A2X: Agent-to-Everything

A Unified Cloud-Native Agent Protocol

Abstract

Modern AI deployments increasingly involve **intelligent agents** that must use tools, exchange data, and collaborate with other agents in real time. Existing standards like Model Context Protocol (MCP) and Agent-to-Agent (A2A) address pieces of this puzzle – MCP connects AI to external data/tools, and A2A links agents to each other – but each has limitations in isolation. This whitepaper presents **A2X (Agent-to-Everything)**, a unified communication protocol that integrates the strengths of MCP and A2A while filling critical gaps such as persistent memory, human oversight, multi-agent orchestration, and environment integration. We detail the A2X architecture's **core message primitives**, session and security model, and extensibility mechanisms, and we illustrate how A2X enables complex workflows that were previously difficult to implement safely or at scale. By standardizing how agents **communicate with everything** (tools, other agents, people, and IoT/physical environments), A2X aims to provide a foundational infrastructure for the next generation of AI systems. Endnotes are provided for reference to relevant prior work and emerging standards.

Introduction and Background

Intelligent software agents powered by AI are rapidly gaining adoption in domains ranging from digital assistants to autonomous business processes. To function effectively, an agent must be able to **access external tools and data** as well as **coordinate with other agents**. Recognizing this, the AI community has introduced open protocols to standardize agent interactions. Two notable developments in late 2024 and 2025 were **Anthropic's Model Context Protocol (MCP)** and **Google's Agent-to-Agent (A2A)**:

- **Model Context Protocol (MCP)** – MCP is an open standard (introduced by Anthropic in Nov 2024) for connecting AI models to external data sources and tools. It provides a mechanism for an AI assistant (typically a large language model) to **inject contextual information** (files, database results, etc.) into its prompts and to **invoke tools or APIs** in a structured way during a session. In essence, MCP acts like a "**USB-C for AI**" – a universal port through which an AI can interface with various resources. MCP sessions often use JSON-RPC over a persistent connection (e.g. server-sent events), allowing the model to maintain stateful dialogs with a tool service. This

enables richer interactions than one-off API calls: for example, an AI agent can retrieve data, get follow-up clarifications from the tool, and so forth, all in one session.

- **Agent-to-Agent Protocol (A2A)** – A2A, announced by Google in April 2025, is an open protocol for **interoperability between autonomous agents**. It defines standard message formats and endpoints (built on HTTP/HTTPS + JSON and Server-Sent Events) so that agents from different vendors or frameworks can discover each other, exchange messages, and delegate tasks. In practice, A2A treats agents as web services – each agent exposes an API (with a schema called an "Agent Card" describing its capabilities) and can be called by other agents via HTTP requests, with streaming responses for long-running tasks. Google positions A2A as complementary to MCP: where MCP connects an agent to tools and data, A2A connects agents to other agents to enable multi-agent workflows. For example, in a car repair scenario one could use MCP to let a mechanic's AI agent execute a *tool action* like lifting a car on a hydraulic platform, while A2A would facilitate a *conversation* between the customer's agent and the mechanic's agent to troubleshoot the issue (e.g. exchanging messages: "The car is making a rattling noise" → "Send me a picture of the left wheel"). By combining these, an agent could both converse with a peer and invoke physical actions as needed.

Limitations of Current Protocols: While MCP and A2A each address important aspects of agent integration, using them in isolation (or even together) reveals significant gaps. Industry analyses warn of a potential *"tug of war"* if developers must juggle multiple disparate agent protocols, noting that teams have limited bandwidth to support many ecosystems simultaneously. Some of the key limitations and unmet needs are outlined below:

- **No Long-Term Memory or Cross-Session State:** MCP maintains context only within a single session – once an interaction ends, the agent "forgets" everything unless manually saved externally. User preferences or learned information do not persist across independent sessions or applications. Likewise, A2A tasks are typically stateless transactions; the protocol itself does not preserve dialogue history or agent state beyond each task request. This lack of built-in memory means **agents cannot personalize or continuously learn** over time without custom memory solutions. In real-world use, this leads to repetitive instructions and fragmented user experiences (each new session starts from scratch).
- **Session Boundaries and Context Management:** Because MCP relies on sending lots of context with each tool invocation, sessions can suffer from **context bloat** – conversation history and data accumulate without a clear mechanism for pruning or summarization. There is no standard way to delineate what context is ephemeral versus what should persist; developers must manually manage when to reset or update the context to avoid stale information or overflow of the model's context

window. A2A, being stateless per request, implicitly avoids long context buildup, but at the cost of continuity – multi-turn dialogues have to be re-established or encoded in each request. Neither protocol provides an elegant solution for **maintaining just the right amount of context** over prolonged interactions.

- **Tool Integration Overhead and Discovery:** MCP requires each external tool or API to be wrapped behind an MCP-compatible server (essentially an adapter exposing a JSON-RPC interface). This standardizes access, but still means a **lot of manual work** to onboard new tools – every integration needs a custom connector. Agents only know about the tools that have been explicitly connected by developers; there is **no dynamic discovery** of new capabilities at runtime. A2A similarly assumes agents know each other's endpoints or are listed in a registry; it has an "Agent Card" for capability metadata, but no global lookup beyond what is configured by the developer. In short, today's agents operate in a somewhat static tool environment – they **cannot spontaneously find or use new APIs** unless those are pre-wrapped or pre-registered. This hampers adaptability in fast-changing ecosystems.
- **Missing Human Oversight Mechanism:** Neither MCP nor A2A defines a standard way for a **human to be in the loop** of an agent's decisions. MCP focuses on agent↔tool communication, and A2A is strictly agent↔agent. If an autonomous agent needs confirmation or clarification from a human, there is no protocol-level message for that – developers must implement ad-hoc channels (e.g. a GUI prompt, an email) to involve humans. Recent commentary has pointed out that this lack of an *Agent-to-Human (A2H)* interaction model is a "missing link" in autonomous AI deployments. Without a standardized oversight loop, it's difficult to ensure safety for high-stakes tasks – e.g. preventing an AI from executing a sensitive action without human approval.
- **No Standard for Environment Integration (Physical/IoT):** Agents are increasingly interfacing not just with software APIs but with **physical devices and environments** (IoT sensors, robots, simulators). MCP treats such interfaces as just another tool API, lacking any specialized support for continuous sensor data streams or real-time control signals. A2A, being HTTP-based, isn't optimized for low-latency local networks or streaming sensor updates. There is currently no "Agent-to-Environment" (A2E) protocol for an agent to subscribe to environment events or safely control actuators. This is a gap for use cases like robotics, smart infrastructure, or augmented reality, where an agent needs a **persistent bidirectional link with its environment** (for telemetry, alerts, commands). Efforts like IBM's **Agent Communication Protocol (ACP)** have arisen to explore on-premise or edge-focused agent communication, underscoring that cloud-centric standards (MCP, A2A) alone don't cover all scenarios.
- **Multi-Agent Coordination and Trust:** While A2A enables basic agent-to-agent calls, it is limited to pairwise task delegation. There is no built-in framework for

coordinating multiple agents working together on a shared goal or maintaining a shared state. Orchestrating a team of agents (with roles like planner, executor, verifier) requires custom logic outside the protocol. Moreover, **security and governance** in multi-agent settings are not fully solved by A2A's authentication alone – once an agent accepts a task, there's no standard way to enforce scope or constraints on what it does with that task's context. Without a richer coordination protocol, scenarios like an agent chain-of-command or policy-limited delegation (e.g. "Agent A can call Agent B but *only* to do X and not share data Y") are hard to implement. Early multi-agent experiments have noted difficulties in debugging and controlling such systems. In short, current protocols lack the means to express **multi-agent workflows, group dialogues, or fine-grained trust** between agents.

- **Adaptation and Continuous Learning:** Modern AI agents could theoretically improve their performance over time by learning from interactions (feedback, mistakes, new information). However, neither MCP nor A2A provides support for an agent to **learn on the fly**. Any model fine-tuning or knowledge update has to be done out-of-band. There's no message type for an agent to, say, update its knowledge base or incorporate a user correction persistently. Similarly, **safety oversight** is static – once a prompt is sent, it's up to the application to filter outputs or intervene. No standard mechanism exists for an automated feedback cycle (like an agent analyzing its own output or another agent critiquing it). This means opportunities for **in-context learning and self-correction** are missed. Recent research by Amazon and others suggests that agents collaborating and critiquing each other can significantly enhance reasoning accuracy, but without protocol support, such patterns must be implemented in a bespoke manner.

These limitations highlight the need for a more **comprehensive communication layer** that spans all aspects of agent interaction. In fact, multiple analysts have called for a unified approach that could combine the functionalities of MCP, A2A, and ACP into one framework. In the next section, we introduce the Agent-To-Everything (A2X) protocol as a response to these gaps, aiming to provide a single **minimal yet extensible** standard that allows AI agents to seamlessly communicate with *everything* in their world.

Introducing the A2X Protocol

A2X (Agent-to-Everything) is proposed as a unified, cloud-native standard that brings together the strengths of MCP and A2A while overcoming their individual shortcomings. Instead of maintaining separate channels for agent-to-tool vs. agent-to-agent, A2X defines a **single cohesive interface** through which an agent can interact with *any* entity – be it a tool/service, another agent, a human user, or an environment. The design philosophy is to simplify the ecosystem so that developers can build agent solutions on one foundation,

"without worrying which protocol is doing what," and avoid the overhead of bridging multiple protocols. This vision aligns with expert predictions that the industry would benefit from a convergence of current standards into one umbrella protocol. DataVesper LLC leads the development of A2X as an open specification in collaboration with the AI community, ensuring that it remains vendor-neutral and widely adoptable.

A2X is **cloud-native by design** – it leverages widely adopted web technologies for transport and interoperability, rather than reinventing the wheel. Implementations may use HTTPS+JSON (like a typical web API) or persistent WebSocket connections for realtime communication, and integrate with cloud identity services for authentication. However, A2X's *protocol semantics* are transport-agnostic – the same message types could be carried over alternative channels (e.g. a high-speed local bus or IoT protocol) if needed, as long as both sides understand the format. The goal is to make A2X easily deployable on modern infrastructure (Kubernetes, serverless, etc.) while also adaptable to on-premise or edge environments (where lightweight or offline operation is required).

Design Principles and Architecture

At its core, A2X defines a small set of **unified message primitives** and a session-based communication model that together support all major interaction patterns. The protocol is designed with extensibility and security in mind, so it can scale from simple use cases (single agent using one tool) to complex ones (dozens of agents collaborating across cloud and edge). Key features and innovations in A2X are summarized below:

- **Unified Minimal Primitives:** Rather than having one protocol for "tools" and another for "agents," A2X uses a single message-passing model for every interaction. It defines a few fundamental message types that cover the spectrum of agent communication. These core types include:
    - **Message** – a general-purpose communication or prompt from one entity to another. This could represent a user query to an agent, an agent's reply, or an agent-to-agent question/request. It is essentially an open-ended dialog message, supporting **multi-turn conversations** in a continuous flow (unlike HTTP request/response which closes after one exchange).
    - **Action** – a request to perform an operation or invoke a capability. This covers what MCP calls a tool/function call or what A2A calls a task delegation. An Action message carries a structured description of the operation (e.g. "execute function X with parameters Y" or "control device Z with setting W"). It may target a tool service or another agent that can perform the action.
    - **Observation** – a conveyance of sensor data, events, or incremental results. This allows agents to receive streaming information from their environment

or tools. For example, an IoT sensor agent might send periodic Observation messages with temperature readings, or an AI service might stream partial results (tokens or progress updates) as an Observation sequence. Observations enable real-time, asynchronous data flow into the agent's context.

- **Feedback** – a structured response providing evaluation or guidance on some other message or action. This is used for oversight and learning loops. For instance, a human operator's approval or disapproval of an agent's Action would be sent as a Feedback message, or an automated policy agent might send a Feedback indicating that an output violates a rule. Feedback messages let the protocol carry **meta-communication** about the agent's behavior, enabling self-correction and human-in-the-loop control.

All A2X messages share a common envelope with fields such as a unique message ID, timestamp, sender and recipient IDs, message type (one of the above), an optional reference to a context or session, and a payload. The payload can be JSON (for readability) or a more efficient binary encoding, and its schema varies slightly by message type. These few primitives, in combination with the session mechanism, are powerful enough to express complex workflows – yet keeping the set small makes the protocol easier to implement and extend. *(Notably, the design of these message types draws on concepts from existing standards: for example, Actions generalize JSON-RPC function calls, Feedback is inspired by human approval flows in AI safety literature, and Observations align with streaming update patterns like SSE or MQTT topics. By unifying them, A2X avoids needing separate subsystems for each context.)*

- **Persistent Sessions & Channels:** A2X is inherently **session-oriented**. When an agent (or any client, human or machine) connects via A2X, it establishes a persistent channel that can host a continuous exchange of messages in both directions, without reopening new connections for each request. In practical terms, this behaves like a long-lived conversation socket – conceptually similar to a WebSocket, though the protocol does not mandate using the WebSocket transport. Once a session is established and authenticated, any number of Message, Action, Observation, or Feedback events can flow back and forth as needed. This design enables **long-lived dialogues and interactions** that span time and maintain state. For example, an agent and a user might keep a session open all day, so the agent can proactively send updates or ask questions contextually, rather than the user always initiating requests. Sessions support continuity: participants can reference earlier messages by ID, build on shared context, and avoid the overhead of re-authenticating or re-negotiating parameters each time. Importantly, sessions are not strictly bound to a single physical connection – an implementation could allow sessions to hibernate and resume (with

the same context) to handle client reconnects or load balancing, providing scalability while preserving state. A2X sessions thus combine the **statefulness of MCP** (context maintained across turns) with the **scalability of stateless services** (the infrastructure can manage idle sessions efficiently). They also naturally enable real-time push: e.g. an agent can send an Observation or Feedback spontaneously to a client it's conversing with, without waiting for a prompt.

- **Cloud-First Identity and Security:** From the ground up, A2X incorporates a robust identity and permission model to ensure secure operation in open environments. Every participant in the network – whether it's an agent, a tool service, a user client, etc. – has a unique **identifier** (like a URI or GUID) and uses strong authentication (such as OAuth tokens, API keys, or mutual TLS keys) when establishing an A2X session. The protocol supports exchanging identity information and validating credentials as part of the session handshake. Once connected, messages can carry auth context or capability tokens so that **delegation** is secure and controlled. For example, if Agent A invokes an Action on Agent B that involves accessing a database, Agent A's message can include a scoped token granting B read-only access for that specific task – B will execute the action using that token, and the receiving database service will verify it. In this way, A2X enables **secure task delegation** without requiring blind trust between agents. Every message's source is authenticated, and fine-grained access control can be enforced at the message level. The protocol also supports end-to-end encryption of message payloads when needed (ensuring that intermediaries or brokers cannot snoop on sensitive data in transit). By baking in modern security practices (enterprise-grade auth schemes, encryption, auditing hooks), A2X aims to be **"secure by default"** – addressing concerns that earlier protocols left to the application layer. This is critical in enterprise settings where agents might carry out high-impact operations; with A2X, organizations can trust that only authorized agents perform actions and that there is an audit trail of all agent communications.
- **Capability Advertisement and Discovery:** A2X tackles the problem of *dynamic discovery* by allowing agents and services to **publish descriptions of their capabilities** in a standardized format. Upon joining an A2X network (or at any time during a session), an agent can broadcast a self-descriptive **Capability Descriptor** – analogous to A2A's Agent Card but more general. This descriptor (expressed in JSON) lists what actions the agent can perform or what services it provides, along with any required parameters, authentication info, and other metadata (e.g. cost of using an API, rate limits, etc.). Other agents can consume these descriptors to learn **what tools or functions are available** in the ecosystem. In practice, this could be facilitated by a directory service: for example, in a cloud deployment, there might be a well-known registry that agents query to find others that match certain capabilities (e.g. "data-analysis" or "email-sending"). A2X defines messages for querying and publishing such

directories, though a directory could also be implemented as an A2X agent itself (responding to capability queries). The net effect is that agents can perform something akin to **service discovery** – much like microservices find each other. If a new tool comes online (say a new weather API), it can announce itself via A2X and *immediately* any authorized agent could start using it, even at runtime, without custom integration code. This is a leap from MCP's model where an agent only knows pre-integrated tools. In A2X, the ecosystem is more **open and self-configuring**. Tools and agents advertise what they can do; agents can also request "Who can do X?" and get a list of candidates to call. This capability discovery is critical for building flexible systems where the toolchain can evolve over time without breaking the agent. (For example, an organization could add a new internal service and all its A2X-enabled agents would know about it and how to call it, as long as permissions are in place.) By reducing the need for hardcoded integrations, A2X moves closer to **autonomous tool use** – agents that can truly adapt to new resources on the fly.

- **Explicit Context and Memory Artifacts:** To address the memory and context issues, A2X introduces the concept of **Context Artifacts** – persistent, sharable state objects that can be attached to sessions or passed between agents. Instead of relying purely on the AI model's internal memory, A2X makes context a first-class element of the protocol. For example, when a user starts a session with an agent, there might be a Context Artifact representing that user's profile and preferences, which the agent can load at session start. This artifact could be stored in a cloud database or memory store and referenced by an ID. Throughout the session, as new information is gathered or decisions made, the agent (or other agents) can update the artifact via special messages or include it in subsequent interactions. This **explicit external memory** approach has several benefits: it provides continuity across sessions (since the context artifact can be saved and reloaded later), it delineates **what is persistent vs. transient** (developers can clearly separate long-term knowledge from short-term context), and it allows collaborative updating (multiple agents can contribute to or consult the same context artifact if appropriate permissions are given). For instance, if one agent learns a new fact or the outcome of a task, it could emit an Observation that updates a shared "knowledge base" artifact. Another agent coming in later could read from that artifact to leverage that knowledge. By managing context in the open, A2X avoids the "hidden state" problem of LLM sessions – everything important can be captured in artifacts that are versioned and auditable. Of course, privacy controls apply: sensitive user data in an artifact can be access-controlled so only certain agents or roles can retrieve it. Overall, context artifacts give a structured way to achieve **persistent user intent and cross-session memory**, solving the MCP limitation where each app had siloed context. In A2X, an agent that interacts with a user over months can maintain a evolving profile of that user (likes, goals, interaction history)

that **travels with the user** – any new A2X agent the user interacts with could (with consent) load that profile and instantly personalize its behavior.

- **Human-in-the-Loop Oversight and Feedback:** A hallmark of A2X is its built-in support for **Agent↔Human collaboration**. Humans are treated as just another type of participant (with appropriate privileges) on the A2X network, which means there is a standard way for agents to request human input or approval and for humans to provide feedback. The protocol accomplishes this through the **Feedback** message type and an **approval workflow**. For example, suppose an agent wants to perform an action that is sensitive (sending an email to a company-wide list). The agent's Action message can be marked as "approval required" and specify a human (or group) as the approver. The recipient (e.g. an email-sending service agent) will then **pause** that action and generate an Approval Request (essentially a specialized Feedback query) to the designated human's client. The human, perhaps using a dashboard or mobile app connected via A2X, receives the request with details of the action (and perhaps the agent's justification or confidence level). The human can then respond with a Feedback message: *Approved*, *Rejected*, or *Approved with modifications*, etc., possibly with comments. The agent receives this feedback and proceeds accordingly (or aborts the action on rejection). All of this occurs through standardized message exchanges, meaning developers **don't have to custom-build a human override mechanism** – it's part of the protocol. This greatly enhances safety and trust, as high-impact AI decisions can be gated behind human judgment when appropriate. In addition, humans can inject feedback proactively: at any time, a human supervisor could send a "Feedback: adjust strategy" or "halt" message into an agent's session if they see it going awry, and the agent is expected to handle that gracefully. Because A2X treats human inputs similarly to agent messages, it is easy to log and audit them alongside everything else, providing transparency. Ultimately, A2X makes it straightforward to implement **Human-in-the-Loop AI**. As an example of the importance: we wouldn't want an autonomous agent to transfer large funds or delete critical data without a human check – and indeed prior work emphasizes making such "human confirmation" a standard part of agent workflows. By supporting asynchronous approval (the agent can continue other work while waiting) and subscriptions for oversight, A2X ensures that using humans as safety nets or collaborators does not break the flow of automation. This approach addresses the oversight gap identified in current protocols, making agents more **reliable and controllable in production**.
- **Multi-Agent Coordination and Composability:** A2X natively facilitates more complex multi-agent structures beyond one-to-one conversations. Any A2X message can have multiple recipients (allowing one agent to broadcast an update to many peers), and the protocol supports tagging messages with a **coordination context** or conversation group ID. This means agents can effectively form teams or sub-networks

for a particular project or goal. For example, if several agents need to collaborate on a workflow (say a "planner" agent, a "worker" agent, and a "reviewer" agent all working on a complex task), an orchestrator could assign them a shared context ID. Thereafter, any messages labeled with that ID are understood to pertain to that project, and all agents in the group will receive relevant broadcasts. An agent can still send direct messages or actions to a single target within the group, but the common context allows **state sharing**: e.g. a shared artifact for the project's status or a log of contributions. This lightweight mechanism acts like a virtual meeting room for agents. By simply including a context identifier, agents know which conversation or multi-agent session a message belongs to, avoiding confusion when multiple tasks are ongoing. Moreover, multi-recipient messages let an agent efficiently notify a whole set of agents at once (e.g. "All sensors, report your status now" as a single broadcast). Security rules can be applied to contexts (only agents with certain roles can join a given context), providing **controlled collaboration**. The end result is that orchestrating a multi-agent system becomes simpler – the protocol itself provides the glue for coordination, rather than needing a separate orchestration engine to route messages. This design addresses scenarios that A2A left open, like multi-step workflows with **several agents contributing in parallel** or in sequence. A2X makes it possible to implement patterns such as agent collectives voting on a solution, or a master agent dynamically recruiting helper agents for subtasks, all using consistent message semantics. In essence, it elevates multi-agent interactions to first-class status, fulfilling the promise of agents that can seamlessly "team up" when needed.

- **Support for In-Context Learning and Adaptation:** While A2X is primarily a communication protocol and not a learning algorithm, it is built to enable **continuous learning loops** by carrying the right information. The inclusion of the Feedback primitive is one example – it allows an agent to receive explicit critiques or evaluations from humans or other agents about its outputs. For instance, one could deploy a secondary "audit agent" whose sole job is to review the primary agent's decisions and send Feedback if something looks off. The primary agent can then incorporate that feedback (perhaps by adjusting its behavior or querying for more info) before finalizing an action. Over time, such feedback could be logged in the agent's context artifact ("lessons learned") and even used to update the model's knowledge via fine-tuning triggers. A2X does not directly fine-tune models, but it can integrate with training pipelines: e.g. an **Action message for model update** could be sent to a training service agent with new examples, enabling on-the-fly model improvement in a controlled way. Additionally, A2X's flexible messaging supports strategies like chain-of-thought prompting and self-reflection. Agents can share their intermediate reasoning steps as messages (which could be saved for transparency or analyzed by a referee agent). They can also request information from knowledge bases mid-task and inject that into context. In summary, A2X is **conducive to agents**

that learn and adapt within their sessions, because it standardizes the exchange of feedback and new data. This is in line with emerging approaches to boost AI robustness by having agents critique or assist each other. By making those interactions part of the normal workflow (rather than hacks on top of a chat interface), A2X can help agents improve themselves safely under oversight. Over the long term, this could lead to ecosystems of agents that not only work together but also *teach* each other and update shared knowledge in a virtuous cycle.

With these design elements, A2X aims to serve as a **universal "agent communication layer"** – akin to how TCP/IP serves as a universal networking layer. It remains minimal in concept (just a handful of message types and simple session semantics), but it is powerful in what it can express, due to the flexibility of those messages and the inclusion of mechanisms for discovery, context-sharing, and feedback. We next illustrate a concrete example of how A2X can be used in practice, to make the above concepts more tangible.

Illustrative Workflow Example

To demonstrate A2X in action, consider a scenario of a **"Smart Office Assistant"** agent deployed in a corporate environment. This agent's job is to manage meeting rooms and IT tasks, in cooperation with human managers and several specialized sub-agents. Below, we step through how such a scenario plays out with A2X, highlighting the protocol's features:

1. **Initialization and Discovery:** Upon startup, the Smart Office Assistant agent connects to the company's A2X broker (or cloud endpoint) and authenticates using its credentials. It then **registers its capabilities** – for example, it publishes that it can handle scheduling requests, control IoT devices (lights, HVAC), send emails, etc. – by sending a Capability Descriptor into the network. The assistant also queries the directory service (via A2X) to discover what other agents or services are available. It finds, for instance, an "Email Agent" (for sending emails), a "Network Monitor Agent" (oversees network health), and an "HVAC Control Service" (manages building climate). Each of these has advertised its own capabilities (the email agent can send or draft emails given content, the HVAC service can adjust temperatures and report sensor data, etc.). Now the Smart Assistant **knows about** these potential collaborators and tools, without any hardcoding – the discovery happened through A2X's dynamic registration.
2. **Persistent Human-Agent Session:** A human manager uses a dashboard interface (connected as a human client via A2X) to engage the Smart Assistant. They initiate a session with the agent – essentially opening a continuous dialog channel. Because of the prior authentication, the agent recognizes the manager's user ID and immediately loads the manager's **context artifact** (say, their preferences and a summary of

current office issues). This means the agent is instantly aware of the user's relevant history (e.g. that the manager likes brief updates and that Room 5A had a projector issue yesterday). The session is now set up for a rich interaction: the manager and agent can exchange messages freely, and the agent can also send observations or ask questions proactively during the session as things evolve.

3. **Dialogue and Contextual Queries:** The manager types a request: *"Please ensure all meeting rooms are set up for today's meetings and send a summary update to the team."* This comes into the Smart Assistant as a **Message** (from User to Agent, containing the request text). Because the session is continuous, the agent can think through the request and even ask clarifying questions if needed without losing context. In this case, the assistant sends back a Message: *"Sure. Do you want me to also check network connectivity and room temperature in those rooms?"* – seeking confirmation on whether to include IT/comfort checks. The manager replies *"Yes, good idea."* This brief back-and-forth illustrates how A2X supports a **multi-turn conversation** between user and agent, more naturally than a one-shot API call.
4. **Invoking Sub-Agents and Tools (Actions):** Now the Smart Assistant proceeds to carry out the manager's request. It needs to make sure each meeting room's equipment is set up, the network is fine, and the climate is comfortable, then compile a summary. The assistant uses **Action** messages to delegate tasks: it sends an Action to the HVAC Control agent – e.g., "Optimize climate in Rooms A, B, C by 9:00 AM" with target temperatures. It also sends an Action to the Network Monitor agent: "Run diagnostics on Wi-Fi and projector connectivity in Rooms A, B, C." Both of these actions are tagged with a **coordination context** ID like MeetingPrep:2025-06-17 so that results can be correlated. The HVAC and Network agents, upon receiving these Actions, begin their work (adjusting thermostats, running network pings, etc.). Notably, these interactions are all happening **in parallel** and asynchronously, thanks to A2X's ability to handle multiple concurrent messages and tasks. The Smart Assistant doesn't pause everything to wait – it can continue with other steps or handle additional input while those sub-tasks execute.
5. **Streaming Observations:** As the HVAC and Network agents carry out their tasks, they send back **Observation** messages to update the Smart Assistant on progress. For example, the HVAC service streams observations like "Room A temperature now 22°C (target 21°C)", "Room B AC turned on, cooling…", etc. The Network Monitor agent streams results like "Room B projector connectivity = OK, Wi-Fi latency = 120ms (higher than normal)". These Observation messages arrive in the Smart Assistant's session asynchronously as soon as they are available. Because A2X supports streaming, the assistant can **aggregate data in real-time** and react promptly if something is amiss (rather than waiting for a final report). In our scenario, suppose the network agent reports that *Room B's Wi-Fi latency is high.* The Smart Assistant recognizes this as a potential problem.

6. **Multi-Agent Coordination:** Upon noticing the network issue in Room B, the Smart Assistant decides to involve another agent – say an **IT Support Agent** that can troubleshoot network problems. Using the same coordination context (MeetingPrep:2025-06-17), the assistant sends a **Message** to the IT Support agent: "We have high network latency in Room B's access point. Please investigate and resolve if possible." This is a direct agent-to-agent communication, enabled by the fact that all these agents speak A2X. The IT Support agent joins the context and replies (Message): "Acknowledged, checking the router now." It may then itself send further Actions to network diagnostic tools or to reboot hardware. Throughout, because the context ID is shared, **all relevant agents (Smart Assistant, Network Monitor, IT Support)** remain on the same page about what issue is being discussed and can see each other's messages as needed. They effectively form a temporary **team** to handle this sub-task, coordinated via A2X messaging. In a few minutes, the IT Support agent fixes a configuration and sends a Message: "Latency issue resolved – network back to normal in Room B."
7. **Human Approval Loop:** Meanwhile, the Smart Assistant has compiled a summary of all meeting rooms and drafted an email to send to the team as requested. The email content includes the status of equipment, network, and climate for each room. Before sending this out via the Email Agent, the assistant knows that company policy requires manager approval for mass emails to the whole team. So it composes an Action to the Email Agent: "Send this summary email to All-Staff list" with the email draft attached, *and it flags this Action as requiring approval from the Manager*. Upon receiving the Action, the Email Agent recognizes the flag and **pauses** – instead of immediately sending, it creates an **Approval Request** (a Feedback message) to the manager's client. The manager gets a notification via the dashboard: "Agent requests approval to send email: [preview of email]. Approve or Reject?" The manager reviews the summary email (which the agent helpfully prepared) and clicks "Approve." This sends a Feedback message back to the Email Agent (and/or the Smart Assistant) indicating approval. The Email Agent now proceeds to send out the email to the team and confirms the Action completion. This whole sequence was handled by A2X's human-in-loop mechanism; the Smart Assistant did not have to implement a custom UI dialog or wait idly – the protocol took care of routing the request and response asynchronously. The manager's **decision was logged** as part of the A2X message trace for future auditing, satisfying compliance requirements.
8. **Safety Feedback and Adjustment:** Let's introduce one more element: suppose the organization has a **Policy Checker agent** that reviews communications for sensitive information. This agent is subscribed to outgoing emails in the environment (with read-only rights) and it sees the draft of the summary email. It notices that the Smart Assistant's draft included an internal ticket number or an overly detailed technical note that might confuse recipients. The Policy agent sends a **Feedback** message

(targeted to the Smart Assistant, referencing the email draft message ID): "Suggestion: remove internal ticket numbers from the summary for clarity." This is essentially an automated peer review. The Smart Assistant receives this feedback before the manager approved, so it quickly modifies the email content to remove the ticket detail. It then updates the draft that went to the manager for approval. The manager might not even need to know this happened – they simply see a slightly cleaned-up email, and the agent has improved the quality of the output thanks to the safety agent's input. This showcases how **third-party agents can provide oversight** and corrections via A2X without halting the entire process. The feedback loop is transparent and structured, which is far better than, say, an unsecured hack that tries to filter text. If the Policy Agent had flagged something truly critical (like a privacy violation), it could even mark it as a **blocking Feedback** that requires resolution before proceeding.

9. **Completion and Session Continuation:** All tasks are now complete – rooms are set up, issues resolved, summary email sent. The Smart Assistant sends a final **Message** to the manager: "All meeting rooms are ready. I've emailed the team a summary of the status." This was not explicitly asked for, but because the session is persistent, the agent can proactively inform the user of the outcome (an example of improved UX through persistence). The manager is pleased and simply closes the dashboard for now. The A2X session with the Smart Assistant remains open in a hibernating state – the agent retains the recent context (stored perhaps in the manager's context artifact or its own memory) so that if the manager returns later, the conversation can pick up where it left off. Later in the day, the manager might reconnect and ask, "Did any issues come up in the meeting rooms this morning?" – and the assistant can recall the context to summarize, rather than treating it as a brand new query. Furthermore, the **knowledge gained** (e.g. that Room B had a network hiccup) could be logged by the assistant into a shared "IT knowledge base" via an Action to a documentation agent, so that this information is available to others or for future troubleshooting.

This scenario demonstrates how A2X seamlessly blends agent-tool interactions, multi-agent collaboration, and human oversight in one unified flow. The Smart Assistant was able to: engage in continuous dialogue with a human; call tool services (email, climate, network) via Actions; spawn a multi-agent coordination (bringing in IT agent) using shared context; handle a human approval loop for safety; and incorporate feedback from a policy agent – all through the same **A2X communication fabric**. The various message types (Message, Action, Observation, Feedback) were used to fulfill different needs, but they worked together within the same session and context, illustrating the power of A2X's simple, consistent design.

Use Cases Unlocked by A2X

By providing a universal and extensible communication layer, A2X enables a range of advanced AI applications that were difficult to implement with earlier protocols. Below are several high-impact use cases facilitated by A2X:

- **Always-On Personal AI Assistants:** With A2X's persistent session and memory features, one can build personal AI assistants that truly accompany a user over long periods and across different platforms. For example, imagine an AI coding assistant that not only helps in one IDE session, but remembers your coding style and project history for months, or a health coach bot that tracks your wellness data over time and interfaces with your smart home devices. A2X allows such an assistant to maintain a **long-term context** for each user (via context artifacts) and to proactively reach out through continuous channels (e.g. sending a reminder or alert without a prompt). It also can seamlessly integrate new tools or IoT devices the user adds: as soon as those advertise themselves via A2X, the assistant can start using them. This is a leap from today's isolated chatbot instances. It unlocks agents that behave more like persistent digital companions, adapting and learning over time while keeping the **user firmly in control** through A2X's oversight capabilities.
- **Collaborative Multi-Agent Teams:** Complex projects (research, software development, business processes) can be handled by a **team of specialized agents** working in concert, often alongside human team members. A2X provides the coordination layer for such multi-agent ecosystems. For example, an "AI product design team" might consist of a market research agent, a design generation agent, a budget analysis agent, and a project manager agent. Using A2X, these agents can share a common context for the project, exchange findings in real time, and divide tasks among themselves – all while a human product lead oversees the discussion via agent-to-human messages. Agents can debate ideas, verify each other's results, and automatically merge their contributions into a final output. Without A2X, one would have to manually integrate multiple protocol channels or build a custom orchestrator to get this level of synergy. With A2X, the *agents themselves* have the tools to organize organically into a workflow, which is a powerful enabler for autonomous project teams and decentralized AI services.
- **Interactive Robotics and IoT Automation:** A2X can serve as the "nervous system" for AI agents embedded in physical environments, such as smart factories, smart homes, or autonomous vehicles. Using the environment integration features, agents controlling different devices can communicate events and commands in real time. For instance, on a factory floor, you could have a safety-monitor agent, a throughput-optimizer agent, and a maintenance agent all connected via A2X. If a sensor (through an Observation) indicates an overheating machine, the safety agent can broadcast an

alert, the maintenance agent can Action the machine to shut down, and the throughput agent can reroute tasks to other machines – coordinating all this through shared context and without human intervention, unless a human supervisor Feedback is required for a critical decision. The persistent channels mean low latency and continuous awareness, crucial for robotics. And importantly, A2X's security model ensures that only authorized agents can send control Actions to physical devices (preventing rogue commands), and human approval can be required for anything dangerous. Essentially, A2X can unify what is today often a patchwork of industrial protocols, IoT hubs, and isolated AI modules into a **cohesive, secure agent network** managing an environment.

- **Knowledge Networks and Continuous Learning:** In knowledge-driven fields like customer support, medicine, or law, it's beneficial to have AI agents that not only answer queries but also **collectively build and refine a knowledge base**. A2X enables a scenario where multiple agents handling different tasks contribute back to a common repository of knowledge via context artifacts or coordinated updates. For example, consider a customer support center with many AI helper agents. As they solve new issues, one agent (or a "learning" agent) can formalize those Q&A pairs into a shared FAQ or documentation by sending an update to a documentation service. If another agent encounters a similar issue later, it can query the updated knowledge base through A2X and find the solution. A2X's pub/sub style Observations also allow an agent encountering a novel problem to **broadcast a request for help** to peers – perhaps a specialist agent picks it up and then everyone learns from the resolution. Over time, this creates a self-improving swarm of agents: each interaction not only serves the immediate user but makes the whole system smarter (subject to validation via Feedback to avoid circulating errors). Human experts can be looped in via A2X when an agent is unsure, ensuring quality. This use case shows A2X supporting a form of **collective intelligence** among AI agents, with the protocol handling the information sharing and permissioning.
- **Composable Cloud AI Services:** As organizations adopt AI, there will be many independent AI services (from different vendors or internal teams) that need to work together. A2X can act as the lingua franca to connect these into **composable workflows** on the cloud. For instance, an e-commerce company might use a third-party AI service for customer support, another for inventory optimization, and its own AI for sales analytics. With A2X, these services – if compliant – could dynamically collaborate on a business process: a customer question about a product delivery could trigger the support agent to consult the inventory agent and logistics agent seamlessly, even though they come from different providers. Each agent advertises capabilities (track shipment, provide status, etc.), and they call each other's Actions through A2X securely. The company's human managers could broadcast a policy update or emergency directive to all agents via one A2X message if needed (e.g. "stop

all automated promotions today") instead of dealing with each system separately. Essentially, A2X can enable an **"Internet of Agents"** – a world where AI services from anywhere can interoperate safely as long as they speak the protocol. This is analogous to how HTTP allowed web servers from any vendor to form the World Wide Web. A2X could do the same for the AI agent ecosystem, accelerating integration and innovation by removing proprietary barriers.

Conclusion

The **Agent-To-Everything (A2X) Protocol** represents a decisive step toward unifying the fragmented landscape of AI agent communication. By learning from the successes and shortcomings of MCP, A2A, and other emerging standards, A2X provides a single framework that covers the full spectrum of agent interactions – tool use, inter-agent dialogue, human collaboration, and environment awareness. It does so with a minimal set of versatile primitives and a modern cloud-friendly architecture, ensuring that the solution is both *elegant* and *practical*. With A2X, developers and researchers no longer need to stitch together multiple protocols or reinvent communication layers for each new project; they can rely on a comprehensive standard that allows agents to **seamlessly talk to anything relevant** in their world.

DataVesper will continue working on A2X as an open specification along with reference implementations and SDKs, inviting the community to adopt and contribute to the protocol's evolution. Our hope is that A2X will catalyze a vibrant, interoperable agent ecosystem – much like HTTP did for web services – where AI agents can easily cooperate across organizational and technological boundaries. By standardizing persistent sessions, shared context, discovery, and safety mechanisms, A2X enables the next generation of AI systems to be **continuous learners, trustworthy collaborators, and deeply integrated** into both our digital and physical environments. In summary, "**Agent-to-Everything**" signifies that an AI agent should not be a siloed point solution, but rather a connected intelligent entity that can engage with whatever it needs – be it another AI, a tool API, a human mentor, or the surrounding world – through one unified language.

References (Endnotes)

1. **Anthropic (2024).** *Introducing the Model Context Protocol (MCP).* Anthropic Announcement, Nov 25, 2024. *(Open standard for connecting AI assistants to data sources and tools.)*
2. **Stytch Engineering (2025).** *"Model Context Protocol: Introduction."* Stytch tech blog, Jan 2025. *(Overview of MCP as a "USB-C for AI" integration protocol.)*

3. **Surapaneni, R. et al. (2025).** *"Announcing the Agent2Agent (A2A) Protocol."* Google Developers Blog, Apr 9, 2025. *(Launch announcement of A2A for open agent interoperability.)*
4. **Broshar, A. (2025).** *"A2A and MCP: Start of the AI Agent Protocol Wars?"* Koyeb Blog, Apr 11, 2025. *(Comparison of A2A vs MCP, discusses complementary roles and potential fragmentation.)*
5. **GetStream.io (2025).** *"LLM Context Protocols: A2A vs MCP."* GetStream blog, 2025. *(Analysis of differences between Google's A2A and Anthropic's MCP, and how they complement each other.)*
6. **GeekyAnts (2025).** *"The Missing Link in Autonomous AI: Agent-to-Human (A2H)."* GeekyAnts Engineering, 2025. *(Discussion of the need for standard human oversight channels in autonomous agent systems.)*
7. **Gupta, A. (2025).** *"Where MCP Falls Short – Drawbacks of Model Context Protocol."* Medium, 2025. *(Review of limitations in Anthropic's MCP, such as context length issues and lack of memory.)*
8. **Gupta, D. (2025).** *"MCP vs A2A: A Comparative Analysis."* Personal blog, 2025. *(In-depth comparison of Anthropic's and Google's protocols, noting gaps and potential integration points.)*
9. **BytePlus (2025).** *"A2A Protocol: Stateless vs Stateful Communication."* BytePlus Tech, 2025. *(Explains how A2A's stateless design contrasts with MCP's stateful sessions, and implications for scalability.)*
10. **Lisowski, E. (2025).** *"What Every AI Engineer Should Know About A2A, MCP & ACP."* Medium, Apr 2025. *(Insights into the convergence of agent communication protocols and the vision of unified standards.)*
11. **Auth0 (2025).** *"Secure Human-in-the-Loop Interactions for AI Agents."* Auth0 Blog, 2025. *(On frameworks for requiring human approval in autonomous agent actions to ensure safety and compliance.)*
12. **IBM Research (2025).** *"What is Agent Communication Protocol (ACP)?"* IBM AI Blog, June 2025. *(Overview of IBM's ACP standard for agent-to-agent communication on edge and hybrid cloud, as part of the BeeAI project.)*
13. **AWS Machine Learning Blog (2024).** *"Design multi-agent orchestration with reasoning using Amazon Bedrock and open-source frameworks."* AWS ML Blog, Dec 19, 2024. *(Demonstration of multi-agent collaboration improving reasoning, and introduction of AWS's Bedrock Agents for multi-agent orchestration.)*