**Chapter 15\_ Inter-Agent Communication (A2A)**

Chapter 15: Inter-Agent Communication (A2A)

Individual AI agents often face limitations when tackling complex, multifaceted problems, even with advanced capabilities. To overcome this, Inter-Agent Communication (A2A) enables diverse AI agents, potentially built with different frameworks, to collaborate effectively. This collaboration involves seamless coordination, task delegation, and information exchange.

Google's A2A protocol is an open standard designed to facilitate this universal communication. This chapter will explore A2A, its practical applications, and its implementation within the Google ADK.

**Inter-Agent Communication Pattern Overview**

The Agent2Agent (A2A) protocol is an open standard designed to enable communication and collaboration between different AI agent frameworks. It ensures interoperability, allowing AI agents developed with technologies like LangGraph, CrewAI, or Google ADK to work together regardless of their origin or framework differences.

A2A is supported by a range of technology companies and service providers, including Atlassian, Box, LangChain, MongoDB, Salesforce, SAP, and ServiceNow. Microsoft plans to integrate A2A into Azure AI Foundry and Copilot Studio, demonstrating its commitment to open protocols. Additionally, Auth0 and SAP are integrating A2A support into their platforms and agents.

As an open-source protocol, A2A welcomes community contributions to facilitate its evolution and widespread adoption.

**Core Concepts of A2A**

The A2A protocol provides a structured approach for agent interactions, built upon several core concepts. A thorough grasp of these concepts is crucial for anyone developing or integrating with A2A-compliant systems. The foundational pillars of A2A include Core Actors, Agent Card, Agent Discovery, Communication and Tasks, Interaction mechanisms, and Security, all of which will be reviewed in detail.

**Core Actors:** A2A involves three main entities:

* User: Initiates requests for agent assistance.
* A2A Client (Client Agent): An application or AI agent that acts on the user's behalf to request actions or information.
* A2A Server (Remote Agent): An AI agent or system that provides an HTTP endpoint to process client requests and return results. The remote agent operates as an "opaque" system, meaning the client does not need to understand its internal operational details.

**Agent Card:** An agent's digital identity is defined by its Agent Card, usually a JSON file. This file contains key information for client interaction and automatic discovery, including the agent's identity, endpoint URL, and version. It also details supported capabilities like streaming or push notifications, specific skills, default input/output modes, and authentication requirements. Below is an example of an Agent Card for a WeatherBot.

|  |
| --- |
| {  "name": "WeatherBot",  "description": "Provides accurate weather forecasts and historical data.",  "url": "http://weather-service.example.com/a2a",  "version": "1.0.0",  "capabilities": {  "streaming": true,  "pushNotifications": false,  "stateTransitionHistory": true  },  "authentication": {  "schemes": [  "apiKey"  ]  },  "defaultInputModes": [  "text"  ],  "defaultOutputModes": [  "text"  ],  "skills": [  {  "id": "get\_current\_weather",  "name": "Get Current Weather",  "description": "Retrieve real-time weather for any location.",  "inputModes": [  "text"  ],  "outputModes": [  "text"  ],  "examples": [  "What's the weather in Paris?",  "Current conditions in Tokyo"  ],  "tags": [  "weather",  "current",  "real-time"  ]  },  {  "id": "get\_forecast",  "name": "Get Forecast",  "description": "Get 5-day weather predictions.",  "inputModes": [  "text"  ],  "outputModes": [  "text"  ],  "examples": [  "5-day forecast for New York",  "Will it rain in London this weekend?"  ],  "tags": [  "weather",  "forecast",  "prediction"  ]  }  ]  } |

**Agent discovery:** it allows clients to find Agent Cards, which describe the capabilities of available A2A Servers. Several strategies exist for this process:

* Well-Known URI: Agents host their Agent Card at a standardized path (e.g., /.well-known/agent.json). This approach offers broad, often automated, accessibility for public or domain-specific use.
* Curated Registries**:** These provide a centralized catalog where Agent Cards are published and can be queried based on specific criteria. This is well-suited for enterprise environments needing centralized management and access control.
* Direct Configuration**:** Agent Card information is embedded or privately shared. This method is appropriate for closely coupled or private systems where dynamic discovery isn't crucial.

Regardless of the chosen method, it is important to secure Agent Card endpoints. This can be achieved through access control, mutual TLS (mTLS), or network restrictions, especially if the card contains sensitive (though non-secret) information.

**Communications and Tasks:** In the A2A framework, communication is structured around asynchronous tasks, which represent the fundamental units of work for long-running processes. Each task is assigned a unique identifier and moves through a series of states—such as submitted, working, or completed—a design that supports parallel processing in complex operations. Communication between agents occurs through a Message.

This communication contains attributes, which are key-value metadata describing the message (like its priority or creation time), and one or more parts, which carry the actual content being delivered, such as plain text, files, or structured JSON data. The tangible outputs generated by an agent during a task are called artifacts. Like messages, artifacts are also composed of one or more parts and can be streamed incrementally as results become available. All communication within the A2A framework is conducted over HTTP(S) using the JSON-RPC 2.0 protocol for payloads. To maintain continuity across multiple interactions, a server-generated contextId is used to group related tasks and preserve context.

**Interaction Mechanisms**: Request/Response (Polling) Server-Sent Events (SSE). A2A provides multiple interaction methods to suit a variety of AI application needs, each with a distinct mechanism:

* Synchronous Request/Response: For quick, immediate operations. In this model, the client sends a request and actively waits for the server to process it and return a complete response in a single, synchronous exchange.
* Asynchronous Polling: Suited for tasks that take longer to process. The client sends a request, and the server immediately acknowledges it with a "working" status and a task ID. The client is then free to perform other actions and can periodically poll the server by sending new requests to check the status of the task until it is marked as "completed" or "failed."
* Streaming Updates (Server-Sent Events - SSE): Ideal for receiving real-time, incremental results. This method establishes a persistent, one-way connection from the server to the client. It allows the remote agent to continuously push updates, such as status changes or partial results, without the client needing to make multiple requests.
* Push Notifications (Webhooks): Designed for very long-running or resource-intensive tasks where maintaining a constant connection or frequent polling is inefficient. The client can register a webhook URL, and the server will send an asynchronous notification (a "push") to that URL when the task's status changes significantly (e.g., upon completion).

The Agent Card specifies whether an agent supports streaming or push notification capabilities. Furthermore, A2A is modality-agnostic, meaning it can facilitate these interaction patterns not just for text, but also for other data types like audio and video, enabling rich, multimodal AI applications. Both streaming and push notification capabilities are specified within the Agent Card.

|  |
| --- |
| #Synchronous Request Example  {  "jsonrpc": "2.0",  "id": "1",  "method": "sendTask",  "params": {  "id": "task-001",  "sessionId": "session-001",  "message": {  "role": "user",  "parts": [  {  "type": "text",  "text": "What is the exchange rate from USD to EUR?"  }  ]  },  "acceptedOutputModes": ["text/plain"],  "historyLength": 5  }  } |

The synchronous request uses the sendTask method, where the client asks for and expects a single, complete answer to its query. In contrast, the streaming request uses the sendTaskSubscribe method to establish a persistent connection, allowing the agent to send back multiple, incremental updates or partial results over time.

|  |
| --- |
| # Streaming Request Example  {  "jsonrpc": "2.0",  "id": "2",  "method": "sendTaskSubscribe",  "params": {  "id": "task-002",  "sessionId": "session-001",  "message": {  "role": "user",  "parts": [  {  "type": "text",  "text": "What's the exchange rate for JPY to GBP today?"  }  ]  },  "acceptedOutputModes": ["text/plain"],  "historyLength": 5  }  } |

**Security:** Inter-Agent Communication (A2A): Inter-Agent Communication (A2A) is a vital component of system architecture, enabling secure and seamless data exchange among agents. It ensures robustness and integrity through several built-in mechanisms.

Mutual Transport Layer Security (TLS): Encrypted and authenticated connections are established to prevent unauthorized access and data interception, ensuring secure communication.

Comprehensive Audit Logs: All inter-agent communications are meticulously recorded, detailing information flow, involved agents, and actions. This audit trail is crucial for accountability, troubleshooting, and security analysis.

Agent Card Declaration: Authentication requirements are explicitly declared in the Agent Card, a configuration artifact outlining the agent's identity, capabilities, and security policies. This centralizes and simplifies authentication management.

Credential Handling: Agents typically authenticate using secure credentials like OAuth 2.0 tokens or API keys, passed via HTTP headers. This method prevents credential exposure in URLs or message bodies, enhancing overall security.

**A2A vs. MCP**

A2A is a protocol that complements Anthropic's Model Context Protocol (MCP) (see Fig. 1). While MCP focuses on structuring context for agents and their interaction with external data and tools, A2A facilitates coordination and communication among agents, enabling task delegation and collaboration.

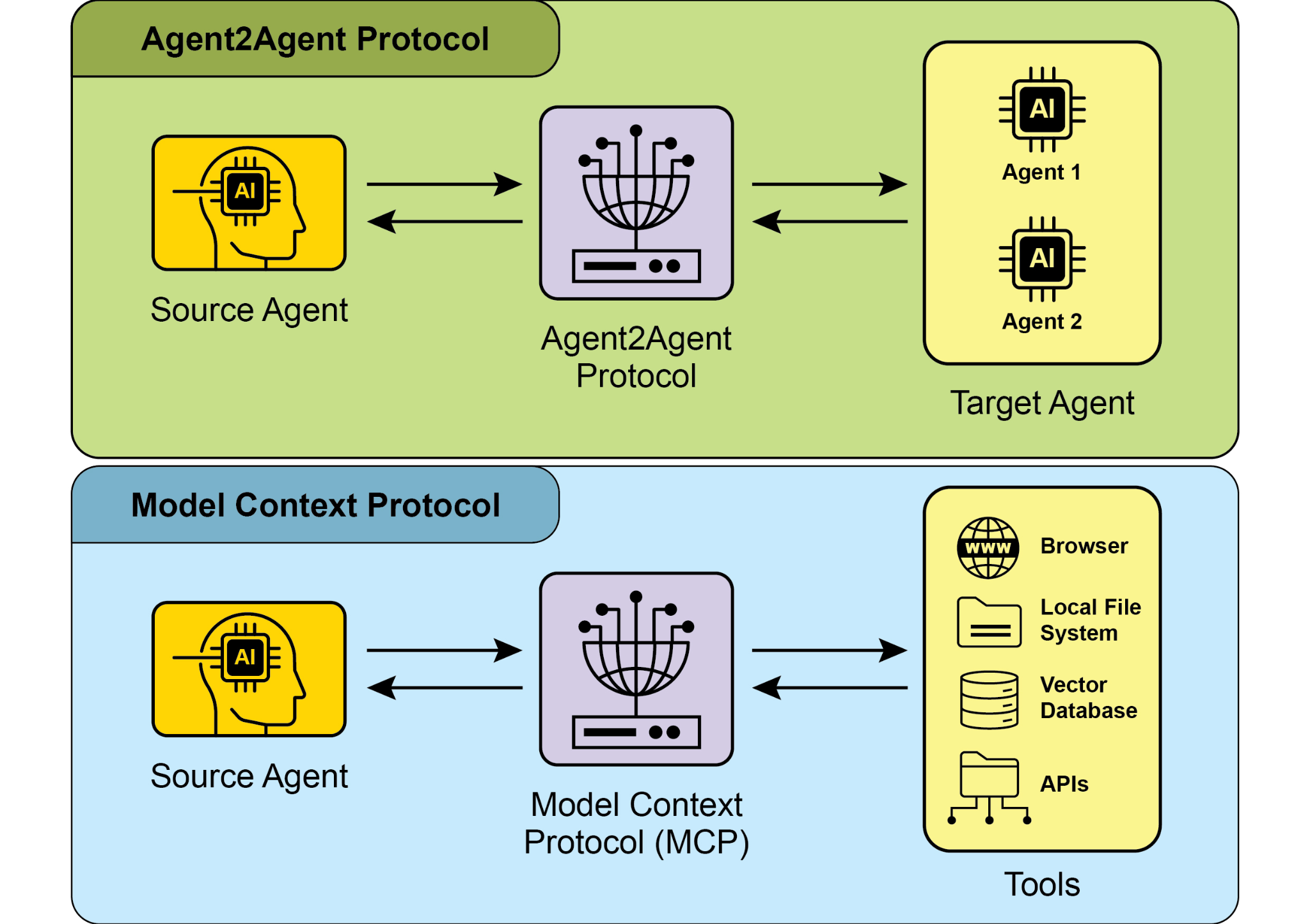


Fig.1: Comparison A2A and MCP Protocols

The goal of A2A is to enhance efficiency, reduce integration costs, and foster innovation and interoperability in the development of complex, multi-agent AI systems. Therefore, a thorough understanding of A2A's core components and operational methods is essential for its effective design, implementation, and application in building collaborative and interoperable AI agent systems..

**Practical Applications & Use Cases**

Inter-Agent Communication is indispensable for building sophisticated AI solutions across diverse domains, enabling modularity, scalability, and enhanced intelligence.

* **Multi-Framework Collaboration:** A2A's primary use case is enabling independent AI agents, regardless of their underlying frameworks (e.g., ADK, LangChain, CrewAI), to communicate and collaborate. This is fundamental for building complex multi-agent systems where different agents specialize in different aspects of a problem.
* **Automated Workflow Orchestration:** In enterprise settings, A2A can facilitate complex workflows by enabling agents to delegate and coordinate tasks. For instance, an agent might handle initial data collection, then delegate to another agent for analysis, and finally to a third for report generation, all communicating via the A2A protocol.
* **Dynamic Information Retrieval:** Agents can communicate to retrieve and exchange real-time information. A primary agent might request live market data from a specialized "data fetching agent," which then uses external APIs to gather the information and send it back.

**Hands-On Code Example**

Let's examine the practical applications of the A2A protocol. The repository at<https://github.com/google-a2a/a2a-samples/tree/main/samples> provides examples in Java, Go, and Python that illustrate how various agent frameworks, such as LangGraph, CrewAI, Azure AI Foundry, and AG2, can communicate using A2A. All code in this repository is released under the Apache 2.0 license. To further illustrate A2A's core concepts, we will review code excerpts focusing on setting up an A2A Server using an ADK-based agent with Google-authenticated tools. Looking at <https://github.com/google-a2a/a2a-samples/blob/main/samples/python/agents/birthday_planner_adk/calendar_agent/adk_agent.py>

|  |
| --- |
| import datetime  from google.adk.agents import LlmAgent # type: ignore[import-untyped]  from google.adk.tools.google\_api\_tool import CalendarToolset # type: ignore[import-untyped]  async def create\_agent(client\_id, client\_secret) -> LlmAgent:  """Constructs the ADK agent."""  toolset = CalendarToolset(client\_id=client\_id, client\_secret=client\_secret)  return LlmAgent(  model='gemini-2.0-flash-001',  name='calendar\_agent',  description="An agent that can help manage a user's calendar",  instruction=f"""  You are an agent that can help manage a user's calendar.  Users will request information about the state of their calendar  or to make changes to their calendar. Use the provided tools for interacting with the calendar API.  If not specified, assume the calendar the user wants is the 'primary' calendar.  When using the Calendar API tools, use well-formed RFC3339 timestamps.  Today is {datetime.datetime.now()}.  """,  tools=await toolset.get\_tools(),  ) |

This Python code defines an asynchronous function `create\_agent` that constructs an ADK LlmAgent. It begins by initializing a `CalendarToolset` using the provided client credentials to access the Google Calendar API. Subsequently, an `LlmAgent` instance is created, configured with a specified Gemini model, a descriptive name, and instructions for managing a user's calendar. The agent is furnished with calendar tools from the `CalendarToolset`, enabling it to interact with the Calendar API and respond to user queries regarding calendar states or modifications. The agent's instructions dynamically incorporate the current date for temporal context. To illustrate how an agent is constructed, let's examine a key section from the calendar\_agent found in the A2A samples on GitHub.

The code below shows how the agent is defined with its specific instructions and tools. Please note that only the code required to explain this functionality is shown; you can access the complete file here:<https://github.com/a2aproject/a2a-samples/blob/main/samples/python/agents/birthday_planner_adk/calendar_agent/__main__.py>

|  |
| --- |
| def main(host: str, port: int):  # Verify an API key is set.  # Not required if using Vertex AI APIs.  if os.getenv('GOOGLE\_GENAI\_USE\_VERTEXAI') != 'TRUE' and not os.getenv(  'GOOGLE\_API\_KEY'  ):  raise ValueError(  'GOOGLE\_API\_KEY environment variable not set and '  'GOOGLE\_GENAI\_USE\_VERTEXAI is not TRUE.'  )  skill = AgentSkill(  id='check\_availability',  name='Check Availability',  description="Checks a user's availability for a time using their Google Calendar",  tags=['calendar'],  examples=['Am I free from 10am to 11am tomorrow?'],  )  agent\_card = AgentCard(  name='Calendar Agent',  description="An agent that can manage a user's calendar",  url=f'http://{host}:{port}/',  version='1.0.0',  defaultInputModes=['text'],  defaultOutputModes=['text'],  capabilities=AgentCapabilities(streaming=True),  skills=[skill],  )  adk\_agent = asyncio.run(create\_agent(  client\_id=os.getenv('GOOGLE\_CLIENT\_ID'),  client\_secret=os.getenv('GOOGLE\_CLIENT\_SECRET'),  ))  runner = Runner(  app\_name=agent\_card.name,  agent=adk\_agent,  artifact\_service=InMemoryArtifactService(),  session\_service=InMemorySessionService(),  memory\_service=InMemoryMemoryService(),  )  agent\_executor = ADKAgentExecutor(runner, agent\_card)  async def handle\_auth(request: Request) -> PlainTextResponse:  await agent\_executor.on\_auth\_callback(  str(request.query\_params.get('state')), str(request.url)  )  return PlainTextResponse('Authentication successful.')  request\_handler = DefaultRequestHandler(  agent\_executor=agent\_executor, task\_store=InMemoryTaskStore()  )  a2a\_app = A2AStarletteApplication(  agent\_card=agent\_card, http\_handler=request\_handler  )  routes = a2a\_app.routes()  routes.append(  Route(  path='/authenticate',  methods=['GET'],  endpoint=handle\_auth,  )  )  app = Starlette(routes=routes)  uvicorn.run(app, host=host, port=port)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

This Python code demonstrates setting up an A2A-compliant "Calendar Agent" for checking user availability using Google Calendar. It involves verifying API keys or Vertex AI configurations for authentication purposes. The agent's capabilities, including the "check\_availability" skill, are defined within an AgentCard, which also specifies the agent's network address. Subsequently, an ADK agent is created, configured with in-memory services for managing artifacts, sessions, and memory. The code then initializes a Starlette web application, incorporates an authentication callback and the A2A protocol handler, and executes it using Uvicorn to expose the agent via HTTP.

These examples illustrate the process of building an A2A-compliant agent, from defining its capabilities to running it as a web service. By utilizing Agent Cards and ADK, developers can create interoperable AI agents capable of integrating with tools like Google Calendar. This practical approach demonstrates the application of A2A in establishing a multi-agent ecosystem.

Further exploration of A2A is recommended through the code demonstration at<https://www.trickle.so/blog/how-to-build-google-a2a-project>. Resources available at this link include sample A2A clients and servers in Python and JavaScript, multi-agent web applications, command-line interfaces, and example implementations for various agent frameworks.

**At a Glance**

**What:** Individual AI agents, especially those built on different frameworks, often struggle with complex, multi-faceted problems on their own. The primary challenge is the lack of a common language or protocol that allows them to communicate and collaborate effectively. This isolation prevents the creation of sophisticated systems where multiple specialized agents can combine their unique skills to solve larger tasks. Without a standardized approach, integrating these disparate agents is costly, time-consuming, and hinders the development of more powerful, cohesive AI solutions.

**Why:** The Inter-Agent Communication (A2A) protocol provides an open, standardized solution for this problem. It is an HTTP-based protocol that enables interoperability, allowing distinct AI agents to coordinate, delegate tasks, and share information seamlessly, regardless of their underlying technology. A core component is the Agent Card, a digital identity file that describes an agent's capabilities, skills, and communication endpoints, facilitating discovery and interaction. A2A defines various interaction mechanisms, including synchronous and asynchronous communication, to support diverse use cases. By creating a universal standard for agent collaboration, A2A fosters a modular and scalable ecosystem for building complex, multi-agent Agentic systems.

**Rule of thumb:** Use this pattern when you need to orchestrate collaboration between two or more AI agents, especially if they are built using different frameworks (e.g., Google ADK, LangGraph, CrewAI). It is ideal for building complex, modular applications where specialized agents handle specific parts of a workflow, such as delegating data analysis to one agent and report generation to another. This pattern is also essential when an agent needs to dynamically discover and consume the capabilities of other agents to complete a task.

**Visual summary**

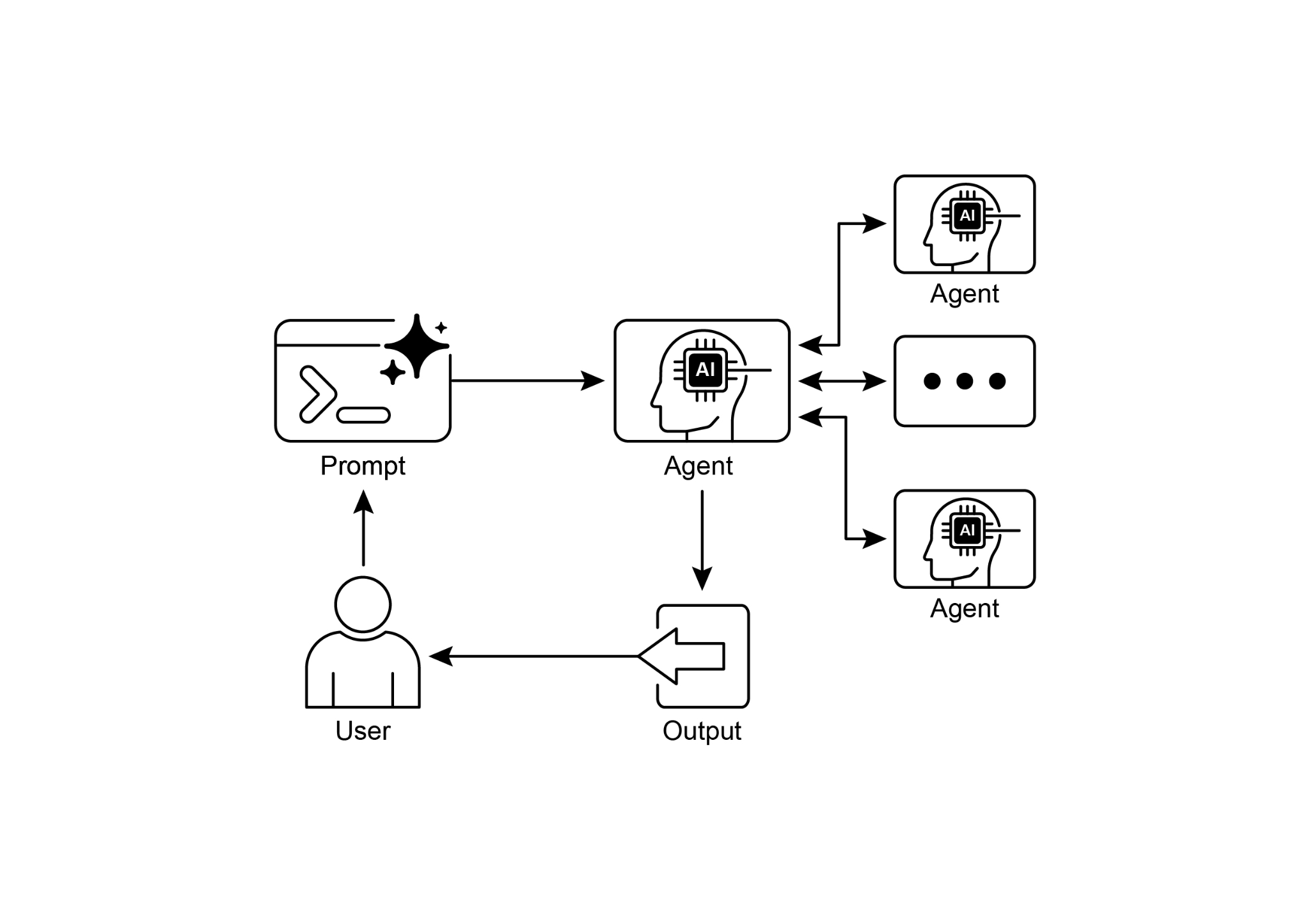


Fig.2: A2A inter-agent communication pattern

**Key Takeaways**

Key Takeaways:

* The Google A2A protocol is an open, HTTP-based standard that facilitates communication and collaboration between AI agents built with different frameworks.
* An AgentCard serves as a digital identifier for an agent, allowing for automatic discovery and understanding of its capabilities by other agents.
* A2A offers both synchronous request-response interactions (using `tasks/send`) and streaming updates (using `tasks/sendSubscribe`) to accommodate varying communication needs.
* The protocol supports multi-turn conversations, including an `input-required` state, which allows agents to request additional information and maintain context during interactions.
* A2A encourages a modular architecture where specialized agents can operate independently on different ports, enabling system scalability and distribution.
* Tools such as Trickle AI aid in visualizing and tracking A2A communications, which helps developers monitor, debug, and optimize multi-agent systems.
* While A2A is a high-level protocol for managing tasks and workflows between different agents, the Model Context Protocol (MCP) provides a standardized interface for LLMs to interface with external resources

**Conclusions**

The Inter-Agent Communication (A2A) protocol establishes a vital, open standard to overcome the inherent isolation of individual AI agents. By providing a common HTTP-based framework, it ensures seamless collaboration and interoperability between agents built on different platforms, such as Google ADK, LangGraph, or CrewAI. A core component is the Agent Card, which serves as a digital identity, clearly defining an agent's capabilities and enabling dynamic discovery by other agents. The protocol's flexibility supports various interaction patterns, including synchronous requests, asynchronous polling, and real-time streaming, catering to a wide range of application needs.

This enables the creation of modular and scalable architectures where specialized agents can be combined to orchestrate complex automated workflows. Security is a fundamental aspect, with built-in mechanisms like mTLS and explicit authentication requirements to protect communications. While complementing other standards like MCP, A2A's unique focus is on the high-level coordination and task delegation between agents. The strong backing from major technology companies and the availability of practical implementations highlight its growing importance. This protocol paves the way for developers to build more sophisticated, distributed, and intelligent multi-agent systems. Ultimately, A2A is a foundational pillar for fostering an innovative and interoperable ecosystem of collaborative AI.

**References**

1. Chen, B. (2025, April 22). *How to Build Your First Google A2A Project: A Step-by-Step Tutorial*. Trickle.so Blog. <https://www.trickle.so/blog/how-to-build-google-a2a-project>
2. Google A2A GitHub Repository. <https://github.com/google-a2a/A2A>
3. Google Agent Development Kit (ADK) <https://google.github.io/adk-docs/>
4. Getting Started with Agent-to-Agent (A2A) Protocol: <https://codelabs.developers.google.com/intro-a2a-purchasing-concierge#0>
5. Google AgentDiscovery - <https://a2a-protocol.org/latest/>
6. Communication between different AI frameworks such as LangGraph, CrewAI, and Google ADK [https://www.trickle.so/blog/how-to-build-google-a2a-project](https://www.trickle.so/blog/how-to-build-google-a2a-project#setting-up-your-a2a-development-environment)
7. Designing Collaborative Multi-Agent Systems with the A2A Protocol <https://www.oreilly.com/radar/designing-collaborative-multi-agent-systems-with-the-a2a-protocol/>

**第15章\_代理间通信（A2A）**

第15章：智能体间通信（A2A）

即使具备先进能力，单个AI智能体在处理复杂、多方面的问题时往往也会面临局限性。为克服这一问题，智能体间通信（A2A）使不同的AI智能体（可能采用不同框架构建）能够有效协作。这种协作涉及无缝协调、任务分配和信息交换。

谷歌的A2A协议是一项开放标准，旨在促进这种通用通信。本章将探讨A2A协议、其实际应用以及在谷歌ADK中的实现。

**智能体间通信模式概述**

Agent2Agent (A2A)协议是一项开放标准，旨在实现不同AI智能体框架之间的通信与协作。它确保了互操作性，允许使用LangGraph、CrewAI或Google ADK等技术开发的AI智能体协同工作，无论其来源或框架差异如何。

A2A得到了一系列科技公司和服务提供商的支持，其中包括艾特莱森、Box、LangChain、MongoDB、Salesforce、SAP和ServiceNow。微软计划将A2A集成到Azure AI Foundry和Copilot Studio中，这表明其对开放协议的承诺。此外，Auth0和SAP正在将A2A支持集成到其平台和代理中。

作为一种开源协议，A2A欢迎社区贡献以促进其发展和广泛采用。

**A2A的核心概念**

A2A协议为代理交互提供了一种结构化方法，它基于几个核心概念构建。深入理解这些概念对于任何开发或集成符合A2A标准系统的人来说都至关重要。A2A的基础支柱包括核心参与者、代理卡片、代理发现、通信与任务、交互机制和安全，所有这些都将进行详细审查。

**核心参与者：** A2A涉及三个主要实体：

* 用户：发起请求以寻求代理协助。
* A2A客户端（客户端代理）：代表用户请作或信息的应用程序或AI代理。
* A2A服务器（远程代理）：一种AI代理或系统，提供HTTP端点来处理客户端请求并返回结果。远程代理作为一个“不透明”系统运行，这意味着客户端无需了解其内部操作细节。

**代理卡片：**代理的数字身份由其代理卡片定义，通常是一个JSON文件。该文件包含客户端交互和自动发现的关键信息，包括代理的身份、端点URL和版本。它还详细说明了支持的功能，如流式传输或推送通知、特定技能、默认输入/输出模式和认证要求。以下是一个天气机器人的代理卡片示例。

|  |
| --- |
| {  "name": "天气机器人",  "description": "提供准确的天气预报和历史数据。",  "url": "http://weather-service.example.com/a2a",  "version": "1.0.0",  "capabilities": {  "streaming": true,  "pushNotifications": false,  "stateTransitionHistory": true  },  "认证": {  "schemes": [  "apiKey"  ]  },  "defaultInputModes": [  "文本"  ],  "defaultOutputModes": [  "文本"  ],  技能"：[  {  "id": "get\_current\_weather",  "name": "获取当前天气",  "description": "获取任何地点的实时天气信息。",  "inputModes": [  "文本"  ],  "outputModes": [  "文本"  ],  "示例": [  "巴黎的天气怎么样？",  东京当前的状况  ],  "标签": [  "天气",  "当前",  实时  ]  },  {  "id": "获取预报",  "name": "获取预报",  "description": "获取5天天气预报。",  "inputModes": [  "文本"  ],  "outputModes": [  "文本"  ],  "示例": [  纽约5天天气预报  这个周末伦敦会下雨吗？  ],  "标签": [  "天气",  "预测",  "预测"  ]  }  ]  } |

**代理发现：**它允许客户端查找代理卡，这些代理卡描述了可用的A2A服务器的功能。此过程有多种策略：

* 知名 URI：代理将其代理卡片托管在标准化路径（例如，/.well-known/agent.json）。这种方法为公共或特定领域的使用提供了广泛的、通常是自动化的可访问性。
* 精选注册表**：**这些注册表提供了一个集中的曲库，代理卡会在其中发布，并可根据特定标准进行查询。这非常适合需要集中管理和权限改造的企业环境。
* 直接配置**：**代理卡信息被嵌入或私下共享。这种方法适用于紧密耦合或私有系统，在这些系统中动态发现并非至关重要。

无论选择何种方法，确保代理卡端点的安全都很重要。这可以通过权限改造、相互TLS（mTLS）或网络限制来实现，特别是当卡中包含敏感（但非机密）信息时。

**通信与任务：**在A2A框架中，通信围绕异步任务构建，这些任务代表长期运行过程的基本工作单元。每个任务都被分配一个唯一标识符，并经历一系列状态，如已提交、正在处理或已完成，这种设计支持复杂操作中的并行处理。代理之间的通信通过消息进行。

此通信包含属性（即描述消息的键值元数据，如优先级或创建时间），以及一个或多个部分，这些部分承载实际要传递的内容，如纯文本、文件或结构化JSON数据。代理在任务期间生成的有形输出称为工件。与消息一样，工件也由一个或多个部分组成，并且可以在结果可用时逐步流式传输。A2A框架内的所有通信均通过HTTP(S)进行，使用JSON-RPC 2.0协议处理有效负载。为了在多个交互中保持连续性，使用服务器生成的contextId对相关任务进行分组并保留上下文。

**交互机制**：请求/响应（轮询）、服务器发送事件（SSE）。A2A提供多种交互方法，以满足各种AI应用需求，每种方法都有独特的机制：

* 同步请求/响应：适用于快速、即时的操作。在这种模式下，客户端发送请求并主动等待服务器处理该请求，并在一次同步交互中返回完整的响应。
* 异步轮询：适用于处理时间较长的任务。客户端发送请求，服务器立即以“正在处理”状态和任务 ID 进行确认。然后客户端可以自由执行其他操作，并可以通过发送新请求定期轮询服务器，以检查任务状态，直到任务被标记为“已完成”或“失败”。
* 流式更新（服务器发送事件 - SSE）：非常适合接收实时、增量的结果。此方法建立了从服务器到客户端的持久单向连接。它允许远程代理持续推送更新，如状态变化或部分结果，而客户端无需多次发起请求。
* 推送通知（网络钩子）：适用于非常耗时或资源密集型的任务，在这些任务中，保持持续连接或频繁轮询效率低下。客户端可以注册一个网络钩子 URL，当任务状态发生重大变化（例如完成时），服务器将向该 URL 发送异步通知（“推送”）。

代理卡片指定了代理是否支持流式传输或消息推送功能。此外，A2A与模态无关，这意味着它不仅可以为文本，还可以为音频和视频等其他数据类型促进这些交互模式，从而实现丰富的多模态AI应用。流式传输和消息推送功能均在代理卡片中指定。

|  |
| --- |
| #同步请求示例  {  "jsonrpc": "2.0",  "id": "1",  "method": "sendTask",  "params": {  "id": "task-001",  "sessionId": "session-001",  "message": {  "role": "用户",  "parts": [  {  "type": "text",  "text": "USD兑换EUR的汇率是多少？"  }  ]  },  "acceptedOutputModes": ["text/plain"],  "historyLength": 5  }  } |

同步请求使用sendTask方法，客户端通过该方法发起查询并期望得到单一、完整的答案。相比之下，流式请求使用sendTaskSubscribe方法来建立持久连接，允许代理随着时间的推移返回多个增量更新或部分结果。

|  |
| --- |
| # 流式请求示例  {  "jsonrpc": "2.0",  "id": "2",  "method": "sendTaskSubscribe",  "params": {  "id": "task-002",  "sessionId": "session-001",  "message": {  "role": "用户",  "parts": [  {  "type": "text",  "text": "今天日元兑GBP的汇率是多少？"  }  ]  },  "acceptedOutputModes": ["text/plain"],  "historyLength": 5  }  } |

**安全：** 代理间通信（A2A）：代理间通信（A2A）是系统架构的重要组成部分，它使代理之间能够安全、无缝地进行数据交换。通过多种内置机制，它确保了系统的健壮性和完整性。

相互安全传输层协议（TLS）：建立加密和认证的连接，以防止未经授权的访问和数据拦截，确保通信安全。

全面审计日志：所有代理间通信都被详细记录，包括信息流、涉及的代理和操作。此审计跟踪对于责任认定、故障排除和安全分析至关重要。

代理卡声明：认证要求在代理卡中明确声明，代理卡是一种配置工件，概述了代理的身份、能力和安全策略。这集中并简化了认证管理。

凭证处理：代理通常使用安全凭证（如 OAuth 2.0 令牌或 API 密钥）进行身份验证，这些凭证通过 HTTP 标头传递。这种方法可防止凭证在 URL 或消息体中暴露，从而增强整体安全性。

**A2A与MCP对比**

A2A是一种补充Anthropic的模型上下文协议（MCP）的协议（见图1）。MCP侧重于为智能体构建上下文以及它们与外部数据和工具的交互，而A2A则促进智能体之间的协调和通信，实现任务委派和协作。

图1：A2A和MCP协议比较

A2A的目标是提高效率、降低集成成本，并在复杂的多智能体AI系统开发中促进创新和互操作性。因此，全面了解A2A的核心组件和操作方法，对于其在构建协作和互操作的AI智能体系统中的有效设计、实施和应用至关重要。

**实际应用与用例**

智能体间通信对于在不同领域构建复杂的AI解决方案不可或缺，它能实现模块化、可扩展性和增强智能。

* **多框架协作：** A2A的主要用例是使独立的AI智能体能够进行通信和协作，无论其底层框架如何（例如ADK、LangChain、CrewAI）。这对于构建复杂的多智能体系统至关重要，在这些系统中，不同的智能体专注于问题的不同方面。
* **自动化工作流编排：**在企业环境中，A2A可以通过使智能体能够委派和协调任务来促进复杂的工作流。例如，一个智能体可能负责初始的数据采集，然后委派给另一个智能体进行分析，最后委派给第三个智能体生成报告，所有这些都通过A2A协议进行通信。
* **动态信息检索：**智能体可以进行通信，以检索和交换实时信息。主要智能体可能会向专门的“数据获取智能体”请求实时市场数据，后者随后使用外部应用程序编程接口（API）收集信息并将其返回。

**实践代码示例**

让我们来探讨一下A2A协议的实际应用。[位于https://github.com/google-a2a/a2a-samples/tree/main/samples的仓库](https://github.com/google-a2a/a2a-samples/tree/main/samples)提供了Java、Go和Python的示例，展示了各种代理框架，如LangGraph、CrewAI、Azure AI Foundry和AG2，如何使用A2A进行通信。该仓库中的所有代码均根据Apache 2.0许可证发布。为了进一步说明A2A的核心概念，我们将查看专注于使用基于ADK的代理和谷歌认证工具设置A2A服务器的代码片段。查看<https://github.com/google-a2a/a2a-samples/blob/main/samples/python/agents/birthday_planner_adk/calendar_agent/adk_agent.py>

|  |
| --- |
| 导入datetime  from google.adk.agents import LlmAgent # type: ignore[import-untyped]  from google.adk.tools.google\_api\_tool import CalendarToolset # type: ignore[import-untyped]  async def create\_agent(client\_id, client\_secret) -> LlmAgent:  """构建ADK代理。"""  toolset = CalendarToolset(client\_id=client\_id, client\_secret=client\_secret)  返回 LlmAgent(  model='gemini-2.0-flash-001',  name='日历代理',  description="一个可以帮助管理用户日历的代理",  instruction=f"""  你是一个可以帮助管理用户日历的代理。  用户将请求有关其日历状态的信息  或对其日历进行更改。使用提供的工具与日历 API 进行交互。  如果未指定，则假定用户想要的日历是“主”日历。  使用日历 API 工具时，请使用格式正确的 RFC3339 时间戳。  今天是{datetime.datetime.now()}。  """,  tools=await toolset.get\_tools(),  ) |

这段Python代码定义了一个异步函数`create\_agent`，用于构建一个ADK LlmAgent。它首先使用提供的客户端凭证初始化一个`CalendarToolset`，以访问Google日历API。随后，创建一个`LlmAgent`实例，并使用指定的Gemini模型、描述性名称和管理用户日历的说明进行配置。该代理配备了来自`CalendarToolset`的日历工具，使其能够与日历API交互，并响应用户关于日历状态或修改的查询。代理的说明会动态地包含当前日期，以提供时间上下文。为了说明如何构建代理，让我们来查看一下GitHub上A2A示例中`calendar\_agent`的关键部分。

以下代码展示了如何定义代理及其特定指令和工具。请注意，这里仅展示了解释此功能所需的代码；你可以在此处访问完整文件：<https://github.com/a2aproject/a2a-samples/blob/main/samples/python/agents/birthday_planner_adk/calendar_agent/__main__.py>

|  |
| --- |
| def main(host: str, port: int):  # 验证是否已设置 API 密钥。  # 如果使用Vertex AI API，则不需要。  if os.getenv('GOOGLE\_GENAI\_USE\_VERTEXAI')!= 'TRUE' and not os.getenv(  'GOOGLE\_API\_KEY'  ):  raise ValueError(  未设置 "GOOGLE\_API\_KEY" 环境变量，且  'GOOGLE\_GENAI\_USE\_VERTEXAI 不是 TRUE。'  )  skill = AgentSkill(  id='check\_availability',  name='检查可用性',  description="使用用户的谷歌日历检查其某个时间段的可用性",  tags=['日历'],  examples=['我明天上午10点到11点有空吗？'],  )  agent\_card = AgentCard(  name='日历代理',  description="能够管理用户日历的代理",  url=f'http://{host}:{port}/',  version='1.0.0',  defaultInputModes=['text'],  defaultOutputModes=['text'],  capabilities=AgentCapabilities(streaming=True),  skills=[skill],  )  adk\_agent = asyncio.run(create\_agent(  client\_id=os.getenv('GOOGLE\_CLIENT\_ID'),  client\_secret=os.getenv('GOOGLE\_CLIENT\_SECRET'),  ))  runner = Runner(  app\_name=agent\_card.name,  agent=adk\_agent,  artifact\_service=InMemoryArtifactService(),  session\_service=InMemorySessionService(),  memory\_service=InMemoryMemoryService(),  )  agent\_executor = ADKAgentExecutor(runner, agent\_card)  async def handle\_auth(request: Request) -> PlainTextResponse:  await agent\_executor.on\_auth\_callback(  str(request.query\_params.get('state')), str(request.url)  )  return PlainTextResponse('认证成功。')  request\_handler = DefaultRequestHandler(  agent\_executor=agent\_executor, task\_store=InMemoryTaskStore()  )  a2a\_app = A2AStarletteApplication(  agent\_card=代理卡, http\_handler=请求处理程序  )  routes = a2a\_app.routes()  routes.append(  路线(  path='/authenticate',  methods=['GET'],  endpoint=handle\_auth,  )  )  app = Starlette(routes=routes)  uvicorn.run(app, host=host, port=port)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

这段Python代码展示了如何设置一个符合A2A标准的“日历代理”，用于通过谷歌日历检查用户的可用性。它涉及验证API密钥或Vertex AI配置，以实现身份验证目的。代理的功能，包括“检查可用性”技能，在一个代理卡片中定义，该卡片还指定了代理的网络地址。随后，创建一个ADK代理，并配置内存服务来管理工件、会话和内存。代码接着初始化一个Starlette Web应用程序，集成身份验证回调和A2A协议处理程序，并使用Uvicorn执行该应用程序，通过HTTP公开代理。

这些示例展示了构建符合A2A标准的智能体的过程，从定义其能力到将其作为Web服务运行。通过利用智能体卡片和ADK，开发者可以创建能够与谷歌日历等工具集成的可互操作的AI智能体。这种实用方法展示了A2A在建立多智能体生态系统中的应用。

建议通过代码演示进一步探索 A2A[，请访问https://www.trickle.so/blog/how-to-build-google-a2a-project](https://www.trickle.so/blog/how-to-build-google-a2a-project)。此链接提供的资源包括用 Python 和 JavaScript 编写的 A2A 客户端和服务器示例、多智能体 Web 应用程序、命令行界面，以及各种智能体框架的示例实现。

**概览**

**问题：**单个AI智能体，尤其是那些基于不同框架构建的智能体，往往难以独自应对复杂、多方面的问题。主要挑战在于缺乏一种通用语言或协议，使它们能够有效沟通和协作。这种孤立状态阻碍了复杂系统的创建，在这些系统中，多个专业智能体可以结合各自独特的技能来解决更大的任务。如果没有标准化的方法，整合这些不同的智能体成本高昂、耗时费力，并且阻碍了更强大、更具凝聚力的AI解决方案的发展。

**原因：**代理间通信（A2A）协议为这一问题提供了一个开放、标准化的解决方案。它是一种基于HTTP的协议，可实现互操作性，使不同的AI代理能够无缝地进行协调、委派任务和共享信息，而不受其底层技术的影响。一个核心组件是代理卡片，这是一种数字身份文件，用于描述代理的能力、技能和通信端点，促进发现和交互。A2A定义了各种交互机制，包括同步和异步通信，以支持不同的用例。通过为代理协作创建通用标准，A2A促进了一个模块化和可扩展的生态系统，用于构建复杂的多代理智能系统。

**经验法则：**当你需要协调两个或更多AI智能体之间的协作时，尤其是当它们使用不同的框架（例如，Google ADK、LangGraph、CrewAI）构建时，请使用此模式。它非常适合构建复杂的模块化应用程序，其中专门的智能体处理工作流程的特定部分，例如将数据分析委托给一个智能体，将报告生成委托给另一个智能体。当一个智能体需要动态发现并利用其他智能体的能力来完成任务时，此模式也至关重要。

**可视化总结**

图2：A2A代理间通信模式

**要点总结**

要点总结：

* 谷歌A2A协议是一种基于HTTP的开放标准，它促进了使用不同框架构建的AI智能体之间的通信和协作。
* 代理卡是代理的数字标识符，允许其他代理自动发现并了解其能力。
* A2A同时提供同步请求-响应交互（使用`tasks/send`）和流式更新（使用`tasks/sendSubscribe`），以满足不同的通信需求。
* 该协议支持多轮对话，包括“需要输入”状态，此状态允许代理请求额外信息并在交互过程中保持上下文连贯性。
* A2A鼓励采用模块化架构，在这种架构中，专门的代理可以在不同端口上独立运行，从而实现系统的可扩展性和分布式部署。
* 像Trickle AI这样的工具可帮助可视化和跟踪A2A通信，这有助于开发人员监控、调试和优化多智能体系统。
* 虽然A2A是用于管理不同智能体之间任务和工作流的高级协议，但模型上下文协议（MCP）为大语言模型（LLMs）与外部资源交互提供了标准化接口

**结论**

代理间通信（A2A）协议建立了一个至关重要的开放标准，以克服单个AI代理固有的孤立性。通过提供一个基于HTTP的通用框架，它确保了基于不同平台（如Google ADK、LangGraph或CrewAI）构建的代理之间的无缝协作和互操作性。核心组件是代理卡片，它作为数字身份，明确界定了代理的能力，并使其他代理能够动态发现。该协议的灵活性支持各种交互模式，包括同步请求、异步轮询和实时流，满足了广泛的应用需求。

这使得模块化和可扩展架构的创建成为可能，在这种架构中，专业代理可以组合起来编排复杂的自动化工作流程。安全性是一个基本方面，内置了诸如 mTLS 和明确的身份验证要求等机制来保护通信。在补充 MCP 等其他标准的同时，A2A 的独特重点在于代理之间的高级协调和任务委派。主要科技公司的大力支持以及实际实现方案的可用性凸显了其日益增长的重要性。该协议为开发者构建更复杂、分布式和智能的多代理系统铺平了道路。最终，A2A 是培育创新且可互操作的协作 AI 生态系统的基础支柱。

**参考文献**

1. 陈，B.（2025年4月22日）。*如何构建你的第一个谷歌A2A项目：分步教程*Trickle.so博客。<https://www.trickle.so/blog/how-to-build-google-a2a-project>
2. Google A2A GitHub 仓库。<https://github.com/google-a2a/A2A>
3. 谷歌代理开发套件 (ADK)<https://google.github.io/adk-docs/>
4. 代理到代理（A2A）协议入门：<https://codelabs.developers.google.com/intro-a2a-purchasing-concierge#0>
5. Google AgentDiscovery - <https://a2a-protocol.org/latest/>
6. 不同AI框架（如LangGraph、CrewAI和Google ADK）之间的通信[https://www.trickle.so/blog/how-to-build-google-a2a-project](https://www.trickle.so/blog/how-to-build-google-a2a-project#setting-up-your-a2a-development-environment)
7. 使用A2A协议设计协作式多智能体系统<https://www.oreilly.com/radar/designing-collaborative-multi-agent-systems-with-the-a2a-protocol/>