What is Agent Communication Protocol (ACP)?

Agent Communication Protocol (ACP) is an open standard for agent-to-agent communication. With this protocol, we can transform our current landscape of siloed agents into interoperable agentic systems with easier integration and collaboration.

Note: *The information provided in this explainer about ACP may not reflect its current status, as ACP has merged with A2A under the Linux Foundation umbrella. The ACP team is winding down active development and contributing its technology and expertise to A2A, which may impact the relevance and accuracy of the content presented. Users of ACP are advised to refer to the official migration paths and documentation for transitioning to A2A.*

Your subscription will be delivered in English. You will find an unsubscribe link in every newsletter. You can manage your subscriptions or unsubscribe here. Refer to our IBM Privacy Statement for more information.

Overview

With ACP, originally introduced by IBM's BeeAI, AI agents can collaborate freely across teams, frameworks, technologies and organizations. It's a universal protocol that transforms the fragmented landscape of today's AI agents into interconnected teammates and this open standard unlocks new levels of interoperability, reuse and scale. As the next step following the Model Context Protocol (MCP), an open standard for tool and data access, ACP defines how agents operate and communicate.[1]

For context, an AI agent is a system or program that is capable of autonomously performing tasks on behalf of a user or another system. It performs them by designing its workflow and by using available tools. Multi-agent systems consist of multiple AI agents working collectively to perform tasks on behalf of a user or another system.

As an AI agent communication standard with open governance, ACP allows artificial intelligence agents to communicate across different frameworks and technology stacks. From taking in user queries in the form of natural language to performing a series of actions, AI agents perform better when provided with communication protocols. These protocols relay this information between tools, other agents and ultimately, to the user.

AI agent communication refers to how artificial intelligence agents interact with each other, humans or external systems to exchange information, make decisions and complete tasks. This communication is especially important in multi-agent systems, where multiple AI agents collaborate, and in human-AI interaction.

ACP is part of a growing ecosystem, including BeeAI. The following are some key features and you can read more about the core concepts and details in the official documentation.

Key features of ACP

- **REST-based communication:** ACP uses standard HTTP conventions for communication that makes it easy to integrate into production. Whereas MCP relies on the JSON-RPC format that requires much more complex communication methods.
- **No SDK required:** ACP doesn't require any specialized libraries. You can interact with intelligent agents by using tools like cURL, Postman or even your browser. For added convenience, there is an SDK available.
- **Offline discovery:** ACP agents can embed metadata directly into their distribution packages, which enables discovery even when they're inactive. This supports scale-to-zero environments, where resources are dynamically allocated and might not always be online.
- **Async-first, sync supported:** ACP is designed with asynchronous communication as the default. This method is ideal for long-running or complex tasks. Synchronous requests are also supported.

Note: ACP enables agent orchestration for any agentic architecture, but it doesn't manage workflows, deployments or coordination between agents. Instead, it enables orchestration across diverse agents by standardizing how they communicate. IBM Research built BeeAI, an open source system designed to handle agent orchestration, deployment and sharing by using ACP as the communication layer.

Why do we need ACP

As agentic AI continues to rise, so does the amount of complexity in navigating how to get the best outcome from each independent technology for your use case, without being constrained to a particular vendor. Each framework, platform and toolkit offer unique advantages but integrating them all into one agentic system is challenging.

Today, most agent systems operate in silos. They're built on incompatible frameworks, expose custom APIs and endpoints and lack a shared protocol for communication. Connecting them requires fragile and nonrepeatable integrations that are expensive to build.

ACP represents a fundamental shift: from a fragmented, ad hoc ecosystem to an interconnected network of agents—each able to discover, understand and collaborate with others, regardless of who built them or what stack they run on. With ACP, developers can harness the collective intelligence of diverse agents to build more powerful workflows than a single system can achieve alone.

Current challenges

Despite rapid growth in agent capabilities, real-world integration remains a major bottleneck. Without a shared communication protocol, organizations face several recurring challenges:

- Framework diversity: Organizations typically run hundreds or thousands of agents built by using different frameworks like LangChain, crewAI, AutoGen or custom stacks.
- Custom integration: Without a standard protocol, developers must write custom connectors for every agent interaction.
- Exponential development: With n agents, you potentially need n(n-1)/2 different integration points, which makes large-scale agent ecosystems difficult to maintain.
- Cross-organization considerations: Different security models, authentication systems and data formats complicate integration across companies.

A real-world example

To illustrate the real-world need for agent-to-agent communication, consider two organizations:

- A manufacturing company that uses an autonomous agent to manage production schedules and order fulfillment based on internal inventory and customer demand.
- A logistics provider that runs an agent to offer real-time shipping estimates, carrier availability and route optimization.

Now imagine the manufacturer's system needs to estimate delivery timelines for a large, custom equipment order to inform a customer quote.

Without ACP: This approach requires building a bespoke integration between the manufacturer's planning software and the logistics provider's APIs. This means handling authentication, data format mismatches and service availability manually. These integrations are expensive, brittle and hard to scale as more partners join.

With ACP: Each organization wraps its agent with an ACP interface. The manufacturing agent sends order and destination details to the logistics agent, which responds with real-time shipping options and ETAs. Both systems perform agentic collaboration without exposing internals or writing custom integrations. New logistics partners can be introduced simply by implementing ACP. The automation that AI agents paired with ACP provide allows for scalability and streamlining data exchanges.

How to get started

Simplicity is a core design principle of ACP. Wrapping an agent with ACP can be done in just a few lines of code. Using the Python SDK, you can define an ACP-compliant agent by simply decorating a function.

This minimal implementation:

1. Creates an ACP server instance.
2. Defines an agent function by using the @server.agent() decorator.
3. Implements an agent by using the LangChain framework with a large language model (LLM) backend and memory for context persistence.
4. Translates between ACP's message format and the framework's native format to return a structured response.
5. Starts the server, making the agent available via HTTP.

Code example:

```
from typing import Annotated
import os
from typing_extensions import TypedDict
```

```
from dotenv import load_dotenv
#ACP SDK
from acp_sdk.models import Message
from acp_sdk.models.models import MessagePart
from acp_sdk.server import RunYield, RunYieldResume, Server
from collections.abc import AsyncGenerator
#Langchain SDK
from langgraph.graph.message import add_messages
from langchain_anthropic import ChatAnthropic

load_dotenv()

class State(TypedDict):
    messages: Annotated[list, add_messages]

#Set up the AI model of your choice
llm = ChatAnthropic(model="claude-3-5-sonnet-latest",
api_key=os.environ.get("ANTHROPIC_API_KEY"))

#------ACP Requirement-------#
#START SERVER
server = Server()
#WRAP AGENT IN DECORACTOR
@server.agent()
async def chatbot(messages: list[Message])-> AsyncGenerator[RunYield, RunYieldResume]:
    "A simple chatbot enabled with memory"
    #formats ACP Message format to be compatible with what langchain expects
    query = " ".join(
        part.content
        for m in messages
        for part in m.parts
    )
    #invokes llm
    llm_response = llm.invoke(query)
    #formats langchain response to ACP compatible output
    assistant_message = Message(parts=[MessagePart(content=llm_response.content)])
    # Yield so add_messages merges it into state
    yield {"messages": [assistant_message]}

server.run()
#--------------------------#
```

Now, you've created a fully ACP-compliant agent that can:

- Be discovered by other agents (online or offline).
- Process requests synchronously or asynchronously.
- Communicate by using standard message formats.
- Integrate with any other ACP-compatible system.

How ACP compares to MCP and A2A

To better understand ACP's role in the evolving AI ecosystem, it helps to compare it to other emerging protocols. *These protocols aren't necessarily competitors.* Instead, they address different layers of the AI system integration stack and often complement one another.

At a glance:

Model Context Protocol (MCP): Designed for enriching a single model's context with tools, memory and resources. Introduced by Anthropic.

Focus: one model, many tools

Agent Communication Protocol (ACP): Designed for communication between independent agents across systems and organizations. Introduced by IBM's BeeAI.
Focus: many agents, securely working as peers, no vendor lock in, open governance

Agent2Agent Protocol (A2A): Designed for communication between independent agents across systems and organizations. Introduced by Google.

Focus: many agents, working as peers, optimized for Google's ecosystem

ACP and MCP

The ACP team initially explored adapting the Model Context Protocol (MCP) because it offers a strong foundation for model-context interactions. However, they quickly encountered architectural limitations that made it unsuitable for true agent-to-agent communication.

Why MCP falls short for multi-agent systems:

Streaming: MCP supports basic streaming, likely of complete messages, but not the finer-grained "delta" style, where updates are sent as soon as they happen. Delta streams, such as tokens and trajectory updates, are streams composed of incremental updates rather than complete data payloads. This limitation stems from the fact that when MCP was created, it wasn't intended for agent-style interactions.

Memory sharing: MCP doesn't support running multiple agents across servers while maintaining shared memory. ACP doesn't fully support this function yet either, but it's an active area of development.

Message structure: MCP accepts any JSON schema but doesn't define the structure of the message body. This flexibility makes interoperability difficult, especially for building agents that must interpret diverse message formats.

Protocol complexity: MCP uses JSON-RPC and requires specific SDKs and runtimes. Whereas ACP's REST-based design with built-in async/sync support is more lightweight and integration-friendly.

Think of **MCP** as giving a person better tools, like a calculator or a reference book, to enhance their performance. In contrast, **ACP** is about enabling people to form *teams*, where each person, or agent, contributes their capabilities collaboratively within the AI application.

ACP and MCP can complement each other:

Google's Agent2Agent protocol (A2A), which was introduced shortly after ACP, also aims to standardize communication between AI agents. Both protocols share the goal of enabling multi-agent systems, but they diverge in philosophy and governance.

Key differences:

ACP was deliberately designed to be:

- Simple to integrate by using common HTTP tools and REST conventions.
- Flexible across a wide range of agent types and workloads.

- Vendor-neutral, with open governance and broad ecosystem alignment.

Both protocols can coexist–each serving different needs depending on the environment. ACP's lightweight, open and extensible design makes it well suited for decentralized systems and real-world interoperability across organizational boundaries. A2A's natural integration might make it a more suitable option for those using the Google ecosystem.

Roadmap and community

As ACP evolves, new possibilities to enhance agent communication are being explored. Here are some key areas of focus:

- Identity federation: How can ACP work with authentication systems to improve trust across networks?
- Access delegation: How can we enable agents to delegate tasks securely while maintaining user control?
- Multiregistry support: Can ACP support decentralized agent discovery across different networks?
- Agent sharing: How can we make it easier to share and reuse agents across organizations or within an organization?
- Deployments: What tools and templates can simplify agent deployment?

ACP is being developed with open source because standards work best when they're developed directly with users. Contributions from developers, researchers and organizations interested in the future of AI agent interoperability are welcome. Join in helping to shape this evolving gen AI standard.

For more information, visit the official ACP site and join the conversation on the GitHub and Discord.