
    session_id = f"demo_session_{uuid.uuid4().hex[:8]}"

    # CRITICAL: Create session BEFORE running agent (synchronous, not async!)
    # This pattern matches the deployment notebook exactly
    session = await session_service.create_session(
        app_name=app_name, user_id=user_id, session_id=session_id
    )

    # Create runner for the Customer Support Agent
    # The runner manages the agent execution and session state
    runner = Runner(
        agent=customer_support_agent, app_name=app_name, session_service=sessi
on_service
    )

    # Create the user message
    # This follows the same pattern as the deployment notebook
    test_content = types.Content(parts=[types.Part(text=user_query)])

    # Display query
    print(f"\n👤 Customer: {user_query}")
    print(f"\n🎧 Support Agent response:")
    print("-" * 60)

    # Run the agent asynchronously (handles streaming responses and A2A commun
ication)
    async for event in runner.run_async(
        user_id=user_id, session_id=session_id, new_message=test_content
    ):
        # Print final response only (skip intermediate events)
        if event.is_final_response() and event.content:
            for part in event.content.parts:
                if hasattr(part, "text"):
                    print(part.text)
```

```
    print("-" * 60)


# Run the test
print("💉 Testing A2A Communication...\n")
await test_a2a_communication("Can you tell me about the iPhone 15 Pro? Is it i
n stock?")
```

Try More Examples

Let's test a few more scenarios to see A2A communication in action!

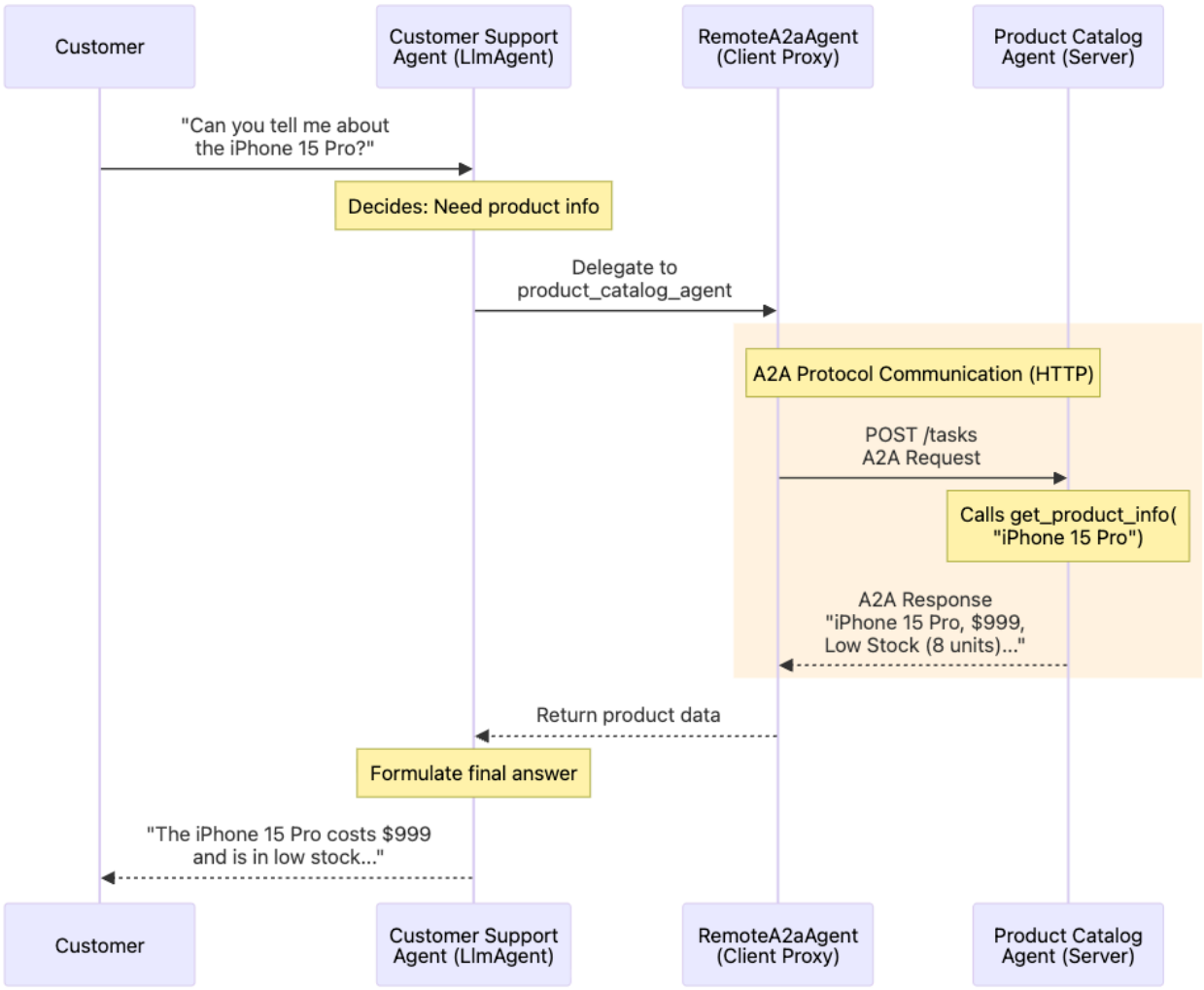
In [ ]:
```
# Test comparing multiple products
await test_a2a_communication(
    "I'm looking for a laptop. Can you compare the Dell XPS 15 and MacBook Pro
14 for me?"
)
```

In [ ]:
```
# Test specific product inquiry
await test_a2a_communication(
    "Do you have the Sony WH-1000XM5 headphones? What's the price?"
)
```

🔍 Section 6: Understanding What Just Happened

A2A Communication Flow

When you ran the tests above, here's the detailed step-by-step flow of how the agents communicated:



A2A Protocol Communication:

Behind the scenes, here's what happens at the protocol level:

- **RemoteA2aAgent** sends HTTP POST requests to the `/tasks` endpoint on `http://localhost:8001`
- Request and response data follow the A2A Protocol Specification (https://a2a-protocol.org/latest/specification/)
- Data is exchanged in standardized JSON format
- The protocol ensures any A2A-compatible agent (regardless of language/framework) can communicate

This standardization is what makes cross-organization, cross-language agent communication possible!

What happened:

1. **Customer** asked about the iPhone 15 Pro
2. **Customer Support Agent** (LlmAgent) received the question and decided it needs product information
3. **Support Agent** delegated to the `product_catalog_agent` sub-agent
4. **RemoteA2aAgent** (client-side proxy) translated this into an A2A protocol request
5. The A2A request was sent over HTTP to `http://localhost:8001` (highlighted in yellow)
6. **Product Catalog Agent** (server) received the request and called `get_product_info("iPhone 15 Pro")`
7. **Product Catalog Agent** returned the product information via A2A response
8. **RemoteA2aAgent** received the response and passed it back to the Support Agent
9. **Support Agent** formulated a final answer with the product details
10. **Customer** received the complete, helpful response

Key Benefits Demonstrated

1. **Transparency**: Support Agent doesn't "know" Product Catalog Agent is remote
2. **Standard Protocol**: Uses A2A standard - any A2A-compatible agent works
3. **Easy Integration**: Just one line: `sub_agents=[remote_product_catalog_agent]`
4. **Separation of Concerns**: Product data lives in Catalog Agent (vendor), support logic in Support Agent (your company)

Real-World Applications

This pattern enables:

- **Microservices**: Each agent is an independent service
- **Third-party Integration**: Consume agents from external vendors (e.g., product catalogs, payment processors)
- **Cross-language**: Product Catalog Agent could be Java, Support Agent Python
- **Specialized Teams**: Vendor maintains catalog, your team maintains support agent
- **Cross-Organization**: Vendor hosts catalog on their infrastructure, you integrate via A2A

🎓 Next Steps and Learning Resources

🚀 Enhancement Ideas

Now that you understand A2A basics, try extending this example:

1. **Add More Agents**:
    - Create an **Inventory Agent** that checks stock levels and restocking schedules
    - Create a **Shipping Agent** that provides delivery estimates and tracking
    - Have Customer Support Agent coordinate all three via A2A
2. **Real Data Sources**:
    - Replace mock product catalog with real database (PostgreSQL, MongoDB)
    - Add real inventory tracking system integration
    - Connect to real payment gateway APIs
3. **Advanced A2A Features**:
    - Implement authentication between agents (API keys, OAuth)
    - Add error handling and retries for network failures
    - Use the alternative `adk api_server --a2a` approach
4. **Deploy to Production**:
    - Deploy Product Catalog Agent to Agent Engine
    - Update agent card URL to point to production server (e.g., `https://vendor-catalog.example.com`)
    - Consumer agents can now access it over the internet!

📚 Documentation

A2A Protocol:

- Official A2A Protocol Website (https://a2a-protocol.org/)
- A2A Protocol Specification (https://a2a-protocol.org/latest/spec/)

ADK A2A Guides:

- Introduction to A2A in ADK (https://google.github.io/adk-docs/a2a/intro/)
- Exposing Agents Quickstart (https://google.github.io/adk-docs/a2a/quickstart-exposing/)
- Consuming Agents Quickstart (https://google.github.io/adk-docs/a2a/quickstart-consuming/)

Other Deployment Options:

- Deploy ADK Agents to Cloud Run (https://google.github.io/adk-docs/deploy/cloud-run/)
- Deploy to Agent Engine (https://google.github.io/adk-docs/deploy/agent-engine/)
- Deploy to GKE (https://google.github.io/adk-docs/deploy/gke/)

📊 Summary - A2A Communication Patterns

Key Takeaways

In this notebook, you learned how to build multi-agent systems with A2A:

- **A2A Protocol**: Standardized protocol for agent-to-agent communication across networks and frameworks
- **Exposing Agents**: Use `to_a2a()` to make your agents accessible to others with auto-generated agent cards
- **Consuming Agents**: Use `RemoteA2aAgent` to integrate remote agents as if they were local sub-agents
- **Use Cases**: Best for microservices architectures, cross-team integrations, and third-party agent consumption

✅ Congratulations! You're an A2A Expert

You've successfully learned how to build multi-agent systems using the A2A protocol!

You now know how to expose agents as services, consume remote agents, and build collaborative multi-agent systems that can scale across teams and organizations.

ℹ️ Note: No submission required!

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

📚 Learn More

Refer to the following documentation to learn more:

- ADK Documentation (https://google.github.io/adk-docs/)
- A2A Protocol Official Website (https://a2a-protocol.org/)
- A2A Tutorials (https://a2a-protocol.org/latest/tutorials/)
- Introduction to A2A in ADK (https://google.github.io/adk-docs/a2a/intro/)
- Exposing Agents Quickstart (https://google.github.io/adk-docs/a2a/quickstart-exposing/)
- Consuming Agents Quickstart (https://google.github.io/adk-docs/a2a/quickstart-consuming/)

🎯 Next Steps

Now that you understand A2A communication, you can build complex multi-agent systems where specialized agents collaborate to solve real-world problems. Consider deploying your agents to production using Cloud Run or Agent Engine to make them accessible over the internet!

Ready for more? Explore advanced ADK features like custom agents, streaming, and production deployment patterns!

Authors

Lavi Nigam (https://www.linkedin.com/in/lavinigam/)