A2A Tutorial & Specification: Complete Implementation Guide

Table of Contents

Introduction: Tutorial Overview

This guide takes you from zero to building your first A2A-compliant agent in Python. We'll start with a simple "Hello World" agent, then progress to more complex agents with LLM integration.

What You'll Learn

☑ **Basic Concepts:**

- How to set up your Python environment
- Creating Agent Cards and Skills
- Building an Agent Executor
- Running an A2A server

☑ **Advanced Topics:**

- Streaming responses with Server-Sent Events
- Multi-turn conversations
- LangGraph integration
- Push notifications

☑ **Technical Understanding:**

- Protocol specification details
- Data structures and types
- RPC methods and bindings

Environment Setup

Prerequisites

Before we start, you need:

- **Python 3.10 or higher**
- **Git** for cloning the repository
- **A code editor** (VS Code recommended)
- **Terminal/Command prompt** access

Step 1: Clone the A2A Samples Repository

```
# Clone the repository
git clone https://github.com/a2aproject/a2a-samples.git -b main --depth 1
cd a2a-samples
```

This repository contains:

- Sample agents (Python, Java, JavaScript)
- A2A Python SDK
- Example implementations
- Test clients

Step 2: Set Up Python Environment

Create a Virtual Environment

A virtual environment isolates your project dependencies.

On macOS/Linux:

```
python -m venv .venv
source .venv/bin/activate
```

On Windows:

```
python -m venv .venv
.venv\Scripts\activate
```

You'll see `(.venv)` in your command prompt when activated.

Install the A2A SDK and Dependencies

```
# Install all required packages
pip install -r samples/python/requirements.txt
```

This installs:

- `a2a` - The A2A Python SDK
- `starlette` - Web framework for the server
- `uvicorn` - ASGI server to run the application
- `httpx` - HTTP client for making requests
- Other dependencies (langchain, langgraph for advanced examples)

Step 3: Verify Installation

```
# Test that the A2A SDK is installed
python -c "import a2a; print('A2A SDK imported successfully')"
```

If you see "A2A SDK imported successfully", you're ready to go! 🎉

Understanding Agent Skills and Agent Cards

Before building an agent, you need to define **what it can do** (Skills) and **how to find it** (Agent Card).

What is an Agent Skill?

An **Agent Skill** describes a specific capability of your agent. Think of it like a feature on a menu.

Key Components:

```
from a2a.types import AgentSkill

skill = AgentSkill(
    id='hello_world',              # Unique identifier
    name='Returns hello world',    # Human-readable name
    description='just returns hello world',  # Detailed explanation
    tags=['hello world'],          # Keywords for search
    examples=['hi', 'hello world'] # Example prompts
)
```

Purpose: Tells clients (other agents or users) what your agent can do.

Real-World Example:

```
# A weather agent skill
weather_skill = AgentSkill(
    id='weather-lookup',
    name='Weather Information',
    description='Provides current weather and forecasts for any location',
    tags=['weather', 'forecast', 'temperature'],
    examples=[
        'What is the weather in Paris?',
        'Will it rain tomorrow in Tokyo?'
    ]
)
```

What is an Agent Card?

An **Agent Card** is like a digital business card for your agent. It tells others:

- Who you are (identity)
- Where to reach you (URL)
- What you can do (skills)
- How to authenticate (security)
- What features you support (capabilities)

Basic Structure:

```python
from a2a.types import AgentCard, AgentCapabilities

agent_card = AgentCard(
    name='Hello World Agent',
    description='Just a hello world agent',
    url='http://localhost:9999/',
    version='1.0.0',
    default_input_modes=['text'],
    default_output_modes=['text'],
    capabilities=AgentCapabilities(streaming=True),
    skills=[skill],  # The skills we defined above
    supports_authenticated_extended_card=True
)
```

Let's Break Down Each Field:

Field	What It Means	Example
`name`	Your agent's name	"Hello World Agent"
`description`	What your agent does	"A simple greeting agent"
`url`	Where your agent lives	"http://localhost:9999/"
`version`	Your agent's version	"1.0.0"
`default_input_modes`	What input it accepts	`['text']` means text only
`default_output_modes`	What output it produces	`['text']` means text only
`capabilities`	What features it supports	Streaming, push notifications, etc.
`skills`	List of things it can do	Array of AgentSkill objects

Capabilities Explained:

```
capabilities = AgentCapabilities(
    streaming=True,            # Can send real-time updates
    pushNotifications=False,   # Can send webhook notifications
    extensions=[]              # Custom protocol extensions
)
```

Public vs. Extended Agent Cards

Some agents have **two versions** of their card:

1. **Public Agent Card** - Basic info, available to everyone
2. **Extended Agent Card** - Detailed info, only for authenticated users

Example:

```
# Public card (anyone can see)
public_agent_card = AgentCard(
    name='Hello World Agent',
    skills=[basic_skill],
    supports_authenticated_extended_card=True  # Indicates extended card exists
)

# Extended card (authenticated users only)
extended_agent_card = public_agent_card.model_copy(
    update={
        'name': 'Hello World Agent - Extended Edition',
        'skills': [basic_skill, premium_skill]  # More skills!
    }
)
```

Why Use Extended Cards?

- Hide premium features from public view
- Provide more details to trusted partners
- Implement tiered access levels

Building Your First Agent: The Agent Executor

The **Agent Executor** is the brain of your agent. It contains the logic that processes requests and generates responses.

Understanding the Architecture

```
┌─────────────────┐
│     Client      │ Sends message
```

```
└────────┬─────────┘
         │
         ▼
┌──────────────────┐
│    A2A Server    │  Receives request
└────────┬─────────┘
         │
         ▼
┌──────────────────┐
│ Request Handler  │  Routes to executor
└────────┬─────────┘
         │
         ▼
┌──────────────────┐
│ Agent Executor   │  ← YOUR CODE GOES HERE
└────────┬─────────┘
         │
         ▼
┌──────────────────┐
│    Your Agent    │  Actual business logic
└──────────────────┘
```

The AgentExecutor Interface

The A2A SDK provides an abstract class you implement:

```python
from a2a.server.agent_execution import AgentExecutor
from a2a.server.request_handlers import RequestContext, EventQueue

class MyAgentExecutor(AgentExecutor):

    async def execute(
        self,
        context: RequestContext,    # Info about the request
        event_queue: EventQueue     # Where to send responses
    ) -> None:
        # YOUR LOGIC HERE
        pass

    async def cancel(
        self,
        context: RequestContext,
        event_queue: EventQueue
    ) -> None:
        # Handle cancellation requests
        pass
```

Key Components:

- `context`: Contains the incoming message, task info, user identity

- `event_queue`: Where you send back messages, tasks, or events
- `execute()`: Handles message/send and message/stream requests
- `cancel()`: Handles task cancellation requests

Hello World Agent Executor: Step-by-Step

Let's build a simple agent that says "Hello World":

Step 1: Create Your Agent Logic

```python
# agent.py
class HelloWorldAgent:
    """The actual agent logic - this is your business logic"""

    async def invoke(self) -> str:
        """Returns a greeting"""
        return 'Hello World'
```

This is the simplest possible agent - it just returns a string.

Step 2: Create the Executor

```python
# agent_executor.py
from a2a.server.agent_execution import AgentExecutor
from a2a.server.request_handlers import RequestContext, EventQueue
from a2a.utils import new_agent_text_message

class HelloWorldAgentExecutor(AgentExecutor):
    """Bridges A2A protocol with your agent logic"""

    def __init__(self):
        self.agent = HelloWorldAgent()

    async def execute(
        self,
        context: RequestContext,
        event_queue: EventQueue,
    ) -> None:
        # Step 1: Get the result from your agent
        result = await self.agent.invoke()

        # Step 2: Convert to A2A Message format
        message = new_agent_text_message(result)

        # Step 3: Send it back to the client
        await event_queue.enqueue_event(message)

    async def cancel(
        self,
        context: RequestContext,
        event_queue: EventQueue
    ) -> None:
        # This simple agent doesn't support cancellation
        raise Exception('cancel not supported')
```

What's Happening Here?

1. `__init__`: Creates an instance of your agent
2. `execute`:
   - Calls your agent's `invoke()` method
   - Wraps the result in an A2A `Message` object
   - Puts it on the event queue to send to client
3. `cancel`: Raises an exception (we don't support cancellation)

Understanding the Event Queue

The **Event Queue** is how you communicate results back to the client.

What Can You Send?

```
# Option 1: Send a simple message
await event_queue.enqueue_event(message)

# Option 2: Send a task with status
await event_queue.enqueue_event(task)

# Option 3: Send status updates (for streaming)
await event_queue.enqueue_event(TaskStatusUpdateEvent(...))

# Option 4: Send artifact updates (for streaming)
await event_queue.enqueue_event(TaskArtifactUpdateEvent(...))
```

Message vs. Task:

- **Message**: For simple, immediate responses (like our Hello World)
- **Task**: For long-running operations that need tracking

Understanding RequestContext

The **RequestContext** gives you information about the incoming request:

```
async def execute(self, context: RequestContext, event_queue: EventQueue):
    # Access the incoming message
    user_message = context.message  # The Message object from client
```

```python
    # Access task information (if continuing an existing task)
    task_id = context.task_id
    context_id = context.context_id

    # Access the full request
    request_params = context.params
```

Example: Reading User Input

```python
async def execute(self, context: RequestContext, event_queue: EventQueue):
    # Get the user's message
    user_message = context.message

    # Extract text from message parts
    for part in user_message.parts:
        if part.text:
            user_text = part.text
            print(f"User said: {user_text}")
```

Starting the A2A Server

Now that we have an Agent Card and Agent Executor, let's start the server!

Complete Server Setup

```python
# __main__.py
import uvicorn
from a2a.server.apps import A2AStarletteApplication
from a2a.server.request_handlers import DefaultRequestHandler
from a2a.server.tasks import InMemoryTaskStore
from a2a.types import AgentCapabilities, AgentCard, AgentSkill
from agent_executor import HelloWorldAgentExecutor

if __name__ == '__main__':
    # Step 1: Define the skill
    skill = AgentSkill(
        id='hello_world',
        name='Returns hello world',
        description='just returns hello world',
        tags=['hello world'],
        examples=['hi', 'hello world'],
    )

    # Step 2: Create the Agent Card
    public_agent_card = AgentCard(
        name='Hello World Agent',
        description='Just a hello world agent',
        url='http://localhost:9999/',
        version='1.0.0',
        default_input_modes=['text'],
        default_output_modes=['text'],
        capabilities=AgentCapabilities(streaming=True),
        skills=[skill],
        supports_authenticated_extended_card=True,
    )

    # Step 3: Create the request handler
    request_handler = DefaultRequestHandler(
        agent_executor=HelloWorldAgentExecutor(),
        task_store=InMemoryTaskStore(),
    )

    # Step 4: Create the A2A server application
    server = A2AStarletteApplication(
        agent_card=public_agent_card,
        http_handler=request_handler,
    )

    # Step 5: Run the server
    uvicorn.run(server.build(), host='0.0.0.0', port=9999)
```

Understanding Each Component

DefaultRequestHandler

```
request_handler = DefaultRequestHandler(
    agent_executor=HelloWorldAgentExecutor(),
    task_store=InMemoryTaskStore(),
)
```

What It Does:

- Routes incoming A2A RPC calls to your executor
- Manages task lifecycle (create, update, complete)
- Stores task state in the TaskStore
- Handles protocol-level details (you focus on logic)

InMemoryTaskStore

```
task_store = InMemoryTaskStore()
```

What It Does:

- Stores task information in memory (not persistent)
- Tracks task state across requests

- Enables multi-turn interactions
- For production, use a database-backed store

A2AStarletteApplication

```
server = A2AStarletteApplication(
    agent_card=public_agent_card,
    http_handler=request_handler,
)
```

What It Does:

- Creates a Starlette web application
- Exposes Agent Card at `/.well-known/agent-card.json`
- Routes HTTP requests to your handler
- Handles protocol bindings (JSON-RPC, REST)

uvicorn.run()

```
uvicorn.run(server.build(), host='0.0.0.0', port=9999)
```

What It Does:

- Starts an ASGI server
- `host='0.0.0.0'` - Listen on all network interfaces
- `port=9999` - The port number (matches Agent Card URL)

Running Your Server

```
# Navigate to the helloworld directory
cd samples/python/agents/helloworld

# Run the server
python __main__.py
```

You should see:

```
INFO:     Started server process [12345]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:9999 (Press CTRL+C to quit)
```

Your agent is now live! 🚀

Interacting with Your Agent

Now let's send requests to your agent and see it respond.

Fetching the Agent Card

First, let's verify the agent is running by fetching its Agent Card:

```
# Using curl
curl http://localhost:9999/.well-known/agent-card.json
```

You should see: The full Agent Card JSON with all the info we defined.

Creating a Client

The A2A SDK includes a client for making requests:

```
# test_client.py
import httpx
import asyncio
from uuid import uuid4
from a2a.client import A2AClient, A2ACardResolver
from a2a.types import SendMessageRequest, MessageSendParams

async def main():
    base_url = 'http://localhost:9999'

    async with httpx.AsyncClient() as httpx_client:
        # Step 1: Fetch the Agent Card
        resolver = A2ACardResolver(
            httpx_client=httpx_client,
            base_url=base_url,
        )
        agent_card = await resolver.resolve_agent_card()

        # Step 2: Create the A2A Client
        client = A2AClient(
            httpx_client=httpx_client,
            agent_card=agent_card
        )

        # Step 3: Send a message
        message_data = {
            'message': {
                'role': 'user',
                'parts': [
                    {'kind': 'text', 'text': 'Hello!'}
                ],
                'messageId': uuid4().hex,
            },
        }

        request = SendMessageRequest(
            id=str(uuid4()),
            params=MessageSendParams(**message_data)
        )

        # Step 4: Get the response
        response = await client.send_message(request)
        print(response.model_dump(mode='json', exclude_none=True))

# Run the client
asyncio.run(main())
```

Understanding the Client Code

Step 1: A2ACardResolver

```
resolver = A2ACardResolver(
    httpx_client=httpx_client,
    base_url=base_url,
)
agent_card = await resolver.resolve_agent_card()
```

What It Does:

- Fetches the Agent Card from `/.well-known/agent-card.json`
- Parses it into an `AgentCard` object
- Validates the structure

Step 2: A2AClient

```
client = A2AClient(
    httpx_client=httpx_client,
    agent_card=agent_card
)
```

What It Does:

- Initializes a client with the agent's information
- Knows where to send requests (from Agent Card URL)
- Handles request/response serialization

Step 3: Send Message

```
message_data = {
    'message': {
        'role': 'user',              # Who's sending
        'parts': [                   # Content
            {'kind': 'text', 'text': 'Hello!'}
        ],
        'messageId': uuid4().hex,    # Unique ID
    },
}
```

Message Structure:

- **role**: `'user'` (from client) or `'agent'` (from server)
- **parts**: Array of content pieces (text, files, data)
- **messageId**: Unique identifier for this message

Step 4: Get Response

```
response = await client.send_message(request)
```

Response Type: Either a `Message` or a `Task` object

For Hello World, you'll get:

```
{
  "jsonrpc": "2.0",
  "id": "some-uuid",
  "result": {
    "type": "message",
    "role": "agent",
    "parts": [
      {
        "type": "text",
        "text": "Hello World"
      }
    ],
    "messageId": "another-uuid"
  }
}
```

Streaming Responses

For real-time updates, use streaming:

```python
async def streaming_example():
    # ... (setup code same as above) ...

    # Send a streaming request
    streaming_request = SendStreamingMessageRequest(
        id=str(uuid4()),
        params=MessageSendParams(**message_data)
    )

    # Get the stream
    stream_response = client.send_message_streaming(streaming_request)

    # Process each chunk as it arrives
    async for chunk in stream_response:
        print(chunk.model_dump(mode='json', exclude_none=True))
```

Output:

```
{"jsonrpc":"2.0","id":"...","result":{"type":"message","role":"agent","parts":
[{"type":"text","text":"Hello World"}],"final":true}}
```

The `final: true` indicates this is the last message in the stream.

Advanced Features: Streaming and Multi-Turn

For real-world agents, you need more advanced features. Let's look at the **LangGraph example** which demonstrates:

- LLM integration
- Streaming events
- Multi-turn conversations
- Task state management

Setting Up LangGraph Example

Step 1: Get a Gemini API Key

Visit Google AI Studio to get an API key.

Step 2: Create .env File

```bash
# In samples/python/agents/langgraph/
echo "GOOGLE_API_KEY=your_key_here" > .env
```

Step 3: Run the Server

```bash
cd samples/python/agents/langgraph/app
python __main__.py
```

The server starts on `http://localhost:10000`

LangGraph Agent Architecture

```
┌─────────────────┐
│  User Message   │
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│ CurrencyAgent   │  ← Uses LangGraph + Gemini LLM
│ (agent.py)      │
└────────┬────────┘
         │
         ▼
┌───────────────────────────┐
│ CurrencyAgentExecutor     │  ← Implements AgentExecutor
│ (agent_executor.py)       │
└────────┬──────────────────┘
         │
         ├─ Enqueues TaskStatusUpdateEvent
         ├─ Enqueues TaskArtifactUpdateEvent
         └─ Tracks task state
```

Understanding the Currency Agent

```python
# agent.py (simplified)
from langchain_google_genai import ChatGoogleGenerativeAI
from langgraph.prebuilt import create_react_agent

class CurrencyAgent:
    def __init__(self):
        # Initialize the LLM
        self.llm = ChatGoogleGenerativeAI(
            model="gemini-2.0-flash-exp",
            temperature=0.7
        )

        # Define tools
        tools = [get_exchange_rate]  # Tool to get currency rates
        # Create the LangGraph agent
        self.agent = create_react_agent(self.llm, tools)

    async def invoke(self, user_message: str):
        # Run the agent with the user's message
        result = await self.agent.ainvoke({
            "messages": [HumanMessage(content=user_message)]
        })
        return result
```

Key Components:

1. **ChatGoogleGenerativeAI**: The Gemini LLM
2. **Tools**: Functions the agent can call (like `get_exchange_rate`)
3. **create_react_agent**: LangGraph's agent builder (ReAct = Reason + Act)

Streaming Task Updates

The CurrencyAgentExecutor sends different types of events:

```
# agent_executor.py (simplified)
async def execute(self, context: RequestContext, event_queue: EventQueue):
    # Send initial status update
    await event_queue.enqueue_event(TaskStatusUpdateEvent(
        taskId=task_id,
        contextId=context_id,
        status=TaskStatus(
            state=TaskState.working,
            message=Message(
                role='agent',
                parts=[TextPart(text="Looking up exchange rates...")]
            )
        ),
        final=False  # Not done yet
    ))

    # Process with LangGraph agent
    result = await self.agent.invoke(user_message)

    # Send artifact with result
    await event_queue.enqueue_event(TaskArtifactUpdateEvent(
        taskId=task_id,
        contextId=context_id,
        artifact=Artifact(
            artifactId=str(uuid4()),
            name="Exchange Rate Result",
            parts=[TextPart(text=result['answer'])]
        ),
        lastChunk=True  # Final artifact
    ))
    # Send final status update
    await event_queue.enqueue_event(TaskStatusUpdateEvent(
        taskId=task_id,
        contextId=context_id,
        status=TaskStatus(state=TaskState.completed),
        final=True  # Stream is done
    ))
```

Event Types:

1. **TaskStatusUpdateEvent**: Status changed (working → completed)
2. **TaskArtifactUpdateEvent**: New result/output available
3. **Task**: Initial task creation

Multi-Turn Conversations

The Currency Agent can ask for clarification:

Turn 1 - User: "How much is 100 USD?"

Agent Response:

```
{
  "task": {
    "id": "task-123",
    "status": {
      "state": "input-required",
      "message": {
        "role": "agent",
        "parts": [{"text": "To which currency would you like to convert?"}]
      }
    }
  }
}
```

Turn 2 - User: "in GBP" (includes taskId from previous response)

Agent Response:

```
{
  "task": {
    "id": "task-123",
    "status": {"state": "completed"},
    "artifacts": [{
      "name": "conversion-result",
      "parts": [{"text": "100 USD = 79.50 GBP"}]
    }]
    }
}
```

How It Works:

1. Client includes `taskId` in the second message
2. Agent retrieves task state from TaskStore
3. Agent continues processing with new information
4. Same task ID maintained across turns

Understanding the Technical Specification

Now let's dive into the technical details from the specification files.

Protocol Overview

A2A uses **three layers**:

```
┌───────────────────────────────────────┐
│  Layer 3: Protocol Bindings          │
│  (JSON-RPC, gRPC, HTTP/REST)          │
└───────────────────────────────────────┘
                  ▲
┌───────────────────────────────────────┐
│  Layer 2: Abstract Operations         │
│  (SendMessage, GetTask, etc.)         │
└───────────────────────────────────────┘
                  ▲
┌───────────────────────────────────────┐
│  Layer 1: Canonical Data Model        │
│  (Task, Message, AgentCard, etc.)     │
└───────────────────────────────────────┘
```

Layer 1 (Data Model): Defines what data looks like (Protocol Buffers) **Layer 2 (Operations):** Defines what you can do (Send, Get, Cancel) **Layer 3 (Bindings):** Defines how to do it over different protocols

---

Core Data Types

1. Task

A **Task** represents work being done by the agent.

Structure:

```
Task(
    id='task-uuid',             # Unique identifier
    context_id='ctx-uuid',      # Groups related tasks
    status=TaskStatus(...),   # Current state
    artifacts=[...],          # Outputs/results
    history=[...],            # Conversation history
    metadata={...}            # Custom data
)
```

Task States:

```
class TaskState:
    SUBMITTED       = 'submitted'      # Just received
    WORKING         = 'working'        # Being processed
    COMPLETED       = 'completed'      # Done successfully
    FAILED          = 'failed'         # Encountered error
    CANCELLED       = 'cancelled'      # User cancelled
    INPUT_REQUIRED  = 'input-required' # Needs more info
    AUTH_REQUIRED   = 'auth-required'  # Needs credentials
    REJECTED        = 'rejected'       # Agent won't do it
```

State Transitions:

```
submitted → working → completed
              │
              ├──→ input-required ──→ working
              │
              ├──→ auth-required ──→ working
              │
              ├──→ failed
              │
              └──→ cancelled
```

---

2. Message

A **Message** is a single communication turn.

Structure:

```
Message(
    message_id='msg-uuid',    # Unique ID
    context_id='ctx-uuid',    # Optional context link
    task_id='task-uuid',      # Optional task link
    role='user',              # 'user' or 'agent'
    parts=[...],              # Content
    metadata={...},           # Custom data
    extensions=[...],         # Extension URIs
    reference_task_ids=[...]  # Related tasks
)
```

---

3. Part

A **Part** is a piece of content within a message or artifact.

Types:

```
# Text content
Part(text="Hello, world!")

# File content (inline)
Part(file=FilePart(
    name="document.pdf",
    media_type="application/pdf",
    file_with_bytes=base64_bytes
))

# File content (URL)
Part(file=FilePart(
    name="image.png",
    media_type="image/png",
    file_with_uri="https://example.com/image.png"
))

# Structured data
Part(data=DataPart(
    data={"currency": "USD", "amount": 100}
))
```

---

4. Artifact

An **Artifact** is a concrete output from the agent.

Structure:

```
Artifact(
    artifact_id='art-uuid',
    name='Currency Conversion Result',
    description='Exchange rate calculation',
    parts=[
        Part(text="100 USD = 79.50 GBP")
    ],
    metadata={...},
    extensions=[...]
)
```

Artifacts vs. Messages:

- **Message**: Conversational turn (questions, status updates)
- **Artifact**: Tangible deliverable (documents, images, data)

Core Operations

1. SendMessage

Send a message to the agent.

Request:

```json
{
  "jsonrpc": "2.0",
  "id": "req-1",
  "method": "message/send",
  "params": {
    "message": {
      "role": "user",
      "parts": [{"kind": "text", "text": "Hello"}],
      "messageId": "msg-1"
    }
  }
}
```

Response Options:

Option A: Direct Message (simple response)

```json
{
  "jsonrpc": "2.0",
  "id": "req-1",
  "result": {
    "kind": "message",
    "role": "agent",
    "parts": [{"kind": "text", "text": "Hi there!"}],
    "messageId": "msg-2"
  }
}
```

Option B: Task (long-running operation)

```json
{
  "jsonrpc": "2.0",
  "id": "req-1",
  "result": {
    "kind": "task",
    "id": "task-1",
    "contextId": "ctx-1",
    "status": {"state": "working"}
  }
}
```

---

2. SendStreamingMessage

Same as SendMessage, but with real-time updates.

Request: Same as SendMessage

Response: Server-Sent Events stream

```
data: {"task": {"id": "task-1", "status": {"state": "working"}}}

data: {"statusUpdate": {"taskId": "task-1", "status": {"state": "working", "message": {"parts": [{"text": "Processing..."}]}}}}

data: {"artifactUpdate": {"taskId": "task-1", "artifact": {"parts": [{"text": "Result: ..."}]}}}

data: {"statusUpdate": {"taskId": "task-1", "status": {"state": "completed"}, "final": true}}
```

---

3. GetTask

Retrieve task status.

Request:

```
{
  "jsonrpc": "2.0",
  "id": "req-2",
  "method": "tasks/get",
  "params": {
    "taskId": "task-1",
    "historyLength": 5  # Optional: limit history
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": "req-2",
  "result": {
    "id": "task-1",
    "contextId": "ctx-1",
    "status": {"state": "completed"},
    "artifacts": [...],
    "history": [...]
  }
}
```

---

4. ListTasks

Get multiple tasks with filtering.

Request:

```
{
  "jsonrpc": "2.0",
  "id": "req-3",
  "method": "tasks/list",
  "params": {
    "contextId": "ctx-1",     # Optional filter
    "status": "working",       # Optional filter
    "pageSize": 10,           # Max results
    "pageToken": "",          # For pagination
    "includeArtifacts": false # Reduce payload
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": "req-3",
  "result": {
    "tasks": [...],
    "totalSize": 25,
    "pageSize": 10,
    "nextPageToken": "token-abc"
  }
}
```

---

5. CancelTask

Stop a running task.

Request:

```json
{
  "jsonrpc": "2.0",
  "id": "req-4",
  "method": "tasks/cancel",
  "params": {
    "taskId": "task-1"
  }
}
```

Response:

```json
{
  "jsonrpc": "2.0",
  "id": "req-4",
  "result": {
    "id": "task-1",
    "status": {"state": "cancelled"}
  }
}
```

---

6. SubscribeToTask

Stream updates for an existing task.

Request:

```json
{
  "jsonrpc": "2.0",
  "id": "req-5",
  "method": "tasks/subscribe",
  "params": {
    "taskId": "task-1"
  }
}
```

Response: SSE stream of TaskStatusUpdateEvent and TaskArtifactUpdateEvent

---

7. Push Notifications

For very long tasks, the agent can send webhook notifications.

Set Up Notification:

```
{
  "jsonrpc": "2.0",
  "id": "req-6",
  "method": "tasks/pushNotificationConfig/set",
  "params": {
    "taskId": "task-1",
    "config": {
      "url": "https://client.example.com/webhook",
      "token": "client-secret",
      "authentication": {
        "schemes": ["Bearer"]
      }
    }
  }
}
```

Later, Agent POSTs to Webhook:

```
POST /webhook HTTP/1.1
Host: client.example.com
Authorization: Bearer agent-token
Content-Type: application/json

{
  "statusUpdate": {
    "taskId": "task-1",
    "status": {"state": "completed"},
    "final": true
  }
}
```

Protocol Bindings Explained

A2A supports multiple ways to communicate:

1. JSON-RPC (Default)

Format: JSON-RPC 2.0 over HTTP

Request:

```
POST /v1/message:send HTTP/1.1
Host: agent.example.com
Content-Type: application/json
```

```
{
  "jsonrpc": "2.0",
  "id": "1",
  "method": "message/send",
  "params": { ... }
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "jsonrpc": "2.0",
  "id": "1",
  "result": { ... }
}
```

---

2. gRPC (High Performance)

Format: Protocol Buffers over HTTP/2

Service Definition:

```
service A2AService {
  rpc SendMessage(SendMessageRequest) returns (SendMessageResponse);
  rpc SendStreamingMessage(SendMessageRequest) returns (stream StreamResponse);
  rpc GetTask(GetTaskRequest) returns (Task);
  // ... other methods
}
```

Usage:

```
import grpc
from a2a.v1 import a2a_pb2, a2a_pb2_grpc

channel = grpc.insecure_channel('agent.example.com:443')
stub = a2a_pb2_grpc.A2AServiceStub(channel)

request = a2a_pb2.SendMessageRequest(...)
response = stub.SendMessage(request)
```

---

3. HTTP/REST (RESTful)

Format: RESTful HTTP with JSON

Endpoints:

```
POST   /v1/message:send        - Send message
POST   /v1/message:stream      - Stream message
GET    /v1/tasks/{id}          - Get task
GET    /v1/tasks               - List tasks
POST   /v1/tasks/{id}:cancel   - Cancel task
POST   /v1/tasks/{id}:subscribe - Subscribe
```

Example:

```
POST /v1/message:send HTTP/1.1
Host: agent.example.com
Content-Type: application/json

{
  "message": {
    "role": "user",
    "parts": [{"text": "Hello"}],
    "messageId": "msg-1"
  }
}
```

Method Mapping Table

Operation	JSON-RPC Method	gRPC RPC	REST Endpoint
Send message	`message/send`	`SendMessage`	`POST /v1/message:send`
Stream message	`message/stream`	`SendStreamingMessage`	`POST /v1/message:stream`
Get task	`tasks/get`	`GetTask`	`GET /v1/tasks/{id}`
List tasks	`tasks/list`	`ListTasks`	`GET /v1/tasks`
Cancel task	`tasks/cancel`	`CancelTask`	`POST /v1/tasks/{id}:cancel`
Subscribe	`tasks/subscribe`	`SubscribeToTask`	`POST /v1/tasks/{id}:subscribe`

Field Naming Conventions

Important: JSON uses camelCase, Proto uses snake_case

```
# Protocol Buffer definition
message Task {
  string context_id = 1;  // snake_case
}

# JSON serialization
{
  "contextId": "ctx-1"  // camelCase
}
```

Conversion Rules:

- Proto: `context_id` → JSON: `contextId`
- Proto: `default_input_modes` → JSON: `defaultInputModes`
- Proto: `task_id` → JSON: `taskId`

Next Steps and Resources

Congratulations! You now understand A2A implementation from basics to advanced features.

What You've Learned

☑ Setting up Python environment for A2A ☑ Creating Agent Cards and Skills ☑ Building Agent Executors ☑ Starting and running A2A servers ☑ Creating clients and sending requests ☑ Streaming and multi-turn conversations ☑ Understanding the technical specification ☑ Protocol bindings and data formats

Practice Projects

1. Weather Agent Build an agent that fetches weather data from an API:

- Skill: "Get weather for location"
- Use httpx to call a weather API
- Return temperature, conditions, forecast

2. Calculator Agent Create a multi-turn calculator:

- First message: "calculate 100 + 50"
- Agent: "Result is 150. Need more calculations?"
- Second message: "multiply by 2"
- Agent: "Result is 300"

3. File Processing Agent Build an agent that processes files:

- Accept PDF/image uploads
- Extract text using OCR
- Return structured data as artifacts

Resources

Official Documentation:

- A2A Protocol Website
- A2A GitHub Repository
- A2A Samples Repository
- Protocol Specification

Community:

- GitHub Discussions
- GitHub Issues

Related Protocols:

- Model Context Protocol (MCP)
- Agent Development Kit (ADK)

Next Challenge

Try building a multi-agent system where:

1. **Orchestrator Agent** receives user requests
2. **Specialist Agents** handle specific tasks
    - Weather Agent
    - Calendar Agent
    - Email Agent
3. Agents communicate using A2A protocol
4. Orchestrator combines results and responds to user

This is the real power of A2A - enabling agents from different developers, frameworks, and organizations to work together seamlessly!

Quick Reference

Common Python Patterns

Create Message:

```python
from a2a.types import Message, Part

message = Message(
    message_id=str(uuid4()),
    role='user',
    parts=[Part(text="Hello")],
    context_id='ctx-1'  # Optional
)
```

Create Task:

```
from a2a.types import Task, TaskStatus, TaskState

task = Task(
    id=str(uuid4()),
    context_id=str(uuid4()),
    status=TaskStatus(state=TaskState.working)
)
```

Send Status Update (in executor):

```
await event_queue.enqueue_event(TaskStatusUpdateEvent(
    task_id=task_id,
    context_id=context_id,
    status=TaskStatus(
        state=TaskState.working,
        message=Message(...)
    ),
    final=False
))
```

Send Artifact (in executor):

```
await event_queue.enqueue_event(TaskArtifactUpdateEvent(
    task_id=task_id,
    context_id=context_id,
    artifact=Artifact(
        artifact_id=str(uuid4()),
        name="result.txt",
        parts=[Part(text="Result data")]
    ),
    last_chunk=True
))
```

Troubleshooting

Problem: Server won't start - "Address already in use" **Solution:** Port 9999 is taken. Either kill the process using it, or change the port in your code.

Problem: Client can't connect - "Connection refused" **Solution:** Make sure the server is running and the URL matches.

Problem: "Module 'a2a' not found" **Solution:** Activate your virtual environment and reinstall: `pip install -r requirements.txt`

Problem: Agent responds but client doesn't receive **Solution:** Check that your executor is enqueuing events properly and not raising exceptions.

Problem: Multi-turn conversation doesn't work **Solution:** Make sure you're including `context_id` and `task_id` in follow-up messages.

You're now ready to build your own multi-agent systems with A2A! 🚀