Use the A2A protocol for AI agent communication

A2A, or Agent2Agent protocol is an open standard that enables structured communication between AI agents, clients and tools. In this tutorial, you can build an agent system where a chat client processes user queries and sends them to an AI agent running on an A2A-compliant server.

Most agentic AI applications implement custom communication between components (for example, ChatDev's ChatChain), making it difficult to reuse the same agent across different applications or integrate external tools. This lack of standardization prevents interoperability and limits the development of a broader agent ecosystem.

A2A solves this limitation by separating the communication layer from the agent logic through a standardized protocol built on HTTP, JSON-RPC 2.0, and Server-Sent Events (SSE). This decoupling allows agents to collaborate with other agents, serve client requests, and access external tools without custom integration code.

A2A supports decentralized architectures that allow teams to evolve their AI systems incrementally without breaking client code. Teams can update tools, swap models, or modify agent behavior while maintaining a consistent interface across complex workflows.

Agents exchange information in messages structured in JSON-RPC format that include metadata that enriches agent interactions with clarity and consistency. Each A2A server exposes an AgentCard at a well-known endpoint (.well-known/agent-card.json) that describes the agent's capabilities as structured JSON data. Thus, it allows clients to dynamically discover what an agent can do, similar to how API documentation describes available endpoints.

Follow along to build and run an A2A agent system, and gain hands-on experience with:

- **BeeAI**: An open source agentic framework for building AI agents.
- **A2A protocol**: A standardized communication protocol for agent interoperability.
- **Ollama**: A tool for running large language models (LLMs) locally.
- **Agent tools**: Specialized capabilities including web search (DuckDuckGo), weather data (OpenMeteo), Wikipedia access (WikipediaTool) and reasoning (ThinkTool)

Note: If you've worked with ACP (Agent Communication Protocol), you can recognize similarities. ACP, originally developed by IBM's BeeAI, has joined forces with Google A2A under the Linux Foundation. BeeAI now uses A2A adapters (A2AServer and A2AAgent) to provide A2A-compliant communication. A2A also works alongside MCP (Model Context Protocol) to enable agents to interact with data sources and tools, creating interoperable agent ecosystems.

How the A2A chat system works

This project demonstrates how A2A enables clean separation between the client interface and agent logic.

The workflow follows this sequence:

1. **User input**: The client captures user input through the terminal interface.

2. **A2A request**: The client formats the input as a JSON-RPC message payload and sends it to the agent server.
3. **Agent processing**: The server forwards the request to theRequirementAgent , which analyzes the task and executes appropriate tools as needed.
4. **A2A response**: The server returns the agent's response as structured data in JSON-RPC format, streaming results in real-time as they're generated.
5. **Display**: The client extracts and displays the response text in the terminal.

This workflow demonstrates a reusable pattern applicable to use cases requiring structured client-agent communication such as chatbots, task automation systems, customer support agents and research assistants with tool orchestration.

A2A agent

This project uses a single AI agent with multiple tool capabilities. In more complex systems, you can deploy multiple specialized agents, each focused on specific domains or tasks.

RequirementAgent (BeeAI): A declarative agent that dynamically selects and coordinates multiple tools based on the user's request. It uses:

- ThinkTool for reasoning and logical operations
- DuckDuckGoSearchTool for web searches
- OpenMeteoTool for weather data
- WikipediaTool for information retrieval

The A2A server

The A2A server (beeai-a2a-server/beeai_chat_server.py) exposes agent functionality through an HTTP API. It handles three key responsibilities:

1. **LLM initialization**: Loads a local language model through Ollama

llm = ChatModel.from_name(os.environ.get("BEEAI_MODEL", "ollama:granite3.3:8b"))

2. **Agent setup**: Creates a RequirementAgent with tools and memory to handle the agent lifecycle
agent = RequirementAgent(llm=llm, tools=[ThinkTool(), DuckDuckGoSearchTool(), OpenMeteoTool(), WikipediaTool()], memory=UnconstrainedMemory(), description="An agent that can search the web, check the weather, and think through problems step-by-step.",

3. **Server configuration**: Exposes the agent through A2A-compliant HTTP endpoints
A2AServer(config=A2AServerConfig(port=int(os.environ.get("A2A_PORT", 9999))), memory_manager=LRUMemoryManager(maxsize=100)).register(agent).serve()

The server automatically exposes an AgentCard at /.well-known/agent-card.json that describes the agent's capabilities and helps validate agent configurations.

The A2A client

The A2A client (beeai-a2a-client/beeai_chat_client.py) provides the user interface and handles communication with the server by using the A2A SDK and Python's asyncio library for asynchronous message handling.

Connection setup: Creates an A2A client adapter
agent = A2AAgent(url=os.environ.get("BEEAI_AGENT_URL", "http://127.0.0.1:9999"), memory=UnconstrainedMemory())

Theurl parameter specifies the endpoint of the A2A-compliant server (default:http://127.0.0.1:9999). Thememory parameter stores conversation history locally, allowing the client to maintain context during interactions and support long-running tasks.

Message exchange: Sends asynchronous prompts and processes responses:
for prompt in reader: response = await agent.run(prompt) # Extract and display response text

TheA2AAgent is a client-side adapter that abstracts JSON-RPC communication details. It's not an autonomous agent—it simply converts user input into A2A-compliant messages and handles server responses, enabling seamless data exchange and observability.

Prerequisites to run this project

System requirements

Here are the system requirements to run this project:

- **Operating system**: macOS, Linux or Windows
- **Memory(RAM)**: >= 8GB (**Recommended**: 16GB or more for running local LLMs)
- **Disk space**: >= 5GB free space (**Recommended**: 10GB or more to accommodate the Python environment and local models)
- **Python version**: >= 3.11

Tool and provider requirements

Before you get started, here's an overview of the tools required for this project:

- **BeeAI**: An open source agent development kit for building AI agents. BeeAI supports multiple LLM providers including Ollama (used in this tutorial), OpenAI and Anthropic.
- **Ollama**: For running local LLMs to power the AI agent.
- **A2A protocol**: Integrated into the BeeAI framework to enable structured communication between the client and server.
- **Terminal or IDE**: A terminal or IDE like Visual Studio Code (recommended for managing multiple terminals and viewing logs).
- **Python virtual environment**: To isolate dependencies for the client and server.

LLM provider requirements

This project uses **Ollama** as a model provider for the AI agent. Follow these steps to set up Ollama:

1. Download and install Ollama:
   - Visit Ollama and install the application for your operating system
2. Start the Ollama server
   - Open a terminal and run:
   ollama serve
3. Pull the default model (requires approximately 5GB of disk space):

   ollama pull granite3.3:8b

Note: You can use any Ollama-compatible model by setting theBEEAI_MODEL environment variable. Check the Ollama model library for available models and their sizes.

Steps

Step 1. Clone the GitHub repository

To run this project, clone the GitHub repository by using https://github.com/IBM/ibmdotcom-tutorials.git as the HTTPS URL. For detailed steps on how to clone a repository, refer to the GitHub documentation.

This tutorial can be found inside the projects directory of the repo.

Inside a terminal, navigate to this tutorial's directory:
cd docs/tutorials/projects/a2a_tutorial

Step 2. Set up development environment

This project requires two separate Python scripts to run simultaneously, one for the server and the other for the client. You need to open two terminal windows or tabs.

Keep you current terminal open, then open a second terminal and ensure both are navigated to the correct project directory (thea2a_tutorial root directory).

Using an IDE?

If you're using an IDE like Visual Studio Code, you can use the Split Terminal feature to manage multiple terminals side by side.

Otherwise, open two stand-alone terminal windows and navigate each to the project directory.

Step 3. Create and activate virtual environments

Virtual environments help keep dependencies separate and maintained. To keep the server and client dependencies separate, create a virtual environment for each component.

For the server:

Navigate to thebeeai-a2a-server directory:
cd beeai-a2a-server

Create a virtual environment with Python 3.11:
python3.11 -m venv venv

Activate the virtual environment:
source venv/bin/activate

Note for Windows users: Use venv\Scripts\activate to activate the virtual environment.

For the client:

Navigate to thebeeai-a2a-client directory:
cd beeai-a2a-client

Create and activate a virtual environment:
python3.11 -m venv venv source venv/bin/activate

Step 4. Install dependencies

Install the required dependencies for each component by running this code in each terminal:
pip install -r requirements.txt

You can runpip freeze in the terminal to verify that the environment is up to date with dependencies.

Step 5. Start the A2A agent server

In the first terminal, start the A2A agent server:
python beeai_chat_server.py

You should see:
INFO: Started server process [88159] INFO: Waiting for application startup. INFO: Application startup complete. INFO: Uvicorn running on http://0.0.0.0:9999 (Press CTRL+C to quit)

The server is now listening for incoming requests from the client application, ready to support agent-to-agent communication.

Step 6. Start the A2A client

In the other terminal, start the A2A client:
python beeai_chat_client.py

This should prompt you for input:
Type your message (Ctrl+C to exit): You:

Step 7. Interact with the agent

Type a message in the client terminal and pressEnter . The agent processes your query and responds:
You: What is the capital of France? Agent 🤖 : The capital of France is Paris.

In the server terminal, you can see A2A protocol logs showing the communication with push notifications:

INFO: 127.0.0.1:49292 - “GET /.well-known/agent-card.json HTTP/1.1” 200 OK INFO: 127.0.0.1:49294 - “POST / HTTP/1.1” 200 OK

The first request retrieves the AgentCard that describes the agent’s capabilities. The second request sends your message as aTextPart (a unit of text content within the A2A message) and receives the response.

Note: Outputs from LLMs are probabilistic and can vary each time you run the workflow, even with the same input.

Try different queries

Experiment with different types of queries to test the agent’s various tools:

- **Web search**: “Search for recent news about artificial intelligence”
- **Weather data**: “What’s the weather like in Tokyo?”
- **Wikipedia**: “Tell me about quantum computing”
- **Reasoning**: “What are three reasons why the sky is blue?”

View the AgentCard

Navigate to https://0.0.0.0:9999/.well-known/agent-card.json in your browser to view theRequirementAgent ’s AgentCard.
{ “capabilities”: { “streaming”: true }, “defaultInputModes”: [“text”], “defaultOutputModes”: [“text”], “description”: “An agent that can search the web, check the weather, and think through problems step-by-step.”, “name”: “RequirementAgent”, “preferredTransport”: “JSONRPC”, “protocolVersion”: “0.3.0”, “skills”: [{ “description”: “An agent that can search the web, check the weather, and think through problems step-by-step.”, “id”: “RequirementAgent”, “name”: “RequirementAgent”, “tags”: [] }], “url”: “http://localhost:9999”, “version”: “1.0.0” }

This JSON document describes:

- The agent’s name (RequirementAgent) and a brief description of its capabilities.
- The supported communication protocols and message formats
- Any requirements or constraints

This AgentCard allows any A2A-compliant client to discover and interact with the agent without prior knowledge of its implementation details.

Conclusion

In this tutorial, you built a chat system by using an A2A-complaint server that exposed a structured interface for client-agent communication. By separating the messaging layer from internal logic, the Agent2Agent protocol enables teams to update agent capabilities, swap models or modify tool configurations without changing client code. This flexibility is especially valuable when coordinating input-required tasks, tracking task status or treating each operation as a discrete unit of work.

A2A works by defining a common message format that any compliant component can understand, allowing autonomous agents to collaborate with other agents. The protocol specification

defines how messages are structured in JSON-RPC format and enriched with metadata to ensure consistency and clarity across interactions.

This tutorial builds on the foundational examples provided by the A2A samples repository. For more information about the original implementation, refer to the readme file in the repository, which provides more context and examples for building A2A-compliant systems.

For real-world deployments, A2A servers can implement authentication mechanisms to secure agent endpoints, use server-sent events for streaming responses, and scale to handle production workflows. By following this workflow, you saw how a command line client can interact with an AI agent through a standardized protocol, enabling the agent to coordinate multiple tools and provide contextual responses. This approach demonstrates the power of A2A to enable maintainable, scalable and flexible AI systems.

Shutting down the system

When you're done experimenting with the system, follow these steps to cleanly shut down all running components:

Stop each running server

In each terminal window, press Ctrl+C to stop the running process.

You should see output like:
INFO: Shutting down INFO: Finished server process

If the server hangs during a shutdown

If the server becomes unresponsive or hangs on shutdown, you can forcefully stop it:

Find the process ID (PID):
ps aux | grep python

Identify the PID of the process that you're trying to stop.

End the process:
kill -9 <PID>

Repeat this process for each server if needed.

That's it. You've successfully run a complete A2A-compliant chat system.