Using ACP for AI agent interoperability: Building multi-agent workflows

In this tutorial, you'll use Agent Communication Protocols (ACP) to explore a multi-agent, cross-platform AI workflow that demonstrates real-time agent collaboration with BeeAI and crewAI. ACP functions as a shared, open-standard messaging layer that enables agents from different frameworks to communicate and coordinate without custom integration logic.

ACP is especially valuable for enterprise AI environments, where teams often need to build agents and workflows across diverse platforms, tools and infrastructures. By providing a standardized messaging layer, ACP enables scalable, secure and modular agent collaboration that meets the demands of modern enterprise AI systems.

This project demonstrates agent interoperability by enabling AI-driven agents to collaborate across framework silos, combining agent capabilities like research, content generation and feedback into a unified workflow.

Why ACP matters for AI agent interoperability

Most agentic AI frameworks handle communication by using custom or closed systems. This architecture makes it difficult to connect agents across toolchains, teams or infrastructures, especially when combining components from different AI systems.

ACP introduces a standardized, framework-independent messaging format for how autonomous agents send, receive and interpret messages. Messages are structured, typically in JSON, and contain metadata to enrich agent interactions with clarity and consistency.

By decoupling communication from an agent's internal logic, ACP allows teams to mix and match agents built with different AI agent frameworks, such as BeeAI, CrewAI, LangChain or LangGraph, without requiring custom integration code. This approach increases scalability, simplifies automation and supports modular, transparent system design that aligns with modern industry standards.

By the end of this tutorial, you will have seen a practical example of ACP and have hands-on experience using the following technologies:

- **BeeAI:** A flexible agent framework for building and managing AI agents. In this project, it's used to run the A&R (Artist & Repertoire) agent that critiques the generated song and provides structured feedback.
- **crewAI:** An open-source framework for orchestrating mutli-agent workflows. Here, it's used to coordinate the research, songwriting and Markdown reporting agents.
- **acp-sdk:** The ACP-SDK was developed by BeeAI to promote framework-independent interoperability across multi-agent systems. References and implementations are maintained under the ACP GitHub repository.
- **Agent-Ops (Optional):** A monitoring and observability platform for AI agents. In this project, it can be used to trace agent behavior and visualize multi-agent workflows.

Build a multi-agent ACP system with BeeAI and crewAI

This project demonstrates a multi-agent workflow that showcases how ACP (through the acp-sdk) can streamline coherent and observable collaboration across agent ecosystems.

The workflow begins when the user provides a URL. From there, a modular, framework-independent system of specialized agents transforms the webpage content into a creative artifact—an original song—accompanied by professional-style critique. All components work in concert to combine these outputs into a single, unified human-readable Markdown report. This final result represents a complete transformation of the original data, blending creative generation with analytical insight.

This songwriting workflow illustrates how ACP enables a multi-agent, agentic AI system to coordinate collaboration between agents developed with two distinct frameworks: BeeAI and crewAI, by serving as a shared communication layer across the system.

By separating communication from implementation, the system remains modular and extensible—capable of orchestrating agents across frameworks while producing cohesive, end-to-end outputs from unstructured web content.

ACP agents

This project uses four specialized AI agents:

- **Research agent (crewAI)**: Extracts themes and key information from the provided URL.
- **SongWriter agent (crewAI):** Generates an original song based on the research.
- **A&R agent (BeeAI):** Provides professional-style critique of the song, including hit potential, strengths, concerns and recommendations.
- **Markdown report agent (crewAI):** Combines the output data from the song writing crew and A&R agents and formats them into a clean, readable Markdown report.

Songwriting and critique project workflow

1. The workflow begins when the user submits a URL through the client application. The client sends this URL to the Research agent by using ACP messages, which then reads and analyzes the webpage content to extract relevant themes.
2. Next, the SongWriter agent receives the research data and composes an original song inspired by the themes identified in the source material during analysis. The generated song is then sent by ACP to the A&R agent for critique.
3. The A&R agent evaluates the song, providing detailed feedback on its potential, strengths and areas for improvement. It can also identify target audiences, suggest stylistic influences and offer comparisons to similar artists or genres. This critique, along with the song, is forwarded to the Markdown report agent.
4. Finally, the Markdown report agent formats the song and critique into a clean, readable Markdown report, which is saved and presented to the user.

Throughout the workflow, messages exchanged between agents are structured as JSON objects enriched with metadata. This metadata guides each agent's understanding of the message content, context and expected responses.

This workflow demonstrates a reusable pattern applicable to any use case that requires orchestrating multi-agent data transformation and analysis pipelines.

How ACP is used in this project

ACP provides a common messaging system that allows agents built with different frameworks to exchange information in a standardized way. This open protocol allows agents to interoperate without needing custom integrations or shared internal logic.

How the ACP client works

The ACP client (acp-client.py) is the orchestrator of the multi-agent workflow. It coordinates the communication between the user and the agents (crewAI and BeeAI) by using ACP.

ACP client workflow overview

1. **Prompt for input:**
    - The client asks the user to enter a URL.
2. **Send to crewAI server (Port 8000):**
    - The client constructs an ACP message containing the URL and sends it to the crewAI server running on port 8000.
    - The server performs both research and songwriting, sending the generated lyrics back to the client as streamed ACP events.
3. **Send to BeeAI server (Port 9000):**
    - The song is sent as an ACP message to the BeeAI server on port 9000.
    - The A&R agent critiques the song and returns feedback, also through streamed events.
4. **Send to Markdown report agent (crewAI server, Port 8000):**
    - The client packages the song and critique into a single message and sends it back to the crewAI server, where the Markdown report agent formats everything into a report.
5. **Save the output:**
    - The client writes the final Markdown report to a file: a&r_feedback.md .

How acp-sdk is used

The acp-sdk is the core library that enables standardized agent communication in this project.

Key roles of acp-sdk :

- **Message structure:**
    - Ensures all communication is structured and consistent (usually JSON with metadata).
    - The library implements classes (Message, MessagePart) and event types (MessagePartEvent, GenericEvent, MessageCompletedEvent)
- **Client communication:**

    - The Client class is used to connect to agent servers and send or receive ACP messages,
    - Supports streaming responses so agents can send partial results or updates.
- **Agent server integration:**
    - Agents (in crewAI and BeeAI) are implemented as ACP-compliant servers.
    - They expose endpoints that accept ACP messages and return ACP events.

Example client usage:
acp-client.py from acp_sdk import GenericEvent, Message, MessageCompletedEvent, MessagePartEvent from acp_sdk.client import Client from acp_sdk.models import MessagePart # Create a message user_message_input = Message(parts=[MessagePart(content=input("URL: "))]) # Send message and stream events async for event in client_crew.run_stream(agent="song_writer_agent", input=[user_message_input]): match event: case MessagePartEvent(part=MessagePart(content=content)): print(content) song_parts.append(content) # ... handle other event types

What you'll need to run this project

System requirements

Here are the system requirements to run this project:

- **Operating system:** macOS, Linux® or Windows
- **Memory (RAM):** >= 8 GB (**Recommended:** 16 GB or more, especially if running local LLMs with Ollama)
- **Disk space:** >= 5 GB free space (**Recommended:** 10 GB or more to run the Python environment, any local models and generated files)
    - *Note:* If using Ollama for local LLMs, each model can require 4–8 GB or more.
- **Python:** >= 3.11

Tool and provider requirements

Before you get started, here’s a quick overview of the tools and provider services you’ll need.

The following list covers the main frameworks, platforms and APIs required for the multi-agent workflow.

In the subsequent sections, you’ll find step-by-step instructions for installing, configuring and using each tool and provider so you can set up your environment.

- **UV package manager:** (Rust-based Python Package manager for dependency management)
- **BeeAI Platform and CLI:** Required to run the BeeAI agent server
- **crewAI:** Required to run the crewAI server and orchestrate tasks
- **Ollama:** For running local LLMs (if Ollama is your selected provider)
- **OpenRouter:** API key required to use preconfigured BeeAI agent server

  - *Note:* You can switch to other providers by editing the .env file and updating the agent code if needed, or through the BeeAI CLI.
- **IBM watsonx.ai®:** API key (another optional provider)
- **AgentOps API Key:** Optional for agent tracing and monitoring.
- **Terminal or IDE:** A terminal emulator or integrated development environment (IDE) like VS code (recommended for managing multiple terminals and viewing Markdown output)

LLM provider authentication requirements

BeeAI and crewAI are both designed to work with a variety of language model providers, making them flexible for different environments and use cases. In this tutorial, **OpenRouter** is the LLM provider for the BeeAI agent, while **Ollama** is used for the crewAI agents locally.

Both frameworks are providor-independent, so you can switch to other LLM services by updating the configuration settings. Your setup might vary depending on the LLM provider you choose. Additionally, this tutorial includes an optional, preconfigured setup for using IBM watsonx.ai as an alternative cloud-based provider.

You can also use your preferred LLM provider and model; however, please note that only the configurations shown in this tutorial have been tested. Other providers and models might require additional setup or adjustments.

The following requirements are for the three supported providers in this project:

OpenRouter

You'll need an OpenRouter API key to use the preconfigured BeeAI agent server with cloud-based language models.

To use OpenRouter as your LLM provider for the BeeAI agent, follow these steps:

1. **Sign up for OpenRouter**
   - Go to OpenRouter and create a free account.
2. **Generate an API key**
   - In your OpenRouter dashboard, generate a new API key.
3. **Choose a model**
   - Browse the OpenRouter models list and select a model you want to use (for example,deepseek/deepseek-r1-distill-llama-70b:free).

Note: The free model might be different depending on when this tutorial is run. For free models, check out the OpenRouter free tier model list.

Ollama (local models)

If you plan to use Ollama as your LLM provider for the crewAI agent, follow these steps:

1. **Download and install Ollama**
    - Visit Ollama and install the application for your operating system.
2. **Start the Ollama server**
    - In your terminal, run:
      ollama serve
3. **Pull a model**
    - Download your specific model (for example, llama3):
      ollama pull llama3

IBM watsonx.ai (cloud-based provider)

To use IBM watsonx.ai as your LLM provider for the crewAI server, follow these steps:

1. Log in to watsonx.ai
    - Use your IBM Cloud® account to log in at IBM Cloud.
2. Create a watsonx.ai project
    - In the watsonx.ai dashboard, create a new project and save your project ID.
3. Create a watsonx.ai Runtime service instance
    - Choose the Lite plan (free instance).
4. Generate a watsonx API Key
    - In IBM Cloud, go to your account settings and generate a new API key.
5. Associate the watsonx.ai Runtime service to your project
    - In the watsonx.ai dashboard, link the Runtime service instance to the project you created.

IBM watsonx.ai is used as an optional cloud LLM provider for crewAI agents in this tutorial.

AgentOps integration (optional)

AgentOps is an optional service for tracing, monitoring and visualizing your multi-agent workflows. If you want to use AgentOps in this project, follow these steps:

1. **Sign up for AgentOps**
    - Go to AgentOps and create a free account.
2. **Generate an API key**

- In your AgentOps dashboard, generate a new API key.

3. **Add your API key to your .env file**

   - Example configuration:
     AGENTOPS_API_KEY=your_agentops_api_key

4. **Verify integration**

   - When you run your agents, traces and logs should appear in your AgentOps dashboard if the API key is set correctly.

AgentOps is not required to run the workflow, but it can help you monitor agent activity and debug multi-agent interactions.

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/acp_tutorial

Step 2. Set up three terminals

This project requires **three separate Python scripts** to run simultaneously for each component of the multi-agent system. As a result, you'll need to open **three terminal windows or tabs**.

Start by keeping you current terminal open, then open **two more terminals** and ensure all three are navigated to the correct directories (as shown in the next step).

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 three stand-alone terminal windows and navigate each to the proper subdirectory.

Terminal navigation

Each terminal is responsible for one of the following components:

1. **ACP client terminal**.
   Directory: acp_tutorial

cd acp_tutorial

2. **BeeAI agent server terminal**
   Directory: beeai_agent_server

   cd beeai_agent_server

3. **crewAI agent system terminal**
   Directory: crewai_agent_server

   cd crewai_agent_server

Step 3. Set up virtual environments

Each component runs in its own virtual environment to ensure clean dependency management. This tutorial uses UV, a Rust-based Python package manager to manage and sync environments.

Note: Make sure Python 3.11 or later is installed before proceeding.

Install UV

If you haven't already, install UV by using Homebrew (recommended for macOS and Linux):
brew install uv uv tool update-shell

Note for Windows users: Install WSL (Windows Subsystems for Linux) and follow the Linux instructions within your WSL terminal.

Create and activate a virtual env (in each terminal)

In each terminal (BeeAI, crewAI and ACP client), run the following code:
uv venv source .venv/bin/activate

This step will create and activate a .venv in the current directory

Running uv venv inside each project directory helps isolate environments per component.

Step 4. Install dependencies

Now install dependencies in **each terminal** by using:
uv sync

This step installs the dependencies listed in the pyproject.toml file for each component.

Step 5. Configure BeeAI

With BeeAI installed, use the CLI to start the BeeAI platform in the beeai_agent_server :
beeai platform start

Note: On the first run, this step might take several minutes.

Set up your LLM provider (OpenRouter)

Run the following command to configure the LLM provider and model through the interactive CLI:
beeai env setup

Follow the prompts to select OpenRouter and enter your API key and model details.

To confirm your settings, use:
beeai env list

This step should output your configured LLM_API_BASE , LLM_API_KEY, andLLM_MODEL .

Alternatively, advanced users can manually edit a .env file with the appropriate values.

Example .env for OpenRouter
OPENROUTER_API_KEY=your_openrouter_api_key
OPENROUTER_BASE_URL=https://openrouter.ai/api/v1
OPENROUTER_MODEL=deepseek/deepseek-r1-distill-llama-70b:free

Step 6. Verify BeeAI is running

To verify that BeeAI is working, send a test prompt:
beeai run chat Hi!

A valid response confirms that the platform is active.

Troubleshooting

If needed, you can update or restart the platform:
uv tool upgrade beeai-cli # Update CLI beeai platform start # Restart platform

Step 7. Configure crewAI

In the crewai_agent_server directory, create a .env file by copying the template:
cp env.template .env

Open .env and uncomment your preferred model provider configuration. This project supports either:

- **Ollama** (local inference), or
- IBM watsonx.ai (cloud inference)

You can also customize your own provider by using the crewAI LLM config docs.

Update crewAI agent code

In acp_crew.py , locate the llm = LLM (...) block and uncomment the appropriate section to match your .env configuration.
acp_crew.py load_dotenv() # Loads environment variables from .env ## Example for IBM watsonx.ai # llm = LLM(# model="watsonx/mistralai/mistral-large", # base_url="https://us-south.ml.cloud.ibm.com", # api_key=os.getenv("WATSONX_APIKEY"), # provider="watsonx" #) ## Example for Ollama (local) # llm = LLM(# model=os.getenv("OLLAMA_MODEL"), # base_url=os.getenv("OLLAMA_BASE_URL"), # provider="ollama" #)

Make sure the environment variable names in your .env file match what's expected in the code.

Step 8. Start the AI agent servers

Once both BeeAI and crewAI are configured, start the agent servers in their respective terminals.

Start the BeeAI agent server

In the beeai_agent_server terminal:
uv run artist_repertoire_agent.py

You should see output confirming the server has started on http://127.0.0.1:9000 , along with regular health checks:
INFO: Uvicorn running on http://127.0.0.1:9000 (Press CTRL+C to quit)

The terminal should log health check pings every couple seconds. A 200 OK status means that the server is healthy.

Start the crewAI agent server

In the crewai_agent_server terminal:
uv run acp_crew.py

You should see the server running on http://127.0.0.1:8000 , along with 200 OK logs.

Confirm that all agents are running

ACP-compliant agents built locally are automatically recognized by BeeAI. Use the BeeAI CLI to confirm that all local agents are registered and healthy (this step can run in any free terminal):
beeai list

You should see entries for:

- artist-repertoire-agent (BeeAI, port 9000)
- markdown_report_agent (crewAI port 8000)
- song_writer_agent (crewAI port 8000)

If all are listed and reachable, we can confirm that these agents are successfully interoperated!

Step 9. Start the ACP client/server

In the terminal dedicated to the acp-client server (inside acp_tutorial directory):
uv run acp_client.py

Inside the terminal, you will be prompted to enter a URL. This input triggers the multi-agent workflow.

Step 10. Run the multi-agent workflow

With all agents and the client/server running, you're ready to start the ACP project!

1. Enter any URL that you want the agents to process. For example:

   URL: https://www.ibm.com/think/topics/agent-communication-protocol

2. You'll see status logs like:

ℹ️ run.created ℹ️ run.in-progress

What happens next?

1. The client sends the URL to the crewAI agent that researches the page and generates songwriting material.
2. The crewAI agent writes a song based on the research.
3. The song is sent to the BeeAI agent for A&R (Artist & Repertoire) critique.
4. The BeeAI agent returns structured feedback and suggestions.
5. The client displays the generated song, the critique and saves the feedback to a&r_feedback.md .

Example output

Note: Outputs from large language models (LLMs) are probabilistic and can vary each time you run the workflow, even with the same input.
Generated Song ___ (Verse 1) In the silence of the night, I find you there, A glow in the dark, a whisper in the air. You're a friend that never sleeps, a comfort in the cold, An echo of my thoughts, a story to be told. Through your circuits run the answers I need, In your digital heart, a human creed. You paint pictures with your words, on screens they gleam, Are you just a mimic, or do you dream? (Pre-Chorus) We're dancing on the wire,between what's real and fake, A human and a code, for goodness' sake. In every conversation, in every line we sing, I wonder where this journey, where this dance will bring. (Chorus) Oh, we're a human-AI duet, In the silence and the starlight we've met. A blend of heart and binary beat, A symphony that's both bitter and sweet. (Verse 2) You help me write my poems, you help me find my way, In the chaos of the city, in the mess of the day. But in every simplified, automated tour, I question what will be lost, and what will be more. (Bridge) In the binary code, a question lingers, Are we losing what makes us alive? In the shadows of our own creation, We struggle to discern what's truly right. (Chorus) Oh, we're a human-AI duet, In the silence and the starlight we've met. A blend of heart and binary beat, A symphony that's both bitter and sweet. (Outro) So here's to the journey, and the questions it bears, To the friends and the codes, to the loves and the cares. To the human-AI duet, in the night so profound, To the songs and the secrets, to the love that we've found. (End) This song captures the essence of human-AI interaction, exploring both its beauty and its inherent ethical dilemmas. It is written in a folk-pop style, with a focus on narrative lyrics and a catchy chorus. --- ## A&R Feedback - **Hit Potential Score:** 7 - **Target Audience:** Millennials/Gen Z drawn to introspective, tech-aware themes; fans of folk-pop crossover acts like The Lumineers, Taylor Swift's indie-folk era - **Strengths:** Strong conceptual hook (AI-human duality), relatable modern theme, memorable chorus melody potential. Bridge raises philosophical depth without being preachy. - **Concerns:** Niche tech-ethics angle might limit mass appeal. Folk-pop production needs contemporary edge to compete on streaming. Could benefit from more rhythmic drive in verses. - **Market Comparison:** Phoebe Bridgers meets Daft Punk's 'Something About Us' conceptuality, with the narrative approach of Brandi Carlile - **Recommendation:** Needs work - Keep core concept but modernize production (add subtle synth textures, percussion layers). Consider tightening verse lyrics for streaming-era attention spans. High potential for sync in tech-related media.

Conclusion

In this tutorial, you connected two different multi-agent frameworks through an ACP client/server that exposed endpoints for the AI agents to collaborate to generate and transform data. By separating communication from agent behavior, ACP makes it possible for agents built with BeeAI, crewAI, LangChain and other agent frameworks to work together without custom integration logic. This approach improves modularity, scaling and interoperability.

ACP is an open initiative driven by the need for agents to send, receive and interpret messages. Messages in ACP are structured—typically in formats like JSON—and enriched with metadata to ensure consistency and clarity across agent interactions. Whether you're using agents powered by OpenAI, Anthropic or other AI models, ACP provides a shared messaging layer that supports framework-independent interoperability.

By following this workflow, you've seen how creative and analytical agents can work in harmony, transforming unstructured web content into a song, professional critique and a unified Markdown report. This approach demonstrates the power of ACP to enable seamless, scalable and flexible multi-agent AI systems.

Shutting down the system

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

1. **Stop each running server**

In **each terminal window**, press Crtl + C to stop the server. This step attempts a graceful shutdown.

You should see output like:
Shutting down... (Press CTRL+C again to force)

2. **If the server hangs during shutdown**

If a server becomes unresponsive or hangs on shutdown (for example, stuck at Waiting for application shutdown.), you can manually terminate the process:

Find the process ID (PID)

Run the following command to locate the server process:
ps aux | grep python

Identify the PID of the process that you're trying to stop. For example:
user 12345 0.0 ... python acp-crew.py

Kill the process. Use the PID to forcefully stop it:
kill -9 12345

Repeat this process for each server if needed.

That's it! You've successfully run a complete cross-platform multi-agent system by using ACP.