The Definitive Guide to the Agent2Agent (A2A) Protocol: Architecting and Building Interoperable AI Agents

Part I: Protocol Foundations and Architecture

1. An Introduction to the Agent Interoperability Challenge

The field of artificial intelligence is undergoing a significant architectural shift. Enterprises are moving beyond monolithic AI models to deploy fleets of specialized, autonomous agents designed to automate and enhance specific business processes.[1] These systems, often built using a diverse array of agentic frameworks like Google's Agent Development Kit (ADK), LangChain's LangGraph, CrewAI, and Microsoft's AutoGen, offer powerful, targeted capabilities.[2] However, this rapid, heterogeneous development has inadvertently created a new and formidable challenge: digital fragmentation.

The Rise of Siloed Agentic Systems

The current AI landscape is characterized by a proliferation of "siloed" agentic systems. An agent designed for supply chain optimization within one framework cannot natively communicate with a customer service agent built on another. This lack of interoperability forces human operators into the role of a "human API," manually transferring data and context between systems to complete a single, complex workflow.[5] This manual intervention negates many of the efficiency gains promised by AI automation and creates a significant barrier to building truly scalable, end-to-end intelligent processes. Each agent, while an expert in its domain, operates in isolation, unable to collaborate, delegate, or coordinate with its peers

across the enterprise. This fundamental disconnect prevents the realization of a true multi-agent ecosystem where the whole is greater than the sum of its parts.

A2A as the "HTTP for Agents": Vision and Goals

The Agent2Agent (A2A) protocol was introduced to solve this critical interoperability problem. Initiated by Google in April 2025 with support from over 50 technology partners and now housed by the Linux Foundation as an open-source project, A2A is designed to be the "HTTP for agents".[1] Its vision is to establish a universal language, a common communication standard that enables any AI agent, regardless of its underlying framework, vendor, or domain, to discover, communicate, and collaborate with any other agent.[7]

The primary goal of A2A is to break down the digital silos that currently define the agentic landscape. By providing a standardized, vendor-neutral protocol, it fosters an open and interconnected ecosystem where specialized agents can be composed into powerful, multi-functional applications.[1] This allows developers to build agents that can connect with any other A2A-compliant agent and gives enterprises the flexibility to combine best-of-breed agents from various providers, reducing vendor lock-in and lowering long-term costs.[1]

Core Design Principles

The architecture of the A2A protocol is guided by five fundamental design principles, ensuring its practicality, security, and ease of adoption within enterprise environments.[1]

- **Embrace Agentic Capabilities:** A2A is fundamentally designed for peer-to-peer collaboration between autonomous entities. It treats agents as intelligent collaborators, not merely as tools to be called. The protocol enables natural, potentially unstructured communication modalities, allowing agents to coordinate on complex tasks without requiring shared memory, tools, or internal context. This principle is crucial for enabling true multi-agent scenarios where agents can delegate, negotiate, and reason together.[1]
- **Build on Existing Standards:** To lower the barrier to adoption and ensure seamless integration with existing enterprise IT infrastructure, A2A is built upon a foundation of well-established and widely understood web standards. It leverages HTTP for transport, JSON-RPC 2.0 for structured data exchange, and Server-Sent Events (SSE) for real-time streaming. This reliance on familiar technologies significantly reduces the learning curve for developers and simplifies integration with existing systems.[1]
- **Secure by Default:** Security is a foundational, non-negotiable aspect of the protocol.

A2A incorporates support for enterprise-grade authentication and authorization mechanisms from the ground up. It aligns with standard practices like the OpenAPI specification's security schemes, supporting methods such as JWT, OAuth 2.0, and OpenID Connect (OIDC). This ensures that all agent interactions are secure, auditable, and compliant with enterprise governance policies.[2]

- **Support for Long-Running Tasks:** Many real-world business processes are not instantaneous. A2A is explicitly designed to manage interactions that may take hours, or even days, to complete, particularly those involving human-in-the-loop workflows. The protocol provides robust mechanisms for managing the lifecycle of these long-running tasks, including event-driven status updates, real-time feedback, and asynchronous push notifications, ensuring that client agents can reliably track progress over extended periods.[1]
- **Modality Agnostic:** Recognizing that agent communication transcends simple text, A2A is designed to be modality-agnostic. It natively supports the exchange of various content types, including structured JSON data, files, audio streams, and video streams. Furthermore, it allows for the negotiation of rich user interface (UI) capabilities, such as interactive forms and iframes, enabling complex and dynamic user experiences orchestrated by collaborating agents.[1]

2. The A2A Architectural Blueprint

The A2A protocol is architected around a clear and familiar client-server model, defining a structured workflow for all interactions. This blueprint not only specifies the roles of the participating agents but also clarifies its relationship with complementary protocols like the Model Context Protocol (MCP), providing a comprehensive framework for building robust, interoperable multi-agent systems.

The Client-Server Model: Core Actors

Every A2A interaction involves three primary participants, each with a distinct role [3]:

- **User:** The entity, which can be a human operator or an automated service, that initiates a request or defines a goal. The user's intent is the ultimate driver of the multi-agent workflow.
- **A2A Client (Client Agent):** An AI agent, application, or service that acts on behalf of the user. The client is responsible for understanding the user's goal, discovering appropriate remote agents, formulating tasks, and initiating communication using the A2A protocol.

- **A2A Server (Remote Agent):** An AI agent or agentic system that exposes an A2A-compliant HTTP endpoint. It receives and processes tasks from clients, returning results, artifacts, or status updates.

A critical architectural concept within this model is that the remote agent operates as an **"opaque"** system from the client's perspective.[9] The client interacts with the remote agent based on a publicly advertised contract (the Agent Card) without any knowledge of its internal workings—its prompts, proprietary logic, memory, or specific tool implementations remain hidden.[2] This principle of opacity is not merely a technical detail; it is a strategic enabler for a secure, B2B agent economy. It allows enterprises to expose the capabilities of their proprietary AI agents as a service without revealing valuable intellectual property. For companies like Adobe, Salesforce, and Box, this assurance is a prerequisite for participation in an open agent ecosystem, as it allows their agents to collaborate while protecting the "secret sauce" that provides their competitive advantage.[1]

The Three-Phase Interaction Workflow

The lifecycle of a typical A2A interaction follows a logical, three-phase workflow, providing a predictable pattern for developers to implement [2]:

1. **Discovery:** The process begins when a client agent needs to perform a task that is outside its own capabilities. It must first discover a suitable remote agent. This is achieved by finding and parsing the remote agent's "Agent Card," a public metadata document that describes its identity, skills, and communication protocols.
2. **Authentication:** Once a suitable remote agent is identified, the client agent must authenticate itself according to the security schemes specified in the Agent Card. A2A supports standard, enterprise-grade authentication methods to ensure that only authorized clients can interact with a remote agent.
3. **Communication:** After successful authentication, the client agent can begin communicating with the remote agent. This involves sending structured messages to initiate tasks, receiving status updates, and ultimately obtaining the final results or artifacts. Communication can occur through several patterns, including simple request/response, real-time streaming, or asynchronous push notifications.

The Complementary Role of the Model Context Protocol (MCP)

To design effective multi-agent architectures, it is essential to understand the distinction and

synergy between A2A and the Model Context Protocol (MCP). These two open standards are designed to be complementary, addressing different aspects of the agentic ecosystem.[1]

- **A2A enables Horizontal Integration (Agent-to-Agent):** A2A is the protocol for communication *between* autonomous agent peers. It acts as the "public internet" for agents, facilitating collaboration, delegation, and coordination across organizational and framework boundaries. The analogy is a project team where different members (agents) work together to achieve a common goal.[7]
- **MCP enables Vertical Integration (Agent-to-Tool):** MCP is the protocol for communication between an agent and its *own* internal or external tools, such as APIs, databases, or predefined functions. It standardizes how an agent accesses the resources it needs to perform its individual tasks. The analogy is a worker (agent) using their personal toolkit to do their job.[7]

A practical example illustrates this relationship clearly: a retail inventory agent might use MCP to connect to and query an internal product database (a tool). Upon discovering that an item is low in stock, the inventory agent would then use A2A to communicate with an external supplier's agent (a peer) to initiate a reorder process.[2]

Table: A2A vs. MCP: A Comparative Analysis

This table provides a clear, at-a-glance reference for architects to understand the distinct roles of A2A and MCP in a modern agentic architecture.

Feature	A2A Protocol	Model Context Protocol (MCP)
Primary Purpose	Communication and collaboration *between* autonomous AI agents.	Communication *between* an AI agent and its tools (APIs, data sources, functions).
Interaction Type	Horizontal (Peer-to-Peer).	Vertical (Agent-to-Resource).
Analogy	Agents forming a project team to collaborate on a complex task.[7]	An agent utilizing its individual toolkit to perform its part of a task.[7]

Key Components	AgentCard, Task, Message, Part, Artifact.[2]	Tools, Resources, Prompts (structured as JSON-RPC messages).[7]
Scope	External, cross-boundary communication between potentially opaque agents.	Internal, within-agent communication to access necessary capabilities.
When to Use	When one agent needs to delegate a task to, or collaborate with, another agent.	When an agent needs to call an API, query a database, or execute a local function.

3. The Language of Agents: Core Data Objects

The A2A protocol defines a set of standardized data objects that form the vocabulary for all inter-agent communication. These structures provide the necessary framework for discovery, task management, and the exchange of rich, multi-modal content.

The AgentCard: Public Identity for Discovery and Negotiation

The AgentCard is the cornerstone of the A2A protocol, serving as the primary mechanism for agent discovery and capability negotiation.[13] It is a machine-readable JSON manifest that functions as an agent's public "business card," allowing other agents to dynamically discover its existence and understand how to interact with it.[2]

- **Purpose:** The AgentCard enables a client agent to identify the best remote agent for a given task and provides all the necessary information to establish a secure connection and initiate communication.[1]
- **Location:** For public discovery, it is recommended that the AgentCard be hosted at a well-known, standardized path on the agent's domain: /.well-known/agent.json.[4] This allows for predictable discovery via DNS.
- **Key Fields:** A valid AgentCard contains several critical fields that describe the agent's identity and capabilities [10]:
  - name, description, version: Basic identification metadata for the agent.
  - url: The base endpoint URL where the agent's A2A service is hosted.

- capabilities: A set of boolean flags indicating support for advanced protocol features, such as streaming (via SSE) and pushNotifications (via webhooks).
- authentication: An object specifying the supported security schemes (e.g., "bearer", "oauth2"), aligning with OpenAPI standards.
- skills: An array of AgentSkill objects, each detailing a specific function the agent can perform. Each skill includes a unique id, a human-readable name and description, relevant tags, and illustrative examples.
- defaultInputModes and defaultOutputModes: The default communication modalities the agent supports (e.g., "text", "data", "audio").

The Task: The Stateful Unit of Work

Unlike stateless REST API calls, A2A communication is architected around the concept of a stateful Task object. This is a fundamental design choice that enables the management of complex, long-running, and multi-turn interactions.[13]

- **Purpose:** A Task represents a complete unit of work initiated by a client agent. It is assigned a unique ID by the server and encapsulates the entire history of the interaction, its current status, and all generated outputs (artifacts). This statefulness is what allows agents to collaborate on processes that may take significant time to complete.[4]
- **Lifecycle:** Each Task progresses through a well-defined lifecycle, represented by a series of states. This state machine provides a clear and robust framework for clients to track progress and handle various outcomes, including errors and requests for additional information.

Table: The A2A Task Lifecycle

This table details the states of the A2A Task lifecycle, providing developers with a clear model for implementing task management logic.

State	Description	Possible Transitions	Triggering Event/Method
submitted	The task has been received by the server but	working, failed	Initial message/send or message/stream

	processing has not yet begun.		call.
working	The server is actively processing the task.	input-required, completed, failed, canceled	Server begins execution of the task logic.
input-required	The server has paused the task and requires additional information from the client to proceed.	working	The remote agent determines it needs more context or user input.
completed	The task has been successfully completed, and all final artifacts have been generated. This is a terminal state.	None	The remote agent's logic finishes successfully.
failed	The task could not be completed due to an error. This is a terminal state.	None	An unrecoverable error occurred during processing.
canceled	The task was canceled by the client before completion. This is a terminal state.	None	Client successfully calls the tasks/cancel method.

Messages, Parts, and Artifacts: Structuring Multi-Modal Communication

The actual content exchanged between agents is structured using a hierarchy of three

objects: Message, Part, and Artifact.

- **Message:** A Message represents a single turn of communication within a Task. Each message has a role (either "user" for client-sent messages or "agent" for server-sent messages) and contains one or more Part objects that hold the content.[13]
- **Part:** The Part is the fundamental, granular container for content. The protocol's modality-agnostic nature stems from its support for different part types [10]:
  - TextPart: Contains plain textual content.
  - FilePart: Represents a file, which can be transmitted either inline as a base64-encoded string or referenced via a URI. It includes metadata such as the filename and MIME type.
  - DataPart: Carries structured data in the form of a JSON object, ideal for passing parameters, form data, or any other machine-readable information.
- **Artifact:** An Artifact is a tangible, self-contained output generated by the remote agent as a result of a Task (e.g., a generated PDF report, a spreadsheet, or an image). Like a Message, an Artifact is also composed of one or more Part objects and represents the final deliverable of the agent's work.[1]

Part II: Building an A2A-Compliant Agent with the Agent Development Kit (ADK)

This section provides a practical, hands-on guide to building A2A-compliant agents using Google's Agent Development Kit (ADK). The ADK is an open-source Python toolkit designed to streamline the development, evaluation, and deployment of sophisticated AI agents, and it includes native support for the A2A protocol.[15] The following steps will walk through the process of both exposing an agent as an A2A server and consuming a remote agent as an A2A client.

4. Setting Up the Development Environment

Before beginning development, the appropriate tools and libraries must be installed. A properly configured environment is essential for building and debugging A2A-enabled agents efficiently.

Installation

The primary requirement is the google-adk Python library. To include the necessary components for A2A communication, it should be installed with the [a2a] extra. Alternatively, for projects that only require A2A functionality without the full ADK framework, the standalone a2a-sdk can be used.[9]

To install the ADK with A2A support, execute the following command:

```
Shell

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

For the standalone SDK:

```
Shell

pip install a2a-sdk
```

It is also recommended to use a virtual environment to manage project dependencies and avoid conflicts.

Essential Tooling

Effective development requires robust debugging tools. The A2A ecosystem provides the **A2A Inspector**, a powerful web-based utility designed to facilitate the development and testing of A2A agents.[15] This tool is invaluable for rapid iteration and troubleshooting, offering several key features [27]:

- **Agent Card Viewer:** Fetches and validates an agent's public AgentCard, allowing developers to verify that their agent is correctly advertising its capabilities.
- **Live Chat Interface:** Provides a simple UI to send messages directly to a deployed agent for immediate, interactive testing of its responses.

- **Debug Console:** Displays the raw JSON-RPC messages being exchanged between the inspector and the agent, offering deep visibility into the protocol-level communication for advanced debugging.

5. Exposing an Agent: The A2A Server Implementation

Exposing an existing agent to the outside world via A2A involves creating a server that listens for incoming requests, translates them into actions for the agent to perform, and returns the results according to the protocol's specifications. This process can be broken down into three main steps: defining the public interface, implementing the core logic handler, and launching the server. This guide follows the pattern demonstrated in official ADK documentation and codelabs.[17]

Step 1: Defining the Public Interface with AgentSkill and AgentCard

The first step is to create the agent's public identity. This is achieved by defining its capabilities as AgentSkill objects and then packaging them into a single AgentCard.[20]

An AgentSkill is a Python object that declares a specific function the agent can perform. It includes a unique ID, a human-readable name and description, and examples of how to use it.

Python

```
# In __main__.py or a similar entrypoint file
from a2a.types import AgentSkill, AgentCard, AgentCapabilities

# Define the skills for a currency conversion agent
currency_skill = AgentSkill(
    id='get_exchange_rate',
    name='Get Currency Exchange Rate',
    description='Retrieves the exchange rate between two currencies.',
    tags=['finance', 'currency', 'exchange'],
    examples=
)
```

```
# Define the AgentCard, which acts as the agent's public metadata
agent_card = AgentCard(
    name='CurrencyConverterAgent',
    description='An agent that provides real-time currency exchange rates.',
    url='http://localhost:8080/',  # The URL where this agent will be hosted
    version='1.0.0',
    capabilities=AgentCapabilities(
        streaming=True,
        pushNotifications=False
    ),
    defaultInputModes=['text'],
    defaultOutputModes=['text'],
    skills=[currency_skill]
)
```

Step 2: Implementing the Core Logic with AgentExecutor

The AgentExecutor class is the critical architectural component that serves as the bridge between the A2A protocol's standardized task management and the agent's specific, internal logic.[17] This class implements a required interface, primarily the

execute method, which is called by the A2A server framework whenever a new task is received.

The AgentExecutor is an implementation of the **Adapter** design pattern. It adapts the generic A2A Task object into a format that the agent's native framework (be it ADK, CrewAI, or custom logic) can understand. It then invokes the agent's logic and translates the output back into standard A2A Artifacts and TaskState updates. This decoupling is what enables A2A's promise of framework interoperability, as the core protocol remains unaware of the agent's internal implementation details.

The execute method receives the RequestContext (containing the incoming request), a TaskUpdater (to change the task's state), and an EventQueue (to stream back results).

Python

```python
# In task_manager.py
import asyncio
from a2a.server.agent_execution import AgentExecutor, RequestContext
from a2a.server.events import EventQueue
from a2a.server.tasks import TaskUpdater
from a2a.types import TaskState, TextPart
from a2a.utils import new_agent_text_message

# This is a placeholder for your agent's actual logic
async def get_exchange_rate_logic(base_currency: str, target_currency: str) -> str:
    # In a real implementation, this would call an API
    await asyncio.sleep(2) # Simulate network latency
    if base_currency == 'USD' and target_currency == 'EUR':
        return f'The exchange rate from {base_currency} to {target_currency} is 0.92.'
    return 'Sorry, I could not find the exchange rate for the specified currencies.'

class CurrencyAgentExecutor(AgentExecutor):
    async def execute(
        self,
        context: RequestContext,
        task_updater: TaskUpdater,
        event_queue: EventQueue
    ):
        # 1. Acknowledge the task and set its state to 'working'
        await task_updater.update_status(TaskState.WORKING)

        # 2. Extract user input from the request
        # For simplicity, we assume the first text part is the user query
        user_query = ""
        if context.task.messages and context.task.messages.parts:
            for part in context.task.messages.parts:
                if isinstance(part, TextPart):
                    user_query = part.text
                    break

        # In a real agent, you would use an LLM to parse the query
        # Here, we'll use simple string matching for demonstration
        base_currency = "USD"
        target_currency = "EUR"

        # 3. Invoke the agent's core logic
        result_text = await get_exchange_rate_logic(base_currency, target_currency)

        # 4. Stream the result back to the client via the EventQueue
```

```
        # The EventQueue is used for streaming responses (SSE)
        response_message = new_agent_text_message(result_text)
        await event_queue.put(response_message)

        # 5. Set the final task state to 'completed'
        await task_updater.update_status(TaskState.COMPLETED)

    async def cancel(self, task_id: str):
        # Implement logic to handle task cancellation if applicable
        print(f"Task {task_id} cancellation requested.")
```

Step 3: Assembling and Launching the A2A Server

With the AgentCard and AgentExecutor defined, the final step is to assemble them into a web application and launch it. The a2a-sdk provides helper classes to make this straightforward. For example, the A2AStarletteApplication can be used with ASGI servers like Uvicorn.[17]

Python

```
# In __main__.py
import uvicorn
from a2a.server.app import A2AStarletteApplication
from.task_manager import CurrencyAgentExecutor
# Assuming agent_card is defined in this file as shown in Step 1

# Create the A2A application instance
app = A2AStarletteApplication(
    agent_card=agent_card,
    agent_executor=CurrencyAgentExecutor()
)

# Run the server using Uvicorn
if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

To run this server, save the files and execute python -m your_package_name.__main__ from the terminal. The A2A agent will now be running on http://localhost:8080, with its AgentCard

discoverable at http://localhost:8080/.well-known/agent.json.

Handling Communication Patterns

The AgentExecutor implementation determines which communication patterns the agent supports.

- **Request/Response (Polling):** This is the default behavior. A client calls the message/send RPC method. The server immediately returns a Task object with a status of submitted or working. The client must then periodically call the tasks/get method with the task ID to poll for status updates and retrieve the final artifact once the task is completed.[13]
- **Streaming with Server-Sent Events (SSE):** To support real-time updates, the client calls the message/stream method. This establishes a persistent HTTP connection over which the server can push events. In the AgentExecutor, any item put into the EventQueue is sent to the client as an SSE event. This is ideal for streaming intermediate thoughts from an LLM or providing progress updates for a multi-step process.[2]
- **Asynchronous Push Notifications:** For very long-running tasks where maintaining an open connection is impractical (e.g., tasks that take hours or require human intervention), A2A supports push notifications. The client provides a secure webhook URL to the server by calling the tasks/pushNotificationConfig/set method. The server can then send an HTTP POST request to this webhook to notify the client of significant state changes (e.g., when the task is complete). The server must advertise the pushNotifications capability in its AgentCard to support this pattern.[2]

6. Consuming an Agent: The A2A Client Implementation

Consuming a remote A2A agent involves discovering its capabilities via its AgentCard and then using an A2A client library to make structured JSON-RPC calls to its endpoint. This process allows one agent to programmatically delegate tasks to another.

Discovering Remote Agents

The first step for any client is discovery. This typically involves making an HTTP GET request to

the remote agent's well-known AgentCard URL and parsing the resulting JSON. The a2a-sdk provides helpers like the A2ACardResolver to simplify this process.[17]

Python

```
import httpx

async def discover_agent(agent_url: str) -> dict:
    """Fetches and returns the AgentCard from a remote agent."""
    card_url = f"{agent_url.rstrip('/')}/.well-known/agent.json"
    async with httpx.AsyncClient() as client:
        response = await client.get(card_url)
        response.raise_for_status()  # Raise an exception for bad status codes
        return response.json()

# Example usage:
# remote_agent_card = await discover_agent("http://localhost:8080")
# print(remote_agent_card)
```

Initiating and Managing Tasks

Once the AgentCard is fetched, the client knows the agent's endpoint URL and capabilities. It can then use a client library to make JSON-RPC calls to the server's methods. The core methods a client will use are message/send (for request/response), message/stream (for streaming), tasks/get (for polling), and tasks/cancel (for cancellation).[24]

Below is a simplified example of a Python client sending a task and polling for the result.

Python

```
import httpx
import uuid
import asyncio
```

```python
async def send_a2a_task(agent_url: str, query: str):
    """Sends a task to an A2A agent and polls for the result."""
    session_id = str(uuid.uuid4())

    async with httpx.AsyncClient(timeout=30.0) as client:
        # 1. Send the initial message to create a task
        send_payload = {
            "jsonrpc": "2.0",
            "method": "message/send",
            "params": {
                "sessionId": session_id,
                "message": {
                    "role": "user",
                    "parts": [{"type": "text", "text": query}]
                }
            },
            "id": str(uuid.uuid4())
        }

        response = await client.post(agent_url, json=send_payload)
        response.raise_for_status()
        task_info = response.json()['result']
        task_id = task_info['id']
        print(f"Task created with ID: {task_id}")

        # 2. Poll the tasks/get endpoint until the task is complete
        while True:
            get_payload = {
                "jsonrpc": "2.0",
                "method": "tasks/get",
                "params": {"id": task_id},
                "id": str(uuid.uuid4())
            }
            response = await client.post(agent_url, json=get_payload)
            response.raise_for_status()
            task_status = response.json()['result']

            print(f"Current task status: {task_status['status']['state']}")

            if task_status['status']['state'] in ["completed", "failed"]:
                print("Final Artifacts:", task_status.get('artifacts',))
                break
```

```
        await asyncio.sleep(1) # Wait before polling again

# Example usage:
# await send_a2a_task("http://localhost:8080", "What is the exchange rate from USD to EUR?")
```

Orchestration with ADK

A powerful pattern is to create a "root" or "orchestrator" agent using the ADK that consumes other specialized, remote A2A agents as if they were local tools. This demonstrates the full potential of A2A for building complex, distributed multi-agent systems.[26] The ADK provides abstractions like the

RemoteA2aAgent class that simplify this integration, handling the underlying JSON-RPC communication and allowing the orchestrator to interact with the remote agent through a clean, tool-like interface.

Part III: Ecosystem Integration and Framework-Specific Implementations

A key promise of the A2A protocol is interoperability—the ability to connect agents built with different frameworks. This section provides practical guidance and architectural patterns for integrating A2A capabilities within popular agentic frameworks beyond Google's ADK, such as LangChain, CrewAI, and AutoGen. The maturity and implementation complexity of A2A support vary significantly across these ecosystems, a critical factor for architects to consider when selecting a technology stack.

7. Integrating with LangChain and LangGraph

The LangChain ecosystem, particularly with the advent of LangGraph, has embraced A2A, offering increasingly native support. This makes it one of the most straightforward frameworks to integrate into a multi-agent A2A network.

Bidirectional Integration

Third-party libraries and emerging native features provide bidirectional integration pathways, allowing developers to both expose LangChain components as A2A services and consume external A2A agents as tools within LangChain.[30] This flexibility is key to creating hybrid systems that leverage the strengths of the vast LangChain tool ecosystem while participating in the broader A2A network.

Exposing a LangGraph Assistant

LangGraph Platform has introduced first-class, native support for A2A, significantly lowering the barrier to entry for LangGraph developers. A deployed LangGraph assistant automatically exposes two key features [31]:

1. **An A2A Endpoint:** A standardized endpoint is available at /a2a/{assistant_id}, ready to receive A2A-compliant JSON-RPC requests.
2. **Automatic Agent Card Discovery:** The platform automatically generates and serves an AgentCard for each assistant at /.well-known/agent-card.json?assistant_id={assistant_id}. This card describes the assistant's capabilities, allowing for seamless discovery by other A2A clients.

To create an A2A-compatible agent in LangGraph, the primary requirement is that the agent's state graph must be message-based, typically using a messages key in its state definition to handle the conversational history. The following is a conceptual example of an A2A-compatible LangGraph agent [31]:

Python

```
from typing import TypedDict, Annotated
from langgraph.graph import StateGraph, END
from langgraph.graph.message import add_messages

# 1. Define the message-based state for A2A compatibility
class State(TypedDict):
```

```
    messages: Annotated[list, add_messages]

# 2. Define the nodes in the graph
def call_model(state: State):
    # This node would contain the core LLM call logic
    # It processes the input messages from the state
    messages = state['messages']
    #... LLM logic here...
    response = "This is a response from the LangGraph agent."
    return {"messages": [("ai", response)]}

# 3. Build the graph
graph_builder = StateGraph(State)
graph_builder.add_node("call_model", call_model)
graph_builder.set_entry_point("call_model")
graph_builder.add_edge("call_model", END)

graph = graph_builder.compile()

# When this graph is deployed via LangGraph Server, it will be accessible via A2A.
```

Consuming an A2A Agent in LangChain

To consume an external A2A agent within a standard LangChain agent, the most common pattern is to wrap the remote A2A service as a LangChain Tool. This involves creating a custom tool that handles the A2A client-side communication (discovery, task creation, and polling) and exposes it as a simple function that the LangChain agent can invoke.[30]

Python

```
from langchain.tools import BaseTool
from typing import Type
from pydantic import BaseModel, Field
# Assume the 'send_a2a_task' client function from the previous section exists

class A2AInput(BaseModel):
    query: str = Field(description="The query to send to the remote A2A agent.")
```

```
class RemoteA2ATool(BaseTool):
    name = "remote_currency_agent"
    description = "Useful for getting currency exchange rates."
    args_schema: Type = A2AInput

    def _run(self, query: str) -> str:
        # In a real implementation, this should be async
        # For simplicity, this is a synchronous wrapper
        import asyncio
        # This is a simplified call; a real implementation would parse the final artifact
        result = asyncio.run(send_a2a_task("http://localhost:8080", query))
        return str(result)
```

This RemoteA2ATool can then be included in the list of tools provided to a LangChain agent, allowing the agent's LLM to decide when to delegate tasks to the remote A2A service.

8. Integrating with CrewAI

CrewAI, a popular framework for orchestrating role-playing autonomous agents, does not currently offer native, out-of-the-box A2A support. However, its architecture is well-suited for integration using the manual "wrapper" or "adapter" pattern, where a CrewAI Crew is exposed as an A2A server.[5]

Architectural Pattern

The standard approach is to build a standard A2A server (as detailed in Part II) where the AgentExecutor class acts as the integration point. The AgentExecutor's execute method is responsible for taking the incoming A2A Task, extracting the user's query, initializing a CrewAI Crew with the appropriate agents and tasks, and running the crew's process using the crew.kickoff() method. The final result from the crew is then formatted into an A2A Artifact and sent back to the client.[5]

Implementation Example

The following code provides a conceptual implementation of an AgentExecutor that wraps a CrewAI crew.

Python

```python
# In a CrewAI project's task_manager.py
from a2a.server.agent_execution import AgentExecutor, RequestContext
from a2a.server.events import EventQueue
from a2a.server.tasks import TaskUpdater
from a2a.types import TaskState, TextPart
from a2a.utils import new_agent_text_message

# Import your CrewAI components
from crewai import Agent, Task as CrewTask, Crew

# Assume you have defined your CrewAI agents (e.g., researcher, writer)
# and a function to create the crew
def create_research_crew(topic: str) -> Crew:
    researcher = Agent(...)
    writer = Agent(...)
    research_task = CrewTask(description=f"Research the topic: {topic}", agent=researcher)
    write_task = CrewTask(description="Write a summary of the research.", agent=writer)

    return Crew(
        agents=[researcher, writer],
        tasks=[research_task, write_task],
        verbose=2
    )

class CrewAIExecutor(AgentExecutor):
    async def execute(
        self,
        context: RequestContext,
        task_updater: TaskUpdater,
        event_queue: EventQueue
    ):
        await task_updater.update_status(TaskState.WORKING)

        # Extract the topic from the A2A request
```

```
        topic = ""
        if context.task.messages and context.task.messages.parts:
            for part in context.task.messages.parts:
                if isinstance(part, TextPart):
                    topic = part.text
                    break

        if not topic:
            await task_updater.update_status(TaskState.FAILED)
            return

        # Initialize and run the CrewAI crew
        research_crew = create_research_crew(topic)
        # Note: crew.kickoff() is synchronous, so run it in an executor
        # for a non-blocking async implementation.
        result = research_crew.kickoff()

        # Format the result as an A2A message and send it back
        response_message = new_agent_text_message(result)
        await event_queue.put(response_message)

        await task_updater.update_status(TaskState.COMPLETED)
```

Case Study: Hybrid ADK and CrewAI System

A powerful demonstration of this pattern is a hybrid multi-agent system where an ADK-based orchestrator agent delegates tasks to a remote CrewAI-based research agent via A2A.[5] In this architecture, the orchestrator acts as the A2A client, handling user interaction and high-level planning. When it requires in-depth web research, it discovers and sends a task to the specialized CrewAI agent, which is wrapped in an A2A server. This microservices-based approach allows each component to be developed, scaled, and maintained independently, showcasing the flexibility unlocked by A2A.[5]

9. Integrating with AutoGen and the Broader Ecosystem

For other frameworks like Microsoft's AutoGen, A2A integration is more conceptual at present,

as there are fewer established libraries or official examples. However, the same architectural wrapper pattern applies.

Conceptual Approach for AutoGen

AutoGen's architecture is centered around "conversable agents" that interact within a GroupChatManager or a similar orchestration structure.[32] To integrate AutoGen with A2A, one would create an A2A server where the

AgentExecutor acts as a gateway to an AutoGen conversation.[4]

When an A2A Task is received, the AgentExecutor would:

1. Instantiate the necessary AutoGen agents (e.g., a UserProxyAgent and an AssistantAgent).
2. Initiate a chat within the AutoGen system, using the input from the A2A task as the initial prompt for the UserProxyAgent.
3. Capture the final response from the conversation.
4. Format this response into an A2A Artifact and return it to the client.

This approach effectively exposes a complex, multi-agent AutoGen conversation as a single, callable A2A skill.

Microsoft Ecosystem and Industry Adoption

The viability of A2A is significantly bolstered by its broad industry adoption beyond the Google ecosystem. Microsoft has announced its commitment to supporting A2A in its flagship AI platforms, including Azure AI Foundry and Copilot Studio.[4] This support will enable agents built with Microsoft tools, like Semantic Kernel, to interoperate with the wider A2A network. A sample implementation demonstrating A2A collaboration between two local agents using Semantic Kernel is already available, signaling a strong commitment to the protocol.[33] This cross-industry alignment is a powerful indicator of A2A's trajectory toward becoming a true industry standard for agent interoperability.

Part IV: Advanced Topics and Production

Considerations

Moving an A2A-compliant multi-agent system from a prototype to a production environment requires careful consideration of non-functional requirements, particularly security, governance, and deployment strategy. This section addresses the advanced topics and best practices essential for building and operating robust, scalable, and secure A2A applications in the real world.

10. Enterprise-Grade Security and Governance

Security is a paramount concern in any distributed system, and A2A was designed with enterprise requirements in mind. The protocol provides a strong foundation for secure communication, but developers and architects must correctly implement and manage these features.

Authentication and Authorization

A2A separates the authentication layer from the core protocol messaging, leveraging standard web security practices.

- **Declaring Security Schemes:** The AgentCard is the authoritative source for an agent's security requirements. The authentication object within the card explicitly declares the supported schemes, such as "bearer" for OAuth 2.0 or OIDC tokens, or other schemes aligned with the OpenAPI specification.[2] A client agent must parse this section of the card to understand how to authenticate.
- **Passing Credentials:** Credentials are not passed within the A2A JSON-RPC message body. Instead, they are transmitted using standard HTTP headers, most commonly the Authorization header (e.g., Authorization: Bearer <your-jwt-token>). This separation of concerns ensures that the A2A protocol itself remains focused on agent collaboration logic while leveraging the mature and well-understood security mechanisms of HTTP.[6]
- **Progressive Capability Disclosure:** A2A supports a mechanism for progressively disclosing an agent's capabilities based on authentication level. A public, unauthenticated AgentCard might only list a basic set of skills. After a client successfully authenticates, it can call the agent/getAuthenticatedExtendedCard RPC method to retrieve an extended card that may reveal more powerful or sensitive skills. This allows for

fine-grained access control.[11]

Transport and Discovery Security

Securing the communication channel and the discovery process is as critical as authenticating the agents themselves.

- **Transport Layer Security (TLS):** All A2A communication in a production environment **MUST** occur over HTTPS using a modern and secure version of TLS. This is a non-negotiable requirement to ensure the confidentiality and integrity of data in transit, protecting against eavesdropping and man-in-the-middle attacks.[10]
- **Securing Agent Card Discovery:** A potential vulnerability in the A2A ecosystem is the tampering of an AgentCard during discovery. A malicious actor could intercept the request for an AgentCard and return a modified version that points to a malicious endpoint. To mitigate this, a proposed best practice is the use of **signed agent cards**. The agent provider can sign the AgentCard JSON object using a JSON Web Signature (JWS). The provider then publishes their public key, which clients can use to verify the signature of any AgentCard they retrieve. If the signature is invalid, the client knows the card has been tampered with and can reject the connection.[15]

The Unsolved Problem of Discovery Governance

A notable omission from the current A2A protocol specification is the definition of a standardized, global, or trusted public registry for discovering agents.[34] The protocol specifies

how an agent should describe itself (AgentCard) but not *where* clients should look to find these cards at scale. This has several implications for the ecosystem:

- **Private Registries:** In the near term, enterprises will likely build their own internal, curated registries or catalogs to manage the AgentCards for their own and trusted third-party agents.[7]
- **Marketplaces:** Platforms like Google Cloud's AI Agent Marketplace may serve as trusted venues for discovering and consuming partner-built A2A agents.[15]
- **Decentralized Approaches:** Emerging research is exploring the use of decentralized technologies, such as blockchain and distributed ledger technology (DLT), to create verifiable, tamper-proof, on-chain registries for AgentCards. This approach could provide a trustless and decentralized solution to the discovery problem in the long run.[34]

Auditing and Observability

The design of the A2A protocol, particularly its use of stateful Task objects, inherently supports robust auditing and observability. Every task is assigned a unique taskId, and related tasks can be grouped using an optional sessionId. By logging these identifiers at each step of a multi-agent workflow, organizations can trace the full provenance of a complex operation across multiple agents and systems. This provides a comprehensive audit trail for compliance, debugging, and performance analysis.[10]

11. Deployment Patterns and Best Practices

Deploying A2A agents into a production environment involves packaging the application and choosing a hosting platform that meets scalability, security, and manageability requirements.

Containerization

The standard best practice for deploying modern applications, including A2A agents, is containerization using Docker. Packaging the agent, its dependencies, and the A2A server into a container image provides portability and consistency across different environments (development, staging, production).[5] A typical

Dockerfile for a Python-based A2A agent would copy the application code, install dependencies from a requirements.txt file, expose the required port, and specify the command to start the web server (e.g., Uvicorn).

Serverless Deployment

Serverless compute platforms like Google Cloud Run are an excellent fit for hosting A2A agents. They offer several key advantages [15]:

- **Automatic Scaling:** Cloud Run automatically scales the number of container instances

up or down (even to zero) based on incoming traffic, ensuring high availability without the need for manual infrastructure management.
- **Managed HTTPS Endpoints:** When a service is deployed to Cloud Run, it is automatically assigned a secure, publicly accessible HTTPS URL with a managed TLS certificate. This simplifies the process of meeting the protocol's transport security requirements.
- **Simplified Configuration:** Environment variables, such as the public URL needed for the AgentCard's HOST_OVERRIDE, can be easily configured through the deployment process.

Managed Deployment

For organizations seeking a more specialized and optimized environment, managed platforms like Vertex AI Agent Engine are emerging as the preferred choice. These platforms are specifically designed for hosting, scaling, and managing AI agents. As A2A support becomes natively integrated into these services, they will offer a turnkey solution for deploying production-ready, Google-scale A2A agents with minimal operational overhead.[15]

12. Real-World Applications and Future Directions

The A2A protocol is not merely a theoretical standard; it is being actively adopted by a growing ecosystem of over 150 organizations to solve real-world business problems across various industries.[15] These applications demonstrate the tangible value of enabling seamless, cross-platform agent collaboration.

Industry Case Studies

- **Supply Chain and E-commerce:** A common and powerful use case involves an inventory management agent that, upon detecting low stock, uses A2A to communicate directly with external supplier agents to automate the procurement process.[2] Food industry leaders Tyson Foods and Gordon Food Service are actively building collaborative A2A systems to create a real-time channel for sharing product data, leads, and orders, reducing friction in the food supply chain.[15]
- **Human Resources and Recruitment:** A2A can significantly streamline the hiring process. A hiring manager's primary agent can act as an orchestrator, using A2A to

delegate specific tasks to specialized agents for candidate sourcing, interview scheduling, and background checks, creating a unified and automated workflow from a single user request.[1]

- **Finance and Customer Service:** In the enterprise software space, A2A enables powerful cross-system automation. Salesforce, for instance, is leveraging A2A to allow its Agentforce agents to resolve complex customer requests that span multiple systems. A single support ticket can trigger a collaborative workflow between agents managing the CRM, technical support, and billing systems, providing a seamless resolution without manual handoffs.[36] Similarly, consulting firms like Deloitte are using A2A to build solutions that connect disparate enterprise systems like CRM and ServiceNow, cutting unproductive agent time by up to 25%.[38]
- **Content Management and Digital Experiences:** Box is integrating its intelligent content management platform with the A2A protocol. This allows Box's AI agents, which can extract metadata and insights from enterprise content, to interoperate with the broader Google Agentspace ecosystem. This enables complex workflows where, for example, insights extracted from a contract in Box can automatically trigger an update in a Salesforce CRM record via A2A.[16] Adobe is also leveraging A2A to make its distributed agents interoperable, enabling them to collaborate on optimizing content creation workflows and automating multi-system data integrations.[15]

The Emerging Payments Ecosystem: AP2 and A2A x402

The long-term vision for A2A extends beyond task collaboration to encompass a fully autonomous agent economy. This is evidenced by the development of the **Agent Payments Protocol (AP2)**, an open protocol designed as an extension to A2A to standardize secure, agent-led financial transactions.[40]

Real-world business processes frequently culminate in a financial transaction. However, a standard A2A message lacks the necessary security primitives—such as non-repudiation and verifiable authorization—required for commerce. AP2 was created to fill this gap. It introduces the concept of **Mandates**, which are tamper-proof, cryptographically-signed digital contracts that serve as verifiable proof of a user's instructions and authorization for a payment.[40] AP2 addresses the critical requirements of Authorization, Authenticity, and Accountability, providing a trusted framework for agents to transact on behalf of users.[40]

Furthermore, to support the web3 ecosystem, the A2A x402 extension has been launched. This extension builds on AP2 to provide a production-ready solution for agent-based crypto payments, leveraging the HTTP 402 "Payment Required" status code.[34]

The development of AP2 reveals a clear strategic roadmap: first, enable agents to communicate and collaborate (A2A), and second, enable them to conduct commerce (AP2). This positions A2A as the foundational transport layer for a future digital economy where autonomous agents can discover, negotiate, collaborate, and transact with one another securely and at scale.

The A2A Roadmap and Community

As an open-source project, the A2A protocol is continuously evolving through community contributions. The official roadmap includes several planned enhancements aimed at increasing the protocol's power and flexibility, such as [9]:

- A QuerySkill() method for dynamically checking an agent's support for unanticipated skills.
- Support for dynamic negotiation of UI and modalities within an active task (e.g., an agent adding audio/video capabilities mid-conversation).
- Improvements to streaming reliability and push notification mechanisms.

Developers and architects are encouraged to explore the official documentation, engage with the project on GitHub, and contribute to the evolution of this foundational protocol for the future of AI.[3]

Conclusion

The Agent2Agent (A2A) protocol represents a critical and timely solution to the growing problem of fragmentation in the AI agent ecosystem. By establishing an open, secure, and vendor-neutral standard for inter-agent communication, A2A provides the foundational "connective tissue" required to move from isolated, single-purpose agents to complex, collaborative multi-agent systems. Its design, rooted in established web standards and guided by principles of security, flexibility, and agent autonomy, makes it a pragmatic and powerful choice for enterprise adoption.

The architectural blueprint of A2A, centered on the AgentCard for discovery and the stateful Task for managing long-running interactions, offers a robust framework for building reliable and observable distributed systems. The principle of agent opacity is a key strategic enabler, fostering a trusted environment where organizations can expose their proprietary AI capabilities as services without compromising intellectual property.

For developers and architects, the path to implementation is becoming increasingly clear. The Google Agent Development Kit (ADK) provides a reference implementation and a powerful toolkit for building A2A-native agents. Concurrently, the rapid integration of A2A support into major agentic frameworks like LangGraph, and the emergence of clear architectural patterns for wrapping frameworks like CrewAI and AutoGen, are paving the way for true interoperability.

The real-world applications already being pioneered by industry leaders in supply chain, finance, HR, and content management validate the protocol's immense potential. More profoundly, the development of extensions like the Agent Payments Protocol (AP2) signals a far-reaching vision: to create not just a network for agent collaboration, but a platform for an autonomous agent economy. As this ecosystem matures, the A2A protocol is poised to become an indispensable piece of the infrastructure for the next generation of artificial intelligence, enabling a future where intelligent systems can work together seamlessly to solve the world's most complex challenges.

Works cited

1. Announcing the Agent2Agent Protocol (A2A) - Google for Developers Blog, accessed September 25, 2025, https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
2. What is A2A protocol (Agent2Agent)? - IBM, accessed September 25, 2025, https://www.ibm.com/think/topics/agent2agent-protocol
3. A2A Protocol Technical Documentation, accessed September 25, 2025, https://agent2agent.info/docs/
4. Orchestrating heterogeneous and distributed multi-agent systems using Agent-to-Agent (A2A) protocol - Fractal Analytics, accessed September 25, 2025, https://fractal.ai/blog/orchestrating-heterogeneous-and-distributed-multi-agent-systems-using-agent-to-agent-a2a-protocol
5. Unlocking Multi-Agent A2A: How to Connect CrewAI and ADK on Google Cloud, accessed September 25, 2025, https://discuss.google.dev/t/unlocking-multi-agent-a2a-how-to-connect-crewai-and-adk-on-google-cloud/265858
6. What Is Agent2Agent Protocol (A2A)? - Solo.io, accessed September 25, 2025, https://www.solo.io/topics/ai-infrastructure/what-is-a2a
7. Agent2Agent Protocol: The ABCs of A2A - Aisera, accessed September 25, 2025, https://aisera.com/blog/a2a-agent2agent-protocol/
8. A2A Protocol, accessed September 25, 2025, https://a2a-protocol.org/
9. a2aproject/A2A: An open protocol enabling communication and interoperability between opaque agentic applications. - GitHub, accessed September 25, 2025, https://github.com/a2aproject/A2A
10. A2A Protocol: An In-Depth Guide. The Need for Agent Interoperability | by Saeed Hajebi, accessed September 25, 2025, https://medium.com/@saeedhajebi/a2a-protocol-an-in-depth-guide-78387f992f59

11. Open Protocols for Agent Interoperability Part 4: Inter-Agent Communication on A2A - AWS, accessed September 25, 2025, https://aws.amazon.com/blogs/opensource/open-protocols-for-agent-interoperability-part-4-inter-agent-communication-on-a2a/
12. Agent2Agent (A2A) – awesome A2A agents, tools, servers & clients, all in one place. - GitHub, accessed September 25, 2025, https://github.com/ai-boost/awesome-a2a
13. Key Concepts - Agent2Agent Protocol (A2A) - Google, accessed September 25, 2025, https://google.github.io/A2A/topics/key-concepts/
14. Core Concepts - A2A Protocol, accessed September 25, 2025, https://a2a-protocol.org/latest/topics/key-concepts/
15. Agent2Agent protocol (A2A) is getting an upgrade | Google Cloud Blog, accessed September 25, 2025, https://cloud.google.com/blog/products/ai-machine-learning/agent2agent-protocol-is-getting-an-upgrade
16. Google Cloud Next 2025: How Box and Google Cloud are transforming enterprise content with AI | Box Blog, accessed September 25, 2025, https://blog.box.com/google-cloud-next-2025-how-box-and-google-cloud-are-transforming-enterprise-content-ai
17. Getting Started with Agent2Agent (A2A) Protocol: A Purchasing Concierge and Remote Seller Agent Interactions on Cloud Run and Agent Engine | Google Codelabs, accessed September 25, 2025, https://codelabs.developers.google.com/intro-a2a-purchasing-concierge
18. Google A2A Protocol Explained: Tutorial, Demo, & How It Works with MCP & AI Agents, accessed September 25, 2025, https://www.youtube.com/watch?v=GUozMSpnmcc
19. What is the Agent2Agent (A2A) Protocol? - Builder.io, accessed September 25, 2025, https://www.builder.io/blog/a2a-protocol
20. Develop an Agent2Agent agent | Generative AI on Vertex AI - Google Cloud, accessed September 25, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/develop/a2a
21. Playing around with A2A — LangGraph & CrewAI | by Heemeng Foo | Sep, 2025 - Medium, accessed September 25, 2025, https://heemeng.medium.com/playing-around-with-a2a-langgraph-crewai-0f47d9414eb6
22. How the Agent2Agent (A2A) protocol enables seamless AI agent collaboration - Wandb, accessed September 25, 2025, https://wandb.ai/byyoung3/Generative-AI/reports/How-the-Agent2Agent-A2A-protocol-enables-seamless-AI-agent-collaboration--VmlldzoxMjQwMjkwNg
23. Unlock AI agent collaboration. Convert ADK agents for A2A | Google ..., accessed September 25, 2025, https://cloud.google.com/blog/products/ai-machine-learning/unlock-ai-agent-collaboration-convert-adk-agents-for-a2a
24. Specification - A2A Protocol, accessed September 25, 2025, https://a2a-protocol.org/dev/specification/

25. google/adk-python: An open-source, code-first Python toolkit for building, evaluating, and deploying sophisticated AI agents with flexibility and control. - GitHub, accessed September 25, 2025, https://github.com/google/adk-python
26. ADK with Agent2Agent (A2A) Protocol - Agent Development Kit, accessed September 25, 2025, https://google.github.io/adk-docs/a2a/
27. Google's Agent Stack in Action: ADK, A2A, MCP on Google Cloud, accessed September 25, 2025, https://codelabs.developers.google.com/instavibe-adk-multi-agents/instructions
28. 2025 Complete Guide: Agent2Agent (A2A) Protocol - The New Standard for AI Agent Collaboration - DEV Community, accessed September 25, 2025, https://dev.to/czmilo/2025-complete-guide-agent2agent-a2a-protocol-the-new-standard-for-ai-agent-collaboration-1pph
29. enso-labs/a2a-langgraph: Agent-to-Agent (A2A Protocol) built on top of LangGraph, accessed September 25, 2025, https://github.com/enso-labs/a2a-langgraph
30. LangChain Integration · themanojdesai/python-a2a Wiki - GitHub, accessed September 25, 2025, https://github.com/themanojdesai/python-a2a/wiki/LangChain-Integration
31. A2A endpoint in LangGraph Server - Docs by LangChain, accessed September 25, 2025, https://docs.langchain.com/langgraph-platform/server-a2a
32. AutoGen Tutorial: Build Multi-Agent AI Applications - DataCamp, accessed September 25, 2025, https://www.datacamp.com/tutorial/autogen-tutorial
33. Empowering multi-agent apps with the open Agent2Agent (A2A) protocol | The Microsoft Cloud Blog, accessed September 25, 2025, https://www.microsoft.com/en-us/microsoft-cloud/blog/2025/05/07/empowering-multi-agent-apps-with-the-open-agent2agent-a2a-protocol/
34. Enhancing the A2A Protocol with Ledger-Anchored Identities and x402 Micropayments for AI Agents - arXiv, accessed September 25, 2025, https://arxiv.org/pdf/2507.19550
35. A2A protocol enterprise implementation case studies - BytePlus, accessed September 25, 2025, https://www.byteplus.com/en/topic/551208
36. Salesforce & Google A2A Protocol: Integration Guide - BytePlus, accessed September 25, 2025, https://www.byteplus.com/en/topic/551107
37. Unleashing the Power of Connected Agents with the Newly Expanded AgentExchange, accessed September 25, 2025, https://www.salesforce.com/blog/connected-agents-agentexchange/
38. Deloitte Global Google Cloud Alliance | AI & GenAI, accessed September 25, 2025, https://www.deloitte.com/global/en/alliances/google/collections/ai-and-genai.html
39. cloud.google.com, accessed September 25, 2025, https://cloud.google.com/blog/products/ai-machine-learning/agent2agent-protocol-is-getting-an-upgrade#:~:text=The%20A2A%20protocol%20enables%20Adobe,system%20processes%20and%20data%20integrations.
40. Announcing Agent Payments Protocol (AP2) | Google Cloud Blog, accessed September 25, 2025,

https://cloud.google.com/blog/products/ai-machine-learning/announcing-agents-to-payments-ap2-protocol