Agent2Agent (A2A) Protocol by Michaël Bettan

01. Definition & Scope

What is A2A?

The **Agent2Agent (A2A)** protocol is an open, vendor-neutral standard initiated by Google with 50+ enterprise partners and governed by the **Linux Foundation**. It defines how autonomous agents **discover capabilities**, **negotiate modalities**, **delegate stateful tasks**, **stream incremental artifacts**, and **enforce enterprise authentication** across organizational, cloud, and framework boundaries.

> **WHAT A2A SOLVES — THE MULTI-AGENT SILO PROBLEM**
>
> Enterprise departments inevitably build specialized agents on heterogeneous stacks (Google ADK on Agent Runtime, LangGraph on GKE, CrewAI on Cloud Run, SaaS agents inside Workday, ServiceNow, Salesforce, or SAP). Without a wire-level peer protocol, integrating *N* agents requires brittle custom glue adapters and exposes internal prompts or memory structures. A2A lets **opaque agents** collaborate as peers without sharing internal code, prompts, session databases, or model providers.

MCP vs. A2A: Vertical Tools vs. Horizontal Peers

MCP and **A2A** are **complementary, orthogonal layers** — never mutually exclusive replacements:

Dimension	Model Context Protocol (MCP)	Agent2Agent Protocol (A2A)
Topology	**Vertical** (`Agent ↔ Tool / Resource / Database`)	**Horizontal** (`Client Agent ↔ Remote Specialist Agent`)
Target abstraction	Deterministic, schema-bound functions (`tools/call`), static/dynamic `resources` , prompt templates	Autonomous, opaque cognitive peers that plan, reason, ask follow-up questions, and run multi-step workflows
State & duration	Primarily stateless/transactional RPC per tool invocation	**Stateful** `Task` **lifecycle** (`submitted` → `working` → `input-required` → `completed`) spanning seconds to days
Discovery artifact	`tools/list` JSON Schema over stdio or Streamable HTTP	`AgentCard` at `/.well-known/agent-card.json` (plus authenticated extended cards & enterprise registries)

> **CANONICAL PRODUCTION PATTERN**
>
> A front-door orchestrator in **Gemini Enterprise** delegates domain goals over **A2A** to specialized department agents (Finance, Legal, SecOps), and each specialist agent invokes its own private databases and SaaS APIs over **MCP**.

02. Core & Wire Protocol Architecture (v0.3 → v1.0)

Three Architectural Roles

- **User (Human or Trigger):** Initiates the high-level objective and holds primary identity/authorization context.
- **A2A Client (Client / Host Agent):** Decomposes the user goal, inspects discovered `AgentCard` skills, dispatches tasks, handles `input-required` / `auth-required` interruptions, and synthesizes multi-agent outputs.
- **A2A Server (Remote / Opaque Agent):** Exposes an HTTPS A2A endpoint, executes tasks autonomously using its own private tools/memory/LLM, and emits `Task` state updates and `Artifact` outputs **without exposing its internal chain-of-thought, system prompt, or tool credentials**.

Three-Layer Protocol Stack (`a2a.proto`)

1. **Canonical Data Model:** Protobuf-defined domain objects (`Agent Card` , `Task` , `Message` , `Part` [`TextPart` , `DataPart` , `FilePart`], `Artifact` , `TaskStatusUpdateEvent` , `TaskArtifactUpdateEvent`) that neutralize framework mismatch.
2. **Abstract Operations (v1.0 PascalCase vs. v0.3 slash names):** - `SendMessage` *(v0.3:* `message/send` *)* — synchronous response or initial async `Task` creation. - `SendStreamingMessage` *(v0.3:* `message/stream` *)* — opens an SSE stream of status + artifact chunks. - `GetTask` *(v0.3:* `tasks/get` *)* — polls current `Task` status and artifacts by `taskId` . - `CancelTask` *(v0.3:* `tasks/cancel` *)* — requests graceful cancellation of an in-flight `Task` . - `SubscribeTo Task` *(v0.3:* `tasks/resubscribe` *)* — **reconnects an SSE stream** to an existing in-flight `Task` after client network drop without restarting work. - `GetExtendedAgentCard` *(REST:* `GET /extended AgentCard` *)* — fetches privileged/internal skills after authentication.
3. **Protocol Bindings (`supportedInterfaces`):** A2A v1.0 supports three official wire bindings declared in `AgentCard.supportedInterfaces[]` (`url` + `protocolBinding`): `JSONRPC` (JSON-RPC 2.0 over HTTPS), `HTTP+JSON` (RESTful HTTP), and `GRPC` (high-throughput binary gRPC).

03. Standard Task Lifecycle & State Machine

Every delegated unit of work is tracked via two distinct identifiers:

- `contextId` : Groups multiple related `Task` objects and follow-up turns into one logical multi-turn conversation thread.
- `taskId` : Uniquely identifies one specific execution lifecycle through three state categories:

State Category	v1.0 Protobuf Enum *(v0.3 JSON)*	Wire & Client Behavior
Active (Non-Terminal)	`TASK_STATE_SUBMITTED` (`submitted`), `TASK_STATE_WORKING` (`working`)	Task is queued or actively running; `/message:stream` (`SendStreamingMessage`) SSE streams stay open; polling continues
Interrupted (Paused)	`TASK_STATE_INPUT_REQUIRED` (`input-required`), `TASK_STATE_AUTH_REQUIRED` (`auth-required`)	**SSE stream closes cleanly**; client must either prompt the human/orchestrator for missing slot data (`TASK_STATE_INPUT_REQUIRED` using the same `taskId` + `contextId`) or complete an out-of-band OAuth 3LO / step-up consent flow (`TASK_STATE_AUTH_REQUIRED`) before resuming
Terminal (Final)	`TASK_STATE_COMPLETED` (`completed`), `TASK_STATE_FAILED` (`failed`), `TASK_STATE_CANCELED` (`canceled`), `TASK_STATE_REJECTED` (`rejected`)	Execution is permanently finished (`TASK_STATE_REJECTED` indicates the remote agent declined the task based on policy or capability); further mutations require creating a new `Task` (optionally sharing `contextId`)

- **`Message` vs. `Artifact` & `gs://` URI handoff:** A `Message` carries conversational turns, instructions, and clarification prompts between client and server (`role: "user"` or `"agent"`), whereas an `Artifact` represents a durable task deliverable (generated report, SQL extract, PDF, or structured JSON). Never base64-encode multi-megabyte PDFs, CSVs, or media inside inline `Part` / `Artifact` JSON payloads (which bloat JSON-RPC serialization and choke **Agent Gateway** `CONTENT_AUTHZ` stream inspection); persist binary blobs in Cloud Storage via `GcsArtifactService` and pass `gs://` **URIs** in `FilePart` (`fileWithUri`).

Standard 5-step execution flow:

1. **Capability Discovery:** Fetch `/.well-known/agent-card.json` (or query **Agent Registry**).
2. **Skill & Modality Matching:** Match goal against `skills[]`, `defaultInputModes` / `defaultOutputModes` (MIME types), and `securityRequirements`.
3. **Dispatch:** Call `SendMessage` or `SendStreamingMessage` (`/message:stream`) with typed `Part` payloads (`TextPart`, `DataPart`, `FilePart` with `gs://` URI) and HTTP `Authorization` headers.
4. **Incremental Collaboration:** Receive `TaskStatusUpdateEvent` transitions and chunked `TaskArtifactUpdateEvent` outputs (or handle `TASK_STATE_INPUT_REQUIRED` / `TASK_STATE_AUTH_REQUIRED` pauses).
5. **Artifact Consumption:** Pass completed `Artifact` outputs downstream (artifact chaining) or render to the user.

04. Transports, Discovery & Agent Cards

Selecting the Right Transport Pattern

Pattern	A2A Operation(s)	Task Duration	When to Choose
1. Synchronous Request/Response	`SendMessage`	`< 5-15 seconds`	Fast deterministic lookups or single-step specialist replies
2. Streaming Updates (SSE)	`SendStreamingMessage` (`/message:stream`, + `SubscribeToTask` on drop)	`5 seconds – several minutes`	Interactive UX needing live progress stages and incremental chunked artifacts (`append: false/true`, `lastChunk: true`)
3. Asynchronous Polling	`SendMessage` → periodic `GetTask`	`Minutes` (or environments blocking SSE/inbound webhooks)	Batch synthesis where client sits behind strict NAT/egress-only firewalls
4. Push Notifications (Webhooks)	`SendMessage` + `TaskPushNotificationConfig` (`CreateTaskPushNotificationConfig`)	`Minutes to hours / days`	Deep research, human approval queues, or batch workflows where holding open HTTP/SSE connections will time out

> **EXAM TRAP**
> Never use synchronous `SendMessage` without webhooks or polling for a multi-hour workflow: load balancers and Cloud Run (`60 min` max HTTP timeout) will sever the connection and orphan client state. For `TaskPushNotificationConfig` webhooks, validate callback domain ownership and verify cryptographic **HMAC / JWT signatures** + replay nonces on incoming callbacks so attackers cannot spoof fake `TASK_STATE_COMPLETED` events!

Agent Cards & Four Discovery Patterns

An `AgentCard` advertises: identity (`name`, `description`, `version`, `provider`), endpoints (`supportedInterfaces` with `url` + `protocolBinding`), `capabilities` (`streaming`, `pushNotifications`, `extendedAgentCard`, `extensions`), `securitySchemes` + `securityRequirements` (OpenAPI-style OAuth2, OIDC, API key, mTLS), `defaultInputModes` / `defaultOutputModes`, and `skills[]` (`id`, `name`, `description`, `tags`, `examples`, per-skill MIME modes).

1. **Well-Known URI (RFC 8615):** `GET https://<domain>/.well-known/agent-card.json` for open/federated discovery. *(Version trap: early drafts showed `agent.json` and `tasks/send`; v0.3 standardized `agent-card.json` with `message/send`; v1.0 keeps `agent-card.json` and uses PascalCase `SendMessage` / `GetExtendedAgentCard`.)*
2. **Curated Enterprise Registry (**Google Cloud Agent Registry**):** Central governed catalog with IAM visibility controls, lifecycle snapshots, and security posture metadata — **the enterprise default** to prevent rogue agent sprawl.

3. **Authenticated Extended Agent Card:** Public card exposes safe metadata + `capabilities.extendedAgentCard: true` *(v0.3: `supportsAuthenticatedExtendedCard`)*; authenticated callers invoke `GetExtendedAgentCard` to unlock sensitive internal skills.
4. **Direct Configuration:** Pinned card URLs/configs injected via Terraform/env vars for tightly coupled internal microservices.

05. Intra-App `AgentTool` vs. Inter-App `A2A` Decision Matrix

One of the highest-frequency **Professional Agentic Architect (PAA)** design questions is deciding when to compose specialists **inside one application** (ADK `sub_agents` transfer or `AgentTool` / agent-as-a-tool) vs. **across a network boundary with A2A**:

Architectural Dimension	Intra-App (`sub_agents` / `AgentTool`)	Inter-App (`A2A Protocol`)
Deployment & team boundary	Single codebase, single repo/CI pipeline, same runtime stack (e.g., all Python ADK in one Agent Runtime service)	Separate repositories, independent release cadences, **polyglot frameworks** (ADK ↔ LangGraph ↔ CrewAI ↔ SaaS vendor) or cross-org partners
Session & memory ownership	**Shared `InvocationContext` & `session.state`** in-process (`sub_agents` share full conversation history; `AgentTool` runs a scoped child call in-process)	**Strict state opacity:** remote agent owns an isolated session store; sees **only** the explicit `Message` parts sent for that `contextId` / `taskId`
IAM & blast-radius isolation	Runs under the **same container & service/agent identity** — one compromised dependency affects the whole agent	**Independent security boundary:** distinct **SPIFFE Agent Identity**, separate VPC-SC perimeter, independent IAM roles, mTLS, and OAuth token exchange
Latency & serialization cost	Zero network hop (in-memory Python/Go call); lowest TTFT	HTTPS/gRPC serialization + auth verification + independent remote reasoning loop
Live audio / modality boundary	Can keep native streaming context inside one process where supported	**Modality normalization boundary:** live voice front-doors (e.g., **CX Agent Studio** or **Gemini 3.8 Live** WebSockets) terminate real-time audio locally and delegate normalized `TextPart` / `DataPart` / `FilePart` payloads over A2A

Decision rule: Start **in-process (`sub_agents` / `AgentTool`)** when one team owns tightly coupled steps sharing the same data permissions. Cross into **A2A** the moment you need **independent scaling/deployment, polyglot frameworks, third-party SaaS agents, or least-privilege IAM / data-perimeter isolation** (e.g., preventing a customer-support front-door agent from directly holding HR or PCI database credentials).

06. The 5-Layer Agentic Interoperability Ecosystem

Modern production architectures combine five specialized open protocols — **know what each layer owns so you never force one protocol to do another's job**:

Layer	Protocol(s)	Direction & Payload	Architectural Purpose
1. Agent ↔ Tool / Data	**MCP** & **OpenAPI**	Agent → Server (`tools/call`, JSON Schema)	Deterministic invocation of databases, APIs, files, and SaaS connectors
2. Agent ↔ Agent	**A2A**	Client Agent ↔ Remote Agent (`Task`, `Message`, `Artifact`)	Stateful horizontal delegation, discovery, and multi-turn collaboration across opaque peers
3. Agent ↔ User Generative UI	**A2UI (v0.8 / v0.9)**, **AG-UI**, **MCP UI**	Agent/Tool → Host Client UI (declarative JSON / event stream)	Safe, interactive rich UI rendering (forms, tables, approval cards, charts) instead of walls of Markdown text
4. Real-Time Multimodal Stream	**Gemini Live API** (WebRTC / WSS)	Human ↔ Voice/Video Front-Door (PCM audio + video frames)	Sub-second full-duplex speech & camera interaction with server-side VAD barge-in
5. Agent ↔ Commerce / Money	**AP2 (Agent Payments Protocol)** & **x402**	User → Agent → Merchant / API (`Mandates` / HTTP `402`)	Cryptographically verifiable human-authorized purchases (**AP2**) and frictionless M2M micropayments (**x402**)

Deep Dive: Generative UI Protocols (`MCP UI` vs. `AG-UI` vs. `A2UI`)

Why not let a remote agent return raw HTML/JavaScript `<iframe>` bundles? Arbitrary remote scripts create **Cross-Site Scripting (XSS)**, credential theft, and jarring brand mismatches. Three complementary UI standards address different layers:

- **MCP UI:** Allows an individual **MCP tool** to return an embedded, sandboxed UI resource alongside tool data (e.g., an interactive seat-map picker returned directly by a flight-search MCP tool).

- **AG-UI (Agent-User Interaction Protocol):** Standardizes the **bidirectional event stream** between a frontend client app and its backend agent — synchronizing streaming text, live tool-call indicators, shared frontend/agent state diffs, and human-in-the-loop interrupts.
- **A2UI (Agent-to-UI, v0.8 / v0.9 — Native in Gemini Enterprise):** Open protocol created by Google for **declarative, streaming JSON UI over A2A**:
- **How it works:** The host surface (such as **Gemini Enterprise**, web, Flutter, or mobile) advertises its **client-approved Component Catalog** (standard basic components, Material Design, or custom enterprise widgets). The remote A2A agent streams flat, LLM-friendly **declarative JSON component descriptions + data bindings** inside A2A `DataPart` payloads (`a2ui` extension).
- **Security & UX win:** The client renders native, accessible, brand-consistent widgets from its own trusted codebase — **zero arbitrary JS execution (XSS-safe by design)** — even when an orchestrator composes UI cards from multiple cross-vendor A2A sub-agents in one chat!

Deep Dive: Agentic Commerce (`AP2` & `x402`)

When an autonomous agent books travel, procures cloud SaaS, or pays per API call, traditional checkout flows break because a human is not clicking *"Buy Now"* on the merchant's page:

- **AP2 (Agent Payments Protocol):** Open extension over **A2A** and **MCP** backed by major payment networks that solves **authorization, authenticity, and non-repudiation** via verifiable cryptographic credentials (**Mandates**): 1. **Intent Mandate:** Signed upfront by the user when delegating bounded authority (e.g., *"Buy two concert tickets if total `< $300` before Friday"* — enabling safe **human-not-present** execution). 2. **Cart Mandate:** Cryptographically signs the exact final merchant SKU list, price, and shipping terms (**human-present** confirmation or verified match against the Intent Mandate), creating an immutable audit trail across user, agent, merchant, and issuer for dispute resolution.
- **x402 Protocol:** Revives the HTTP `402 Payment Required` status code for **autonomous machine-to-machine (M2M) micropayments**. When an agent calls a metered API, dataset, or remote compute service without a pre-negotiated subscription, the server responds `402` with machine-readable payment terms in headers; the agent signs a programmatic micropayment (e.g., stablecoin/ledger settlement within policy caps) and retries automatically with proof-of-payment headers.

07. Collaborative Topologies, Google Cloud Implementation & Security

Multi-Agent Workflow Patterns over A2A

- **Hierarchical Supervisor / Router:** A front-door orchestrator (e.g., **Gemini Enterprise** or **CX Agent Studio**) discovers specialist `Agent Card` skills and delegates sub-tasks sequentially or in parallel, synthesizing returned `Artifact` parts.
- **Critic-Reviewer (Generator-Verifier):** A producer agent drafts code, SQL, or legal text; an independent A2A reviewer agent (with distinct prompts, model family, and read-only policy tools) audits accuracy and compliance before finalization.
- **Structured Debate & Consensus:** Multiple domain peers (e.g., Security, FinOps, SRE agents) evaluate a change proposal from orthogonal corpora before an adjudicator synthesizes a risk-weighted recommendation.

Implementing A2A on Google Cloud (ADK, Agent Runtime, Gemini Enterprise & CX Agent Studio)

- **Exposing an ADK Agent (`to_a2a()` & `A2aAgentExecutor`):** Wrap any ADK agent with `to_a2a(root_agent, port=8001)` (which wires `AgentCard` generation at `/.well-known/agent-card.json`, `A2aAgent Executor`, task state tracking, and Starlette/ASGI routes) and deploy to **Cloud Run** (`--functional-type=agent --identity-type=agent-identity`), **GKE**, or natively onto **Agent Runtime** via `A2aAgent` (**Preview**).
- **Dynamic Discovery via `AgentRegistry` & `.clone()` Single-Parent Rule:** Instead of hardcoding fragile peer URLs, resolve governed remote specialists once at process startup via `AgentRegistry.get_remote_a2a_agent()`. Because every ADK agent instance enforces a strict **single-parent rule** (raising `ValueError` if one `RemoteA2a Agent` object is attached to two parent orchestrators or workflow branches), call `.clone()` whenever reusing a discovered peer across multiple branches:

```
registry = AgentRegistry(project_id="gov-proj", location="global")
legal_peer = registry.get_remote_a2a_agent("agents/legal-specialist", httpx_client=authed_client)

intake_flow = LlmAgent(name="intake", model="gemini-3.8-flash", sub_agents=[legal_peer])
renewal_flow = LlmAgent(name="renewal", model="gemini-3.8-flash", sub_agents=[legal_peer.clone()])
```

- **Gemini Enterprise & CX Agent Studio:** Register third-party or custom ADK A2A agents directly into **Gemini Enterprise** (supporting OAuth consent brokering and **A2UI** rich cards) or delegate complex backend tasks from **CX Agent Studio** conversational/voice flows over A2A.

Enterprise Security & A2AOps Hardening

- **Header-Only Credentials:** Pass OAuth 2.0 bearer tokens, OIDC JWTs, or API keys **strictly in HTTP `Authorization` headers**, never in URL query strings (logged in plaintext by proxies/load balancers) and never inside `Message` / `Part` payloads (persisted in task logs and echoed into LLM prompts).

- **Scoped Token Exchange (Prevent Confused Deputy):** Never forward a broad user uber-token blindly down an A2A chain. Use **OAuth 2.0 Token Exchange (RFC 8693)**, **Agent Identity** (`spiffe://…`), and **Agent Gateway** (`REQUEST_AUTHZ` + `CONTENT_AUTHZ`) to down-scope tokens per target audience, paired with **Mutual TLS (mTLS)** and **VPC Service Controls**.
- **Prompt / Task Injection Across Peers:** Treat every incoming `Message` and returned `Artifact` from a remote A2A agent as **untrusted external input**: screen cross-boundary payloads with **Vertex AI Model Armor** and enforce **Task Depth / Hop Limits** + token budgets to block infinite delegation loops.
- **Distributed Tracing (`traceparent`):** Propagate W3C Trace Context (`traceparent` / `tracestate`) across every A2A HTTP/gRPC hop so **Cloud Trace** reconstructs a single end-to-end waterfall across client and remote agents.

08. Glossary

- **AgentCard:** JSON capability, endpoint (`supportedInterfaces`), skill, modality, and security manifest published at `/.well-known/agent-card.json` (and `GetExtendedAgentCard`).
- **Task / `taskId` / `contextId`:** Stateful unit of delegated A2A work (`taskId`) correlated across multi-turn threads by `contextId` across `TASK_STATE_SUBMITTED`, `TASK_STATE_WORKING`, `TASK_STATE_INPUT_REQUIRED`, `TASK_STATE_AUTH_REQUIRED`, and terminal states.
- **`Part` (`TextPart`, `DataPart`, `FilePart`) & `Artifact`:** Typed content blocks inside conversational `Message` turns and durable task deliverables (using `gs://` URIs via `GcsArtifactService` for large binaries).
- **`SubscribeToTask` & `TaskPushNotificationConfig`:** A2A v1.0 operations to re-attach an SSE stream after disconnection (`SubscribeToTask`) or deliver signed asynchronous webhook updates (`CreateTaskPushNotificationConfig`).
- **`AgentRegistry.get_remote_a2a_agent()` & `.clone()`:** ADK startup discovery client resolving governed A2A peers from Agent Registry and cloning instances to satisfy ADK's single-parent rule.
- **A2UI (v0.8 / v0.9):** Google's declarative JSON UI protocol over A2A rendering native widgets from a client-approved component catalog (natively supported in **Gemini Enterprise**).
- **AG-UI & MCP UI:** Client-agent bidirectional state/event streaming protocol (**AG-UI**) and tool-level embedded UI views (**MCP UI**).
- **AP2 (Agent Payments Protocol):** Open commerce protocol using cryptographically signed **Intent Mandates** and **Cart Mandates** over A2A/MCP for verifiable agent purchases.
- **x402:** HTTP `402 Payment Required` protocol for autonomous machine-to-machine API/resource micropayments.