Architecture Design of the Runtime Engine of Google A2A

Xiao-Feng Li

xli@apache.org

Aug 10, 2025

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

A2A server as a proxy agent

The Google Agent-to-Agent (A2A) protocol enables collaboration between heterogeneous AI agents by exposing any server-side agent as a standardized interface to its clients. The A2A runtime engine is designed as a **proxy agent**: it does not need to know the internal implementation of the actual agent but instead provides a uniform way for clients to interact with it.

This proxy pattern is the foundation of the A2A server's architecture:

- It **proxies responses** from the underlying agent to the client.
- It **manages execution state** and message history in a consistent manner.
- It **supports multiple communication modes** (streaming, non-streaming, notifications).

Core Components

Consumer side (architecturally):

- **RequestHandler**: Entry point for client requests. Creates async AgentExecutor tasks, listens on the EventQueue, and delivers responses back to the client.
- **ResultAggregator**: Consumes Events and updates the Task object. Ensures client-visible state reflects consumed events only.

Producer side (architecturally):

- **AgentExecutor**: Wraps the actual agent execution. Produces Events from agent responses and enqueues them via TaskUpdater.
- **TaskUpdater**: Produces task deltas encoded as Events. Does not directly mutate the Task object.

Data in between (architecturally):

- **EventQueue**: Central async buffer. Mediates between event production (executor) and consumption (handler). Supports multiple consumption patterns.
- **Task Object**: Persistent representation of client-visible state and history. Supports retrieval, continuation, and multi-turn interactions.

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

Design philosophy:
1. Proxy Pattern as the Guiding Principle

The A2A server positions itself between a client and an actual agent. It acts as a **standardized proxy agent** with its own lifecycle management, ensuring that clients see a consistent and predictable view of tasks regardless of the agent framework in use.

Key implication: **The Task object on the server represents what the client sees, not what the agent internally produces.**

- Maintains a clean boundary between server (proxy) and agent (implementation).
- Enables A2A to work with heterogeneous agent frameworks (e.g., ADK, LangGraph).

Design philosophy:
2. Event-Driven Separation of Concerns

A distinguishing design decision is the **temporal separation** between:

- **Event Production (agent executor)** – agents generate raw outputs, encoded as Events.
- **Event Consumption (request handler + result aggregator)** – events are processed into client-visible state.

This separation provides resilience, filtering, and flexibility in how the client-facing state is updated.

- Fire-and-forget event production by AgentExecutor.
- Task state updates only on consumption by ResultAggregator.
- Guarantees consistency and prevents corruption of Task state.

Design philosophy:
3. Central State for Multi-Turn Interactions

A2A maintains a **Task object** as the central abstraction of execution state.

- Tracks lifecycle states (submitted, in-progress, input-required, completed).
- Persists message history for multi-turn conversations.
- Can be retrieved or resumed across requests.

Design philosophy:
4. Asynchronous and Flexible Communication

The design leverages async producer-consumer patterns:

- **AgentExecutor** asynchronously enqueues events.
- **RequestHandler** consumes and forwards events to clients.

This allows support for **streaming**, **batch responses**, and **webhook notifications** without altering the agent-facing API.

- Clients can flexibly choose response patterns without affecting agent execution logic.

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

Simplified Workflow (high-level)

Setup – RequestHandler creates EventQueue and starts AgentExecutor.

Agent Execution – AgentExecutor runs agent logic and enqueues Events.

- Events may encode status deltas, artifacts, or notifications.

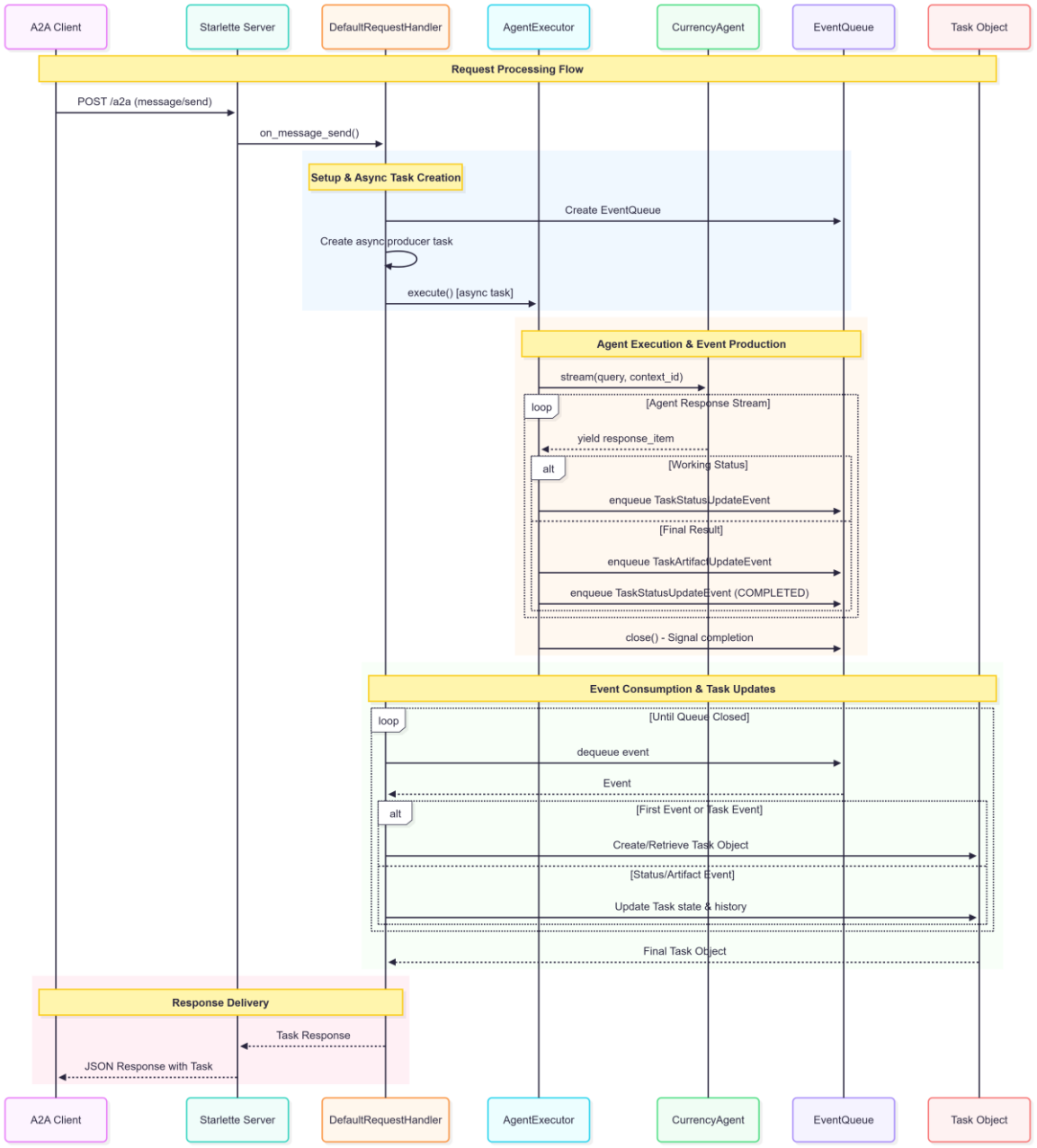
Event Consumption – RequestHandler dequeues Events, ResultAggregator applies updates to Task.

- Task creation may occur either explicitly (at request start) or implicitly (via ensure_task in event processing).

Response Delivery – Responses returned to client (batch or streaming).

Overall Flow

Agents register capabilities in Agent Cards and communicate over HTTP (JSON-RPC) with streaming feedback. The A2A *server* (often built on a framework like Starlette) routes client requests to the appropriate agent executor. A simplified common workflow of the A2A server is as follows (shown in the simplified sequence diagram), a **RequestHandler** may create an asynchronous event queue and spawn a producer **AgentExecutor** to run the agent logic. The remote agent (e.g. *CurrencyAgent* for exchange rate) processes the query and yields intermediate responses. These are enqueued as **Events** (status updates or data/artifact chunks). Meanwhile, an asynchronous consumer in the **RequestHandler** reads from the queue, updating a persistent **Task** object with each event. Once the task completes, the final Task state, especially the artifact, is returned to the client in a JSON response. If the client requested streaming, the server keeps the HTTP connection open and sends each event immediately via Server-Sent Events (SSE).



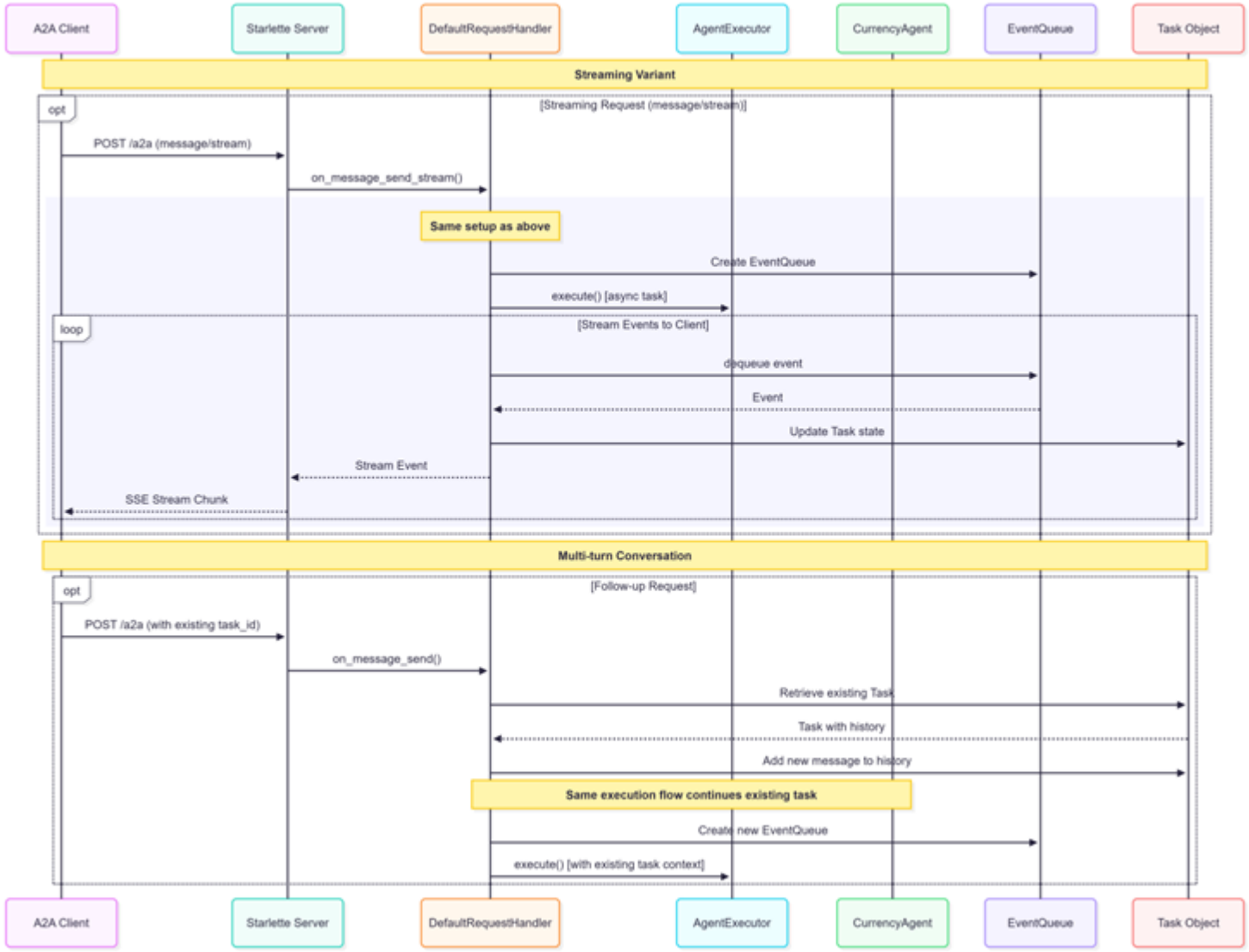
Task and Event

In A2A, every interaction (from user query up to the result returned that may involve single-turn or multi-turn conversations) is framed as a **Task** (the stateful unit of work) with a well-defined lifecycle. When **AgentExecutor** receives a request for the first time, it creates a Task object indicating the start of the task, and enqueues it into the **EventQueue**, then invoking the agent's execution interface by passing the request to the agent . As the agent produces response, the executor wraps each piece of the response data into an event in one of two event types: a **TaskStatusUpdateEvent** (to signal progress or state changes, e.g. "working", "input-required", "completed") or a **TaskArtifactUpdateEvent** (to deliver a chunk of generated data/artifact). These events are then enqueued.

Meanwhile, the **RequestHandler**'s event consumer dequeues events from the queue, updates the **Task** state and append messages or artifacts. Once a final event is received (e.g. status = *COMPLETED*), the handler concludes the flow: it awaits any remaining async tasks, removes the Task from its running set, and returns the Task object (with full history and outputs) to the client after encapsulating the data into structured A2A messages according to the communication protocol, e.g., JSON-RPC.

Streaming

Multi-turn



Streaming

A feature of A2A is **streaming of incremental results**. When a client needs immediate feedback (e.g. a long document or continuous updates), it uses the message/stream RPC method. In this mode, the server responds with Content-Type: text/event-stream and pushes every event as an SSE packet. Each SSE “chunk” carries a JSON-RPC result (matching the original request ID) that contains a Task or update Event. For example, a TaskStatusUpdateEvent may include an interim message (“Analyzing data…”) or mark the task as final, while TaskArtifactUpdateEvents carry payload chunks to be reassembled. The SSE connection remains open until the agent signals completion (final: true). Clients can then consume these events in real time, allowing low-latency UI updates or incremental processing. If the connection drops prematurely, clients may later call the tasks/resubscribe method to catch up on missed events.

Notification and Multi-turn

For clients that cannot stay connected (e.g. mobile apps, or in a long-time task), A2A also supports **push notifications**: the server can send an HTTP POST to a client-provided webhook when a task reaches a significant state (completed or awaiting input). The client then uses tasks/get to retrieve the final Task.

A2A supports **multi-turn conversations**. If an agent processing a Task enters an "input_required" state, it will emit a status event with input_required: true. The client can then retrieve the task_id from event, and send a follow-up request including the task_id indicating it as a follow-up request of the same Task. The RequestHandler will load the saved Task (including its history), append the new message, and start a new execution cycle. In effect, the conversation resumes within the context of the same Task. This allows complex, branching interactions: each turn produces additional events that update the same Task record, preserving context across messages.

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

A2A Server (Proxy) for ADK Agent

To implement an A2A server for an ADK agent, it is basically to **implement the AgentExecutor using the agent runner**, where the **agent Event is converted to A2A server Event**.

- Google has implemented a version adk-python\src\google\adk\a2a\executor\a2a_agent_executor.py

- to_a2a() in agent_to_a2a.py creates an A2AStarletteApplication that creates an agent runner instance for the specified agent, and passes the runner to the AgentExecutor for the A2A server to call.

a2a_app = to_a2a(root_agent, port=int(os.getenv('PORT', '8001')))

- Run it with “python file:a2a_app --host localhost --port 8000”

ADK Agent (Proxy) for A2A Server

To implement an ADK agent to wrap a remote A2A server, **just implement the Agent._run_async_impl, where the Agent interaction request/response are converted to remote A2A server's request/response**.

- Google has implement a version adk-python\src\google\adk\agents\remote_a2a_agent.py

- RemoteA2aAgent() is a custom agent that passes user request to A2A server request, and converts the response to agent event and returns it (yield).

```
remote_agent = RemoteA2aAgent(
    name="hello_world_agent",
    description=("Helpful assistant that can roll dice and check if numbers are prime." ),
    agent_card=f"http://localhost:8001/{AGENT_CARD_WELL_KNOWN_PATH}",
)
```

- Run with “adk run”

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

Summary

The A2A server runtime engine embodies a **proxy pattern with event-driven state management**. Its architecture emphasizes separation between agent execution and client-visible state, asynchronous communication, and flexible task persistence.

This design achieves three critical goals:

- **Consistency** – Client sees a coherent task state regardless of agent framework.
- **Scalability** – Async event queue enables streaming, long-running tasks, and notifications.
- **Extensibility** – Framework-agnostic design makes A2A adaptable to new agent ecosystems.

Together, these principles make A2A a robust runtime engine for multi-agent collaboration in enterprise-grade environments.

Agenda

Introduction

Design Philosophy

Core Components

Workflow Overview

Proxy Examples

Summary

Appendix

Google A2A vs. Google ADK

The concepts of Google A2A and Google ADK agent has following mapping relation:

A2A: Context ⇔ **ADK: Session** (an ongoing conversational interaction with the agent, can be multi-task, multi-turn, as long as the client wants to continue.)

A2A: Task ⇔ **ADK: N/A** (from query up to finishing the query, may incur multi-turn asking for more inputs.)

A2A: N/A ⇔ **ADK: Invocation** (one turn from user query up to a text response to the user.)

Google ADK does not have the Task concept – this is a little weird design decision, since whether a task a complete (no more input is needed) is decided by the agent. Google A2A introduces Task concept based on Agent's response that has a status value to indicate the task completion.

When an agent responds with a message to user asking for further input, it will be marked as **final response.** For A2A server, this is not a terminal state of a task, but an interrupted state that can be resumed.