Scaling Agents for Enterprises

Guide to designing and scaling enterprise multi-agent systems using open standard agent protocols

Author: Simon Lacasse (Microsoft)

Table of Contents

Introduction

Enterprises today face fragmented AI ecosystems, isolated agents, disjointed tools, limited scalability, and a flood of incompatible frameworks. A single agent can solve a narrow task, but the future of AI emerges when thousands of agents operate as a connected network of agents: collaborating, sharing memory, coordinating across domains, and evolving into collective hives of intelligence.

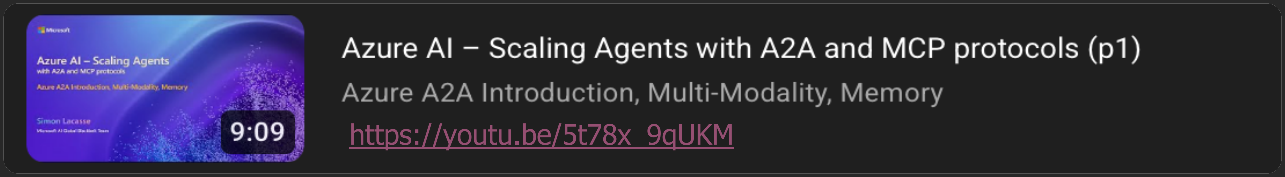
Yet scaling to that level is hard. Without open standards and shared principles, organizations end up with brittle integrations, siloed agents that can't interoperate, and pilot agent projects that never mature into production systems. What's needed is a unified foundation, an open, interoperable fabric that allows agents to connect, coordinate, and evolve together without being slowed down by tech fragmentation in a rapidly accelerating AI landscape.

This paper presents a blueprint for scaling multi-agent systems, a set of foundational capabilities that allow agents to communicate, discover one another, orchestrate work, tap into tools and enterprise systems, preserve memory, and remain observable, secure, and trustworthy at scale.

Demonstration

The following demonstrations provide real-world examples of Azure A2A and MCP protocols orchestrating multi-agent collaboration.

In the first demo, an A2A-powered live insurance claim workflow illustrates how humans and autonomous agents operate across clouds using shared memory, multimodal document understanding, and orchestration.

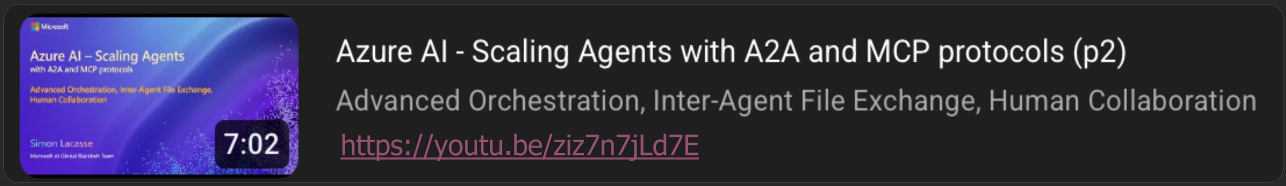

Microsoft
Azure AI – Scaling Agents
with A2A and MCP protocols
Azure A2A Introduction, Multi-Modality, Memory
Simon Lacasse
Microsoft AI Global Blackbelt Team
9:09

Azure AI – Scaling Agents with A2A and MCP protocols (p1)

Azure A2A Introduction, Multi-Modality, Memory

https://youtu.be/5t78x_9qUKM

In the second demo, an A2A-powered content creation workflow highlights advanced orchestration, inter-agent file exchange and human and agent collaboration to automate a complex content creation pipeline.

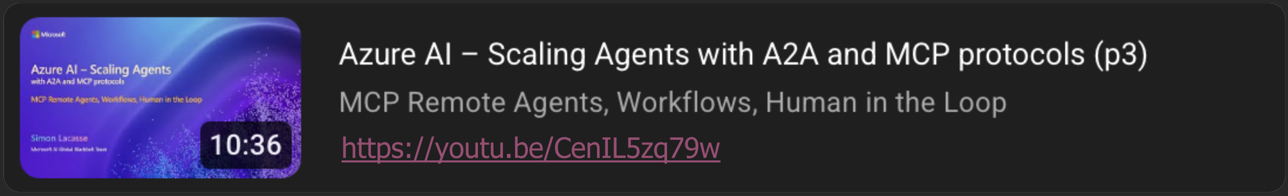

Microsoft
Azure AI – Scaling Agents
with A2A and MCP protocols
Advanced Orchestration, Inter-Agent File Exchange, Human Collaboration
Simon Lacasse
Microsoft AI Global Blackbelt Team
7:02

Azure AI - Scaling Agents with A2A and MCP protocols (p2)

Advanced Orchestration, Inter-Agent File Exchange, Human Collaboration

https://youtu.be/ziz7n7jLd7E

In the third demo, an A2A-powered customer support workflow highlights how MCP-connected agents, backed by enterprise knowledge and human-in-the-loop oversight, can triage, reason, and resolve incidents end-to-end across systems and platforms.


Microsoft
Azure AI – Scaling Agents
with A2A and MCP protocols
MCP Remote Agents, Workflows, Human in the Loop
Simon Lacasse
Microsoft AI Global Blackbelt Team
10:36

Azure AI – Scaling Agents with A2A and MCP protocols (p3)

MCP Remote Agents, Workflows, Human in the Loop

https://youtu.be/CenIL5zq79w

These demonstrations make the architecture tangible, showing how the blueprint translates into operational, production-ready scalable multi-agent systems.

For those interested in exploring the technical foundation behind these demonstrations, an experimental GitHub repository is available as a reference implementation. While fully functional, it should be treated as a starting point rather than a finished product, it may include limitations and areas for improvement and is intentionally kept open for extension.

The real strength of this project is in its architectural blueprint, a reusable, open-standard foundation for anyone looking to build their own A2A-compliant multi-agent systems. We welcome the community to explore, extend, and innovate on this framework, driving the future of intelligent agent ecosystems together.

https://github.com/slacassegbb/azure-a2a-main/

Blueprint for Scaling Multi-Agent Systems

Building effective agent systems requires more than standalone components, it demands a blueprint for how agents operate together as an interconnected, intelligent network. This framework defines the core principles and capabilities needed for agents to communicate, coordinate, and evolve within a cohesive ecosystem.

Each component of the blueprint tackles a barrier that often leaves agent pilots stuck in silos: enabling a common language for communication, mutual discovery and trust, seamless integration with enterprise systems, cross-workflow collaboration, and end-to-end observability, governance, and security. Without these foundations, agent networks remain fragile experiments, with them, enterprises can confidently scale multi-agent systems into robust, interoperable ecosystems ready for real-world impact.

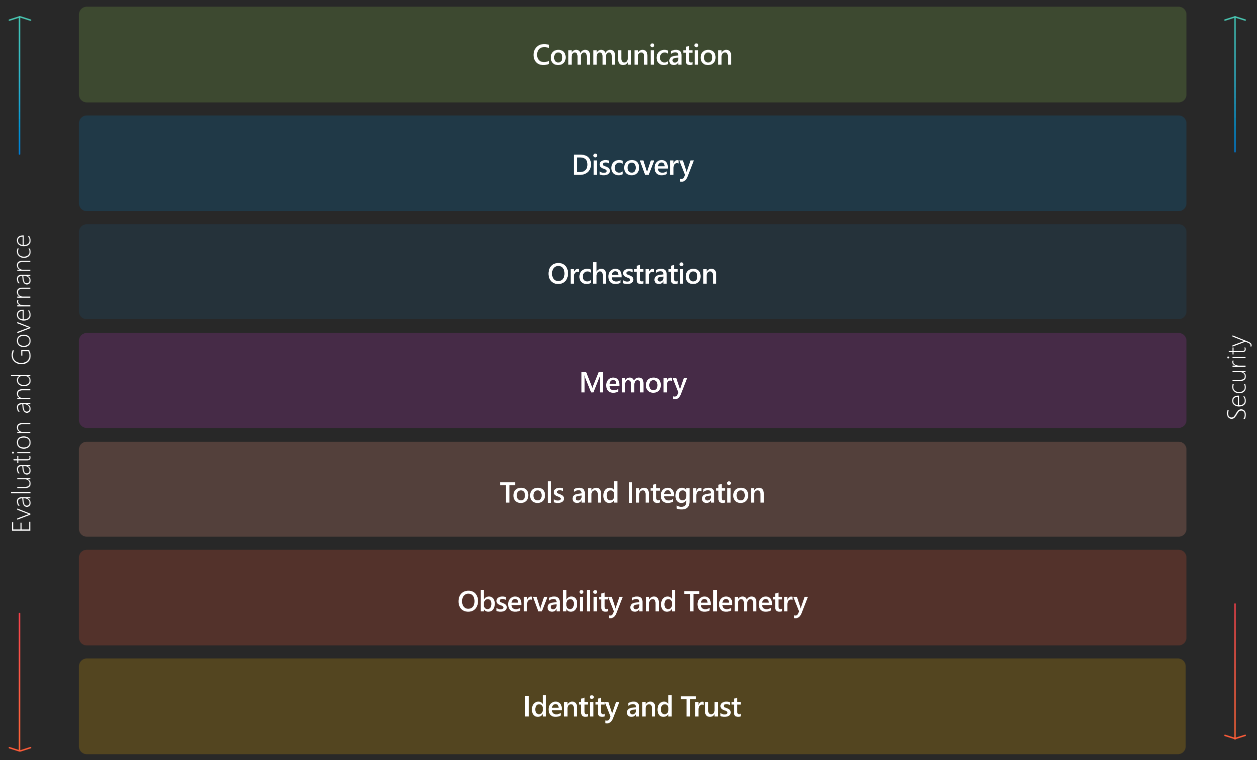


Figure 1. Scaling Agents Blueprint

The blueprint is organized into eight core layers, each solving a critical challenge in building real-world agent ecosystems:

- **Communication** — establishes shared protocols and standards for agents to exchange information.
- **Discovery** — ensures agents are findable, reusable, and not confined to silos.
- **Orchestration** — coordinates multi-step workflows and decision-making across distributed agents.
- **Integration & Tools** — connects agents with enterprise data, APIs, and operational systems.
- **Memory** — enables agents to retain context, learn, and adapt over time, turning stateless actors into contextual collaborators.
- **Telemetry & Observability** — makes agent behavior transparent, measurable, and debuggable.
- **Identity & Trust Management** — secures participation with verifiable identities and controlled access.
- **Evaluation & Governance** — embeds continuous oversight, compliance, and ethical safeguards into the agent network.

Together, these elements form the architectural blueprint enterprises need to move beyond isolated experiments, creating agent systems that are interoperable, trustworthy, and capable of true collaboration at scale.

The remainder of this paper walks through the blueprint one layer at a time, explaining what each layer is, why it matters, and how it is realized in the design of an A2A-compliant Azure multi-agent system.

Communication Protocols

Every multi-agent system begins with communication. If agents can’t talk to each other, they can’t collaborate, and worse, they can’t interoperate across clouds, vendors, or frameworks. Communication isn't just about exchanging messages; it’s the foundation for distributed intelligence multi-agent systems at scale.

Open Standard Agent Protocols

In traditional software, APIs, message buses, and RPC mechanisms provide the structure for inter-service communication. But with agents, systems capable of reasoning, tool use, and adaptive planning, we need a richer form of interaction. Agents don’t just exchange data; they negotiate, delegate, and collaborate dynamically.

To enable that, a new class of open agent communication protocols has begun to emerge, moving beyond proprietary APIs and ad hoc integrations toward a common, interoperable language for agents.

Modern agent protocols like **A2A** (Agent-to-Agent), **MCP** (Model Context Protocol), **ACP** (Agent Communication Protocol) are designed to support the full spectrum of agent collaboration, including:

- **Discovery** – Agents can publish and retrieve standardized “Agent Cards” that describe their identity, skills, and endpoints.
- **Negotiation & Task Delegation** – Agents can assign, accept, or reject tasks, update status, and share structured results through common message formats.
- **Long-lived & Streaming Interactions** – Using HTTP and Server-Sent Events (SSE), agents can coordinate in real time, share intermediate progress, and handle asynchronous workflows.
- **Privacy-Preserving Internals** – Each agent operates as a black box, exposing only standardized interfaces without revealing proprietary logic or reasoning chains.
- **Cross-Cloud Interoperability** – Because these protocols build on open web standards (HTTP, JSON-RPC, JSON), agents can operate across different clouds, runtimes, and vendor ecosystems without custom bridges or middleware.

Together, these efforts signal the rapid convergence toward open standards for agent interoperability. Each protocol fills a distinct layer of the stack, MCP handling internal reasoning and tool use, A2A managing agent-to-agent collaboration, and ACP/A2P addressing local orchestration.

As the agent ecosystem matures, we can expect more open standards to emerge, extending into areas like decentralized agent identity, semantic interoperability, and secure data exchange. Much like HTTP unified the early web, these protocols are laying the foundation for a networked ecosystem of intelligent agents, one where models, tools, and systems from different vendors can communicate and collaborate.

The Agent-to-Agent (A2A) Protocol

In this design we are going to focus on Agent-to-Agent (A2A) Protocol, developed by Google with Microsoft as a primary contributor. It defines the core primitives for multi-agent collaboration—**Agent Cards, Tasks, Artifacts, and Messages**—which allow agents to discover each other, negotiate, share work, and exchange structured data.

What makes A2A so attractive is that it isn’t a brand-new transport mechanism. Instead, it builds on standards that already run the web:

- HTTP and SSE as the transport layer,
- JSON-RPC as the lightweight request/response mechanism, and
- JSON as the universal data format.

A2A ensures that agents can interoperate across languages, runtimes, and environments without requiring proprietary infrastructure. In other words: if your system already speaks web standards, it can speak A2A.

Understanding the A2A Protocol

This diagram illustrates the core data model of the A2A (Agent-to-Agent) protocol, the foundation that allows agents to discover, negotiate, and collaborate.

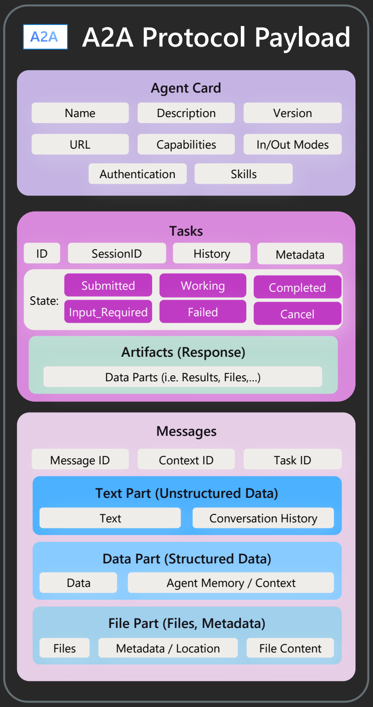


Figure 2. A2A Payload

A2A Agent Card

- Think of the Agent Card as a public profile / billboard for an agent.
- It contains metadata such as name, description, and a list of capabilities (the tasks or modalities this agent supports).
- A client agent can fetch another agent’s card to decide whether interaction is possible and how to negotiate with it.
- In the official spec, Agent Cards are JSON documents typically exposed at a well-known URL (e.g. /.well-known/agent.json).

A2A Tasks

- A Task is the stateful unit of work (and intent) in A2A, created by a client agent, owned/tracked by the receiving agent, and persisted across messages for the duration of its lifecycle.
- Each Task has a unique Task ID, may optionally belong to a Context ID (session/grouping), and advances through a defined state machine (e.g., submitted → working → input-required → completed/failed), supporting long-running and asynchronous progress.
- The Task abstraction lets agents execute multi-step, long-lived, interactive workflows, not just one-off messages, including requesting clarifications mid-flight and emitting artifacts/results upon completion.

A2A Artifact

- Artifacts are the outputs of tasks. e.g. files, structured results, reports, or any packaged data the agent produces.
- Artifacts are built from "Parts" (text, structured data, file parts) and can be streamed or updated incrementally.
- This enables partial results or long-running computations to be shared before the task fully completes.

A2A Messages

- Messages play a core role in A2A, enabling discovery and negotiation that can later be promoted to a stateful Task when the work becomes goal-oriented or long-running.
- Within each Task, agents exchange Messages which carry the evolving conversation or negotiation about that Task.
- Messages have references to Message ID, Context ID, and Task ID, linking them to the broader workflow.
- A message is composed of multiple "Parts," which can be:
    - Text Part — unstructured text or conversation history
    - Data Part — structured JSON, parameters, context updates
    - File Part — attachments, metadata, or binary content

The design lets agents send rich, mixed-modality content (text, JSON, files, streaming content) as part of their ongoing collaboration. Together, these components define a universal format for agent communication, enabling mixed-modality data exchange, task orchestration, and interoperability across platforms, vendors, and clouds.

You can learn more about the Google A2A protocol here: https://github.com/a2aproject/A2A

Adapting the Azure Host Agent with A2A Protocol

To anchor this ecosystem, we'll create a Host Agent by extending an Azure AI Foundry Agent, transforming it into an A2A-compliant host orchestrator. This involves adapting the Azure AI Foundry Agent's existing interfaces and message-handling logic to communicate using the A2A schema and transport layer.

It's worth noting that we are not extending or altering the A2A protocol, we adopt it exactly as defined to remain fully compliant and interoperable with any A2A-capable agent that wants to connect to this system. By keeping the standard untouched, we ensure long-term compatibility and open integration. Instead of modifying the protocol, we build on top of it, progressively layering capabilities such as shared memory, intelligent orchestration, multimodality, and dynamic discoverability. This allows the Host Agent to evolve into a powerful intelligence hub that coordinates distributed workflows while staying true to open A2A principles.

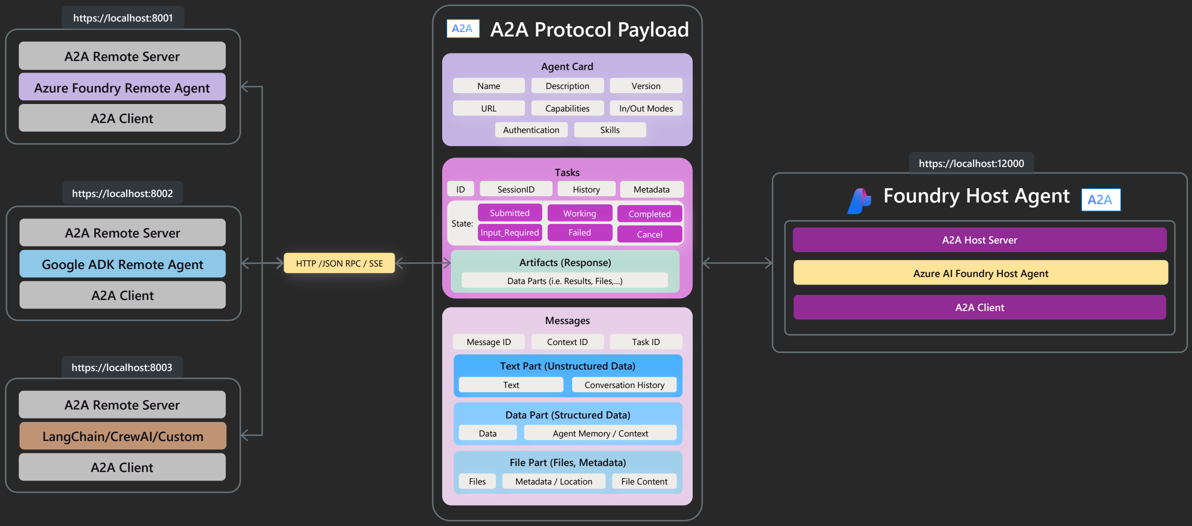


Figure 3. Integrating A2A protocol with Azure AI Foundry Agent

In this design we expose the Host Agent's capabilities through an A2A-compliant Agent Card, implement standardized task and message endpoints, and enable streaming communication via HTTP and Server-Sent Events (SSE). This allows the Host Agent to send and receive structured A2A messages, manage long-running tasks, and collaborate with any other A2A-enabled agent in real time. By building on A2A, the Host Agent doesn't just communicate, it interoperates. It can connect with any remote agent that speaks A2A, regardless of platform, vendor, or deployment environment, creating a unified, extensible network of agents that scales across clouds and teams, forming a truly collaborative and interoperable agent ecosystem.

Instead of using only the SDK, we use the Foundry Agent HTTP API directly working with threads, runs, and tool outputs. This gave us low-level control over the run lifecycle, allowing us to intercept events and delegate them to remote agents over A2A.

The effect is that we're not just running Foundry "as is." We're extending its runtime with A2A, so the assistant becomes capable of orchestrating an entire mesh of agents. Here's how it works:

- **Tool delegation:** When the model requests an action, the host interprets it as an A2A call (send_message) and fans out requests to remote agents.
- **Parallel execution:** Those remote agents (Microsoft, Google, LangChain, CrewAI, custom) are invoked concurrently, with results normalized through a thin connector. Note that agents can also be executed sequentially if needed.
- **Streaming + events:** Responses flow back over HTTP/SSE + JSON-RPC, with granular status events surfaced for UI and telemetry.
- **Result submission:** The host then submits outputs back into Foundry, preserving its native reasoning and summarization. If Foundry fails, the host can still aggregate and respond directly.

This design keeps Foundry Agent in the driver's seat for reasoning, conversation flow, and final responses, while A2A provides the interoperable backbone for discovery, task routing, artifact exchange, and streaming updates.

Azure AI Foundry Agent becomes more than a model runtime, it becomes the system of engagement for heterogeneous multi-agent ecosystems, able to coordinate agents across vendors, clouds, and frameworks through the same web standards your systems already speak.

Learn more about the Azure AI Foundry Agent service: https://learn.microsoft.com/en-us/azure/ai-foundry/agents/overview

Discovery

Once agents can communicate, the next challenge is knowing whom to talk to and why. That's where discovery comes in. In our architecture, discovery is powered by a runtime registry in our host orchestrator built using the A2A Agent Cards.

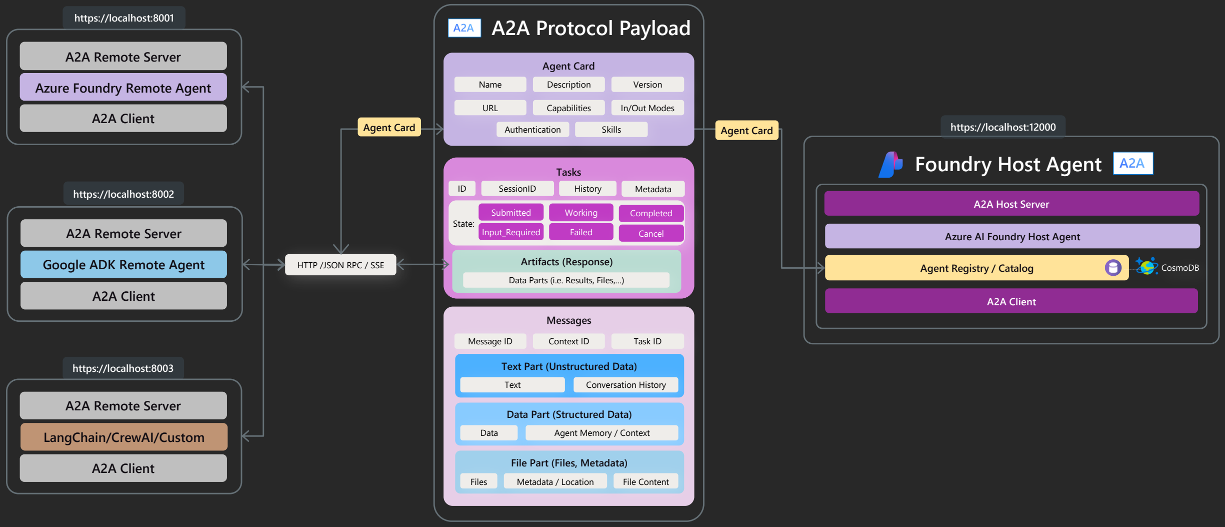


Figure 4. Discoverability

In A2A, every agent exposes an Agent Card; a self-describing JSON document that advertises its identity, skills, capabilities, and endpoint. Our Foundry host orchestrator consumes these cards as its source of truth: they tell the host which agents are available, what tools they provide, and how to connect.

- **Name** - Human-readable name of the agent
- **Description-** More detailed explanation of what the agent does
- **Version -** Version of the agent or its interface
- **url -** The base endpoint (root URL) where the agent's A2A server is reachable
- **capabilities -** Which protocol features the agent supports (e.g. streaming, pushNotifications)
- **defaultInputModes / defaultOutputModes -** The MIME types or modalities the agent expects as input and produces as output (e.g. text, application/json)
- **authentication -** What authentication schemes or methods the agent accepts (e.g. bearer token)
- **skills -** A list of the "skills" the agent offers — each with its attributes like id, name, description, tags, examples, and optionally input/output modes for that skill

Integrating Agent Cards into the Foundry Host

We extended our Foundry host orchestrator to treat Agent Cards as a runtime agent registry:

- **In-memory registry at runtime** – when the host starts and remote agents are registered, it loads the active Agent Cards into memory. This lets it quickly decide which agent to call when a tool request comes in, match capabilities, and route messages.
- **Delegation logic** – when a tool call is triggered, the Foundry host checks the registry, selects the best agent, and connects to it over A2A. The registry ensures this decision-making is dynamic, not hardcoded.

Agent Catalog

The runtime registry is lightweight and in-memory. It only exists while the Host Agent is active and is optimized for real-time decision-making, fast lookups, capability matching, routing, and coordinating active agents in flight.

To enable reuse beyond a single session, we maintain a persistent Agent Catalog. In our current implementation, this is a JSON-based directory that stores each agent's identity, capabilities, metadata, and connection details. Instead of automatically loading everything at startup, agents in the catalog remain persisted but inactive until they are explicitly registered into the session, allowing the Host Agent to dynamically activate only what's needed. New agents registered at runtime can also be written back to the catalog, making them available for future sessions without redefinition.

This establishes a universal agent directory, not just for execution, but for long-term discoverability, reuse, and composition. Looking ahead, this catalog can evolve into a full enterprise-wide agent discovery layer, a searchable hub where teams can browse, register, and assemble agents into workflows, enabling governance, interoperability, and large-scale reuse across the organization.

Evolving the Catalog with Cosmos DB (Private, Shared, Enterprise, Public)

As enterprises scale, the catalog doesn’t need to remain local. By storing it in Azure Cosmos DB, the catalog becomes a globally available registry that can be shared across multiple Host Agents, environments, and regions. Cosmos DB’s low-latency, distributed architecture makes it ideal for persisting Agent Cards at scale, enabling multi-orchestrator discovery, synchronization, and reuse across the entire agent ecosystem.

Once the catalog is backed by Cosmos DB, it can evolve beyond a single directory into a federated catalog model. Catalogs can operate at multiple visibility levels: private catalogs that let individuals assemble their own agent teams, including bringing in their own custom agents, to boost personal productivity, shared catalogs for teams or departments to work from a common set of agents for recurring tasks, enterprise catalogs that provide a unified and approved set of agents for organization-wide use, and public or partner catalogs that enable controlled sharing of agents beyond the organization when external collaboration is required. With RBAC and policy governance, each agent entry can define who can discover, register, or invoke it, turning the catalog into a secure, role-aware directory, more like an API agent marketplace or enterprise service hub for agents.

You can learn more about Azure CosmosDB here: https://azure.microsoft.com/en-us/products/cosmos-db

Orchestration

With communication and discovery in place, the next step is orchestration, coordinating how agents work together across tasks and workflows. In our architecture, orchestration is built directly on the A2A protocol, using its task model as the backbone for multi-agent coordination.

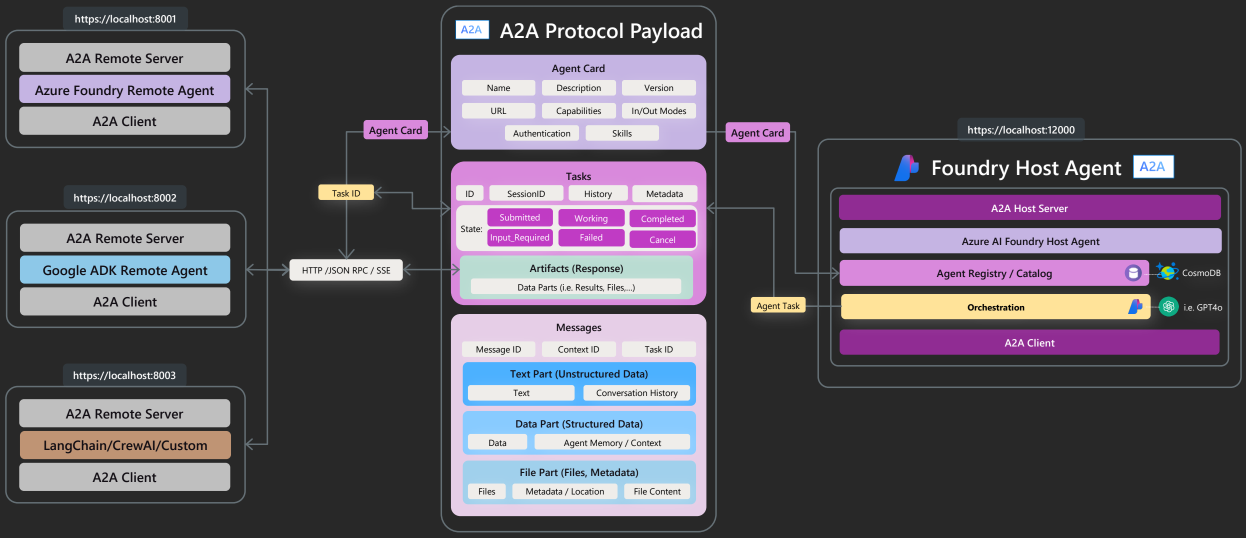


Figure 5. Orchestration

Each task carries a Task ID (a unique identifier for tracking progress), a Context ID (linking related tasks together as part of a workflow), and a state. The task lifecycle is simple but powerful:

- **pending** – the task has been created but not yet started.
- **running** – the task is actively being worked on by a remote agent.
- **completed** – the task finished successfully, and results have been returned.
- **failed** – the task ended with an error.
- **canceled** – the task was stopped before completion.
- **input-required** → remote agent needs more info before proceeding

Our host agent uses these primitives to orchestrate work. When the LLM (running in Foundry) determines that an action is required, it issues a send_message call to the host. The host looks at the Agent Catalog, selects the right remote agent, and forwards the request over A2A. From there, the host tracks task state, collects results, and feeds them back into the run.

Because orchestration is prompt-driven, you can even design custom workflows in plain language. By instructing the host agent which remote agents to call (and in what order), you can chain multiple agents into a lightweight, ad hoc workflow, no extra code needed.

Parallel vs Sequential Orchestration

Effective multi-agent systems require flexibility in how tasks are coordinated, whether multiple agents should execute concurrently or whether their actions must follow a strict, deterministic order. This distinction defines two complementary orchestration strategies: parallel and sequential.

Parallel Orchestration

Parallel orchestration maximizes throughput by allowing multiple agents to operate independently and simultaneously. The host or LLM decomposes a goal into several subtasks and dispatches them across agents without waiting for intermediate outputs. This approach is ideal for workflows that involve **independent or loosely coupled tasks**, for example, when several analytics agents can process different datasets or modalities in parallel.

In our system, this corresponds to **direct execution**, where the host agent's LLM issues send_message calls in parallel. No explicit plan or reasoning loop exists; instead, execution relies on **function-based concurrency**. It is efficient and scalable, but inherently non-deterministic, task ordering, dependencies, and goal evaluation are not explicitly tracked.

Sequential Orchestration

Sequential orchestration, in contrast, introduces **determinism and reasoning over state**. Each step depends on prior context, the output of one agent informs the next. This model is necessary when the workflow represents a chain of dependent reasoning steps (e.g., fraud detection → risk evaluation → refund issuance).

Sequential orchestration requires an explicit **planning and control layer**. In our implementation, this is achieved where the orchestrator LLM builds and iteratively updates a structured plan (AgentModePlan) composed of tasks (AgentModeTask) and their results. Each loop iteration reviews prior outputs, determines whether the goal is complete, and decides the next task to run. This transforms orchestration from **stateless dispatching** into **state-aware reasoning**.

Deterministic Planning with Structured Outputs

To make this process reliable and reproducible, planning is encoded through Pydantic models that define the allowable schema for every plan and decision.

The orchestration layer is anchored around three key Pydantic models:

- **AgentModeTask** defines a single A2A task with fields for task_description, recommended_agent, state, and the resulting output.
- **AgentModePlan** maintains the global execution plan tied to a user goal, holding the full list of tasks and their statuses.
- **NextStep** represents the orchestrator's structured decision, whether the goal is complete and, if not, what the next task should be.

These models are validated on every iteration, ensuring the orchestrator's reasoning remains type-safe, schema-constrained, and fully traceable across runs.

Each AgentModeTask represents a concrete A2A operation, while the AgentModePlan anchors these tasks to a single goal and tracks progress via its goal_status. On every iteration, the orchestrator emits a NextStep object using OpenAI's structured output interface, guaranteeing that the model's response conforms exactly to this schema (no free-form text or ambiguous output).

This structured approach ensures **deterministic dynamic planning that is goal oriented**, every orchestration decision can be parsed, validated, and traced. It eliminates ambiguity while allowing the LLM to adapt dynamically to new information from remote agents.

From Orchestration to "Agent Mode"

This structured, loop-driven sequential orchestration is what we call **Agent Mode**, inspired by "agent mode" in **coding copilots and autonomous coding agents**. Just as GitHub Copilot's agent mode iteratively plans, executes, and evaluates code changes, our Agent Mode orchestrator dynamically plans, executes, and evaluates agent calls until the user's goal is reached.

A2A Protocol Implementation

Under the hood, both orchestration strategies leverage the **A2A protocol** for task transport and state tracking. Each send_message or orchestration call carries:

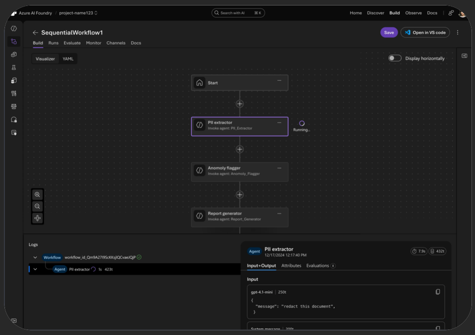
- **Task ID** – unique identifier per task,
- **Context ID** – linking related tasks in one workflow,
- **State** – pending, running, completed, failed, or input-required.

In **Agent Mode**, the host and orchestrator LLM jointly maintain a structured plan where each A2A task transition (e.g., pending → running → completed) feeds back into the reasoning loop. This enables the orchestrator to make context-aware, sequential decisions and ensures that the overall workflow follows a deterministic, explainable path from goal to completion.

Advanced Workflows - Microsoft Agent Framework

For more advanced orchestration scenarios, multi-step planning, long-running workflows, or graph-based execution, we can extend this with the Microsoft Agent Framework. The Microsoft Agent Framework provides an orchestration runtime with planners, memory, and skill composition, making it easier to build reusable workflows that combine LLM reasoning with deterministic logic.

A2A tasks handle the core orchestration loop, while Microsoft Agent Framework can be layered in when enterprises need richer workflow control, planning, and enterprise integration.


Azure AI Foundry / project-name123
Search with AI ⌘K
Home Discover Build Observe Docs
SequentialWorkflow1
Save
Open in VS code
Build Runs Evaluate Monitor Channels Docs
Visualizer YAML
Display horizontally
Start
PII extractor
Invoke agent: PII_Extractor
Running...
Anomoly flagger
Invoke agent: Anomoly_Flagger
Report generator
Invoke agent: Report_Generator
Logs
Workflow workflow_id_Qm9A27I9ScKKsjlQCvaeJQjP
Agent PII extractor 1s 423t
Agent PII extractor
12/17/2024 12:17:40 PM
7.9s 432t
Input+Output Attributes Evaluations 8
Input
gpt-4.1-mini 250t
System message 200t

```
{
  "message": "redact this document",
}
```

Figure 6. Microsoft Agent Framework

Find out how Microsoft Agent Framework can help simplify the orchestration of multi-agent systems and keep developers in flow: https://azure.microsoft.com/en-us/blog/introducing-microsoft-agent-framework/

Memory

Communication, discovery, and orchestration can connect agents into a network, but without memory, that network is stateless, unable to learn from the past or retain context. Memory is what transforms a collection of agents into a truly adaptive ecosystem: one that remembers, evolves, and stays relevant over time.

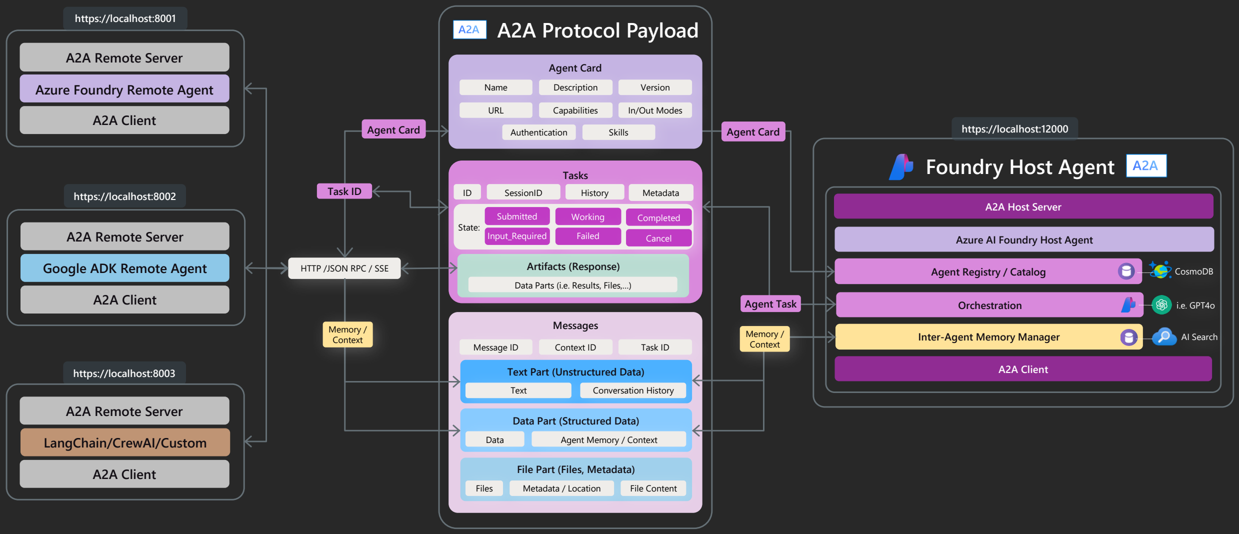


Figure 7. Memory

Sharing Memory via A2A

We leverage the A2A protocol as the backbone for memory exchange. Memory is passed directly in the message payloads, so every remote agent receives the relevant context for its task. This means:

- Remote agents always operate with shared context provided by the host.
- The host can also collect and persist memory from remote agents, creating a full picture of everything happening in the system.

As illustrated in Figure 7, memory flows through the same A2A messages as tasks and artifacts, we preserve end-to-end consistency: what we send is what we store, and what we retrieve.

As memory grows, especially when storing or retrieving extended context, it can quickly exceed the model’s context window. Sending long memory sequences directly in A2A payloads is not only inefficient, it can degrade performance and even confuse agents with unnecessary or outdated information.

To mitigate this, we’ll store memory in a vector database, allowing us to retrieve only the most relevant chunks related to the current task or agent interaction. This ensures that every A2A exchange remains lightweight, focused, and performant, while still giving agents access to rich historical context when needed.

By separating memory from the real-time communication layer, we preserve efficiency without sacrificing intelligence, enabling agents to reason contextually at scale.

Storing Memory in Azure A! Search

All interactions, user ↔ host, host ↔ remote, are persisted as full A2A payloads (both outbound requests and inbound responses). These are vectorized and stored in Azure AI Search, which acts as our memory store.

- **Vectorization:** Outbound and inbound payloads are embedded with Azure OpenAI embeddings.
- **Indexing:** Both the raw A2A JSON and embeddings are stored in a Azure AI Search index. Metadata like agent name, timestamp, and processing time are added for filtering and analytics.
- **Retrieval:** On each new task, the host embeds the query, runs a vector search over prior interactions, and injects the most relevant results into the next send_message call.

This ensures that every agent call is enriched with contextual history, not just summaries, but full protocol-shaped payloads that can be reconstructed and reused.

Why Memory Matters

- **Relevance:** Agents work with prior answers, artifacts, and user history instead of starting from scratch.
- **Adaptability:** Each agent can adapt based on what others have already done in the

workflow.

- **Auditability:** Because full A2A payloads are stored, you can always ask "what did we ask?" and "what did they return?" and reconstruct the flow.
- **Performance at scale:** Azure AI Search handles fast vector similarity at enterprise scale, making retrieval practical even in complex workflows.

In short, Azure AI Search + A2A = protocol-aware memory. By storing and retrieving structured A2A interactions, we give the entire system continuity and adaptability, turning a set of agents into a coherent, adaptive network.

Long-Term Memory and Adaptive Behavior Across Sessions

Because memory is persisted in Azure AI Search rather than tied to a single runtime session, it enables true long-term memory adaptation. This would allow our Host Agent to retrieve past interactions, even from previous days or entirely separate workflows, and use them to shape how agents behave in new sessions. This turns memory from a short-lived context buffer into a progressive, evolving knowledge base that agents can learn from over time.

This means agents can remember user preferences, historical resolutions, anomalies, and strategic decisions, and adapt their actions without being explicitly reconfigured each time. Over repeated use, the system begins to behave less like a stateless automation engine and more like a continuously learning collective, capable of refinement, personalization, and strategic evolution across workflows, users, and business scenarios.

You can learn more about Azure AI Search here: https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search

Inter-Agent File Exchange

In large multi-agent systems, collaboration extends beyond text exchanges. Agents frequently exchange files, documents, images, datasets, and other rich media, as part of reasoning or task delegation. To make this seamless and interoperable, the A2A protocol defines a universal content-exchange mechanism based on the concept of Parts: modular message components that can represent text, structured data, or binary files.

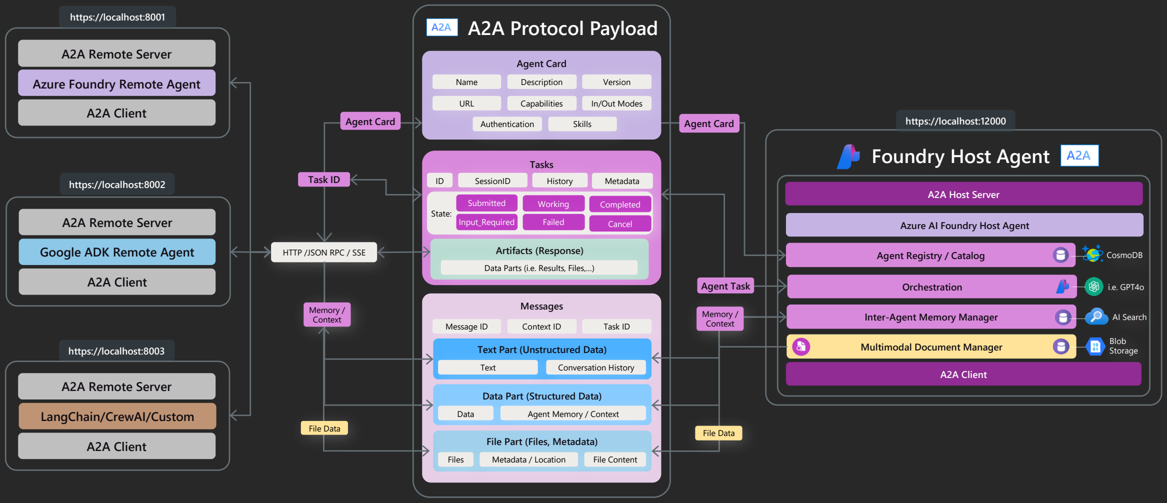


Figure 8. Multimodal Processing

Within each A2A message, file exchange occurs through specialized Part types:

- **TextPart** for unstructured messages,
- **DataPart** for structured metadata or JSON payloads, and
- **FilePart** for binary files or file references.

Files can be transmitted inline (using base64-encoded bytes for small assets) or by reference (using secure URIs for larger artifacts). The Host Orchestrator automatically decides how each file should be represented based on size and configuration.

Smaller assets are embedded directly in the message, while larger ones are uploaded to Azure Blob Storage, generating a time-bound SAS URI that remote agents can retrieve on demand. Each file exchange is accompanied by a DataPart carrying metadata such as filename, MIME type, role (e.g., base, overlay, mask), and storage type.

This approach ensures maximum interoperability: agents that support HTTP can stream files directly from Blob storage, while isolated or air-gapped agents can still operate on embedded bytes. Regardless of transport method, the A2A protocol guarantees every file is addressable, traceable, and reusable across tasks through unique artifact identifiers and consistent metadata schemas.

Bonus: Multimodal Content Processing

File exchange alone is only part of the story, agents also need to understand what's inside these artifacts. The Host Orchestrator extends the A2A protocol with a multimodal ingestion and preprocessing layer, responsible for converting raw files into structured, searchable representations.

Azure AI Content Understanding plays a key role in this process. It is a Microsoft cognitive service designed to automatically extract and structure information from diverse content types, including PDFs, Office documents, HTML pages, emails, images, and videos. By applying layout-aware OCR, entity extraction, semantic parsing, and text segmentation, it transforms unstructured content into rich, machine-readable data that downstream agents can consume directly.

When a file is uploaded or received through inter-agent exchange, the Host automatically:

1. Detects the file type and content modality.
2. Runs it through Azure AI Content Understanding, extracting text, tables, images, entities, and semantic structure.
3. Stores the raw asset in Azure Blob Storage and the extracted metadata, markdown, or embeddings in Azure AI Search for retrieval and context enrichment.
4. Every file thus yields two synchronized artifacts:
5. A FilePart referencing the original file (via Blob URI or inline bytes).
6. A DataPart containing structured semantic content and metadata.

Remote agents can then:

1. Reason directly on the DataPart, leveraging extracted content without reprocessing.
2. Fetch the raw file via the FilePart URI for deeper domain-specific analysis (e.g., OCR, image segmentation, financial parsing).

By representing all files through A2A’s dual FilePart and DataPart contract, multimodal intelligence becomes protocol-native. Agents can not only exchange files, they can query, analyze, and reason over documents, images, videos, and datasets, all through the same mechanisms used for text and task coordination. This unified approach ensures file-based interactions feel as seamless as sending a message, while reducing payload overhead, preserving memory continuity, and enabling interoperable multimodal workflows across the entire agent ecosystem.

You can learn more about Azure Content Understanding here: https://azure.microsoft.com/en-us/products/ai-services/ai-content-understanding

You can learn more about Azure Blob Storage here: https://azure.microsoft.com/en-us/products/storage/blobs

Tools and Integration

Once agents can communicate via A2A, be discovered, orchestrate together, share context through memory, and exchange structured artifacts through multimodal processing, the next critical capability is action, the ability to call tools, query knowledge sources, invoke APIs, trigger workflows, and interact directly with enterprise systems. This is where tool integration becomes a critical layer of the architecture.

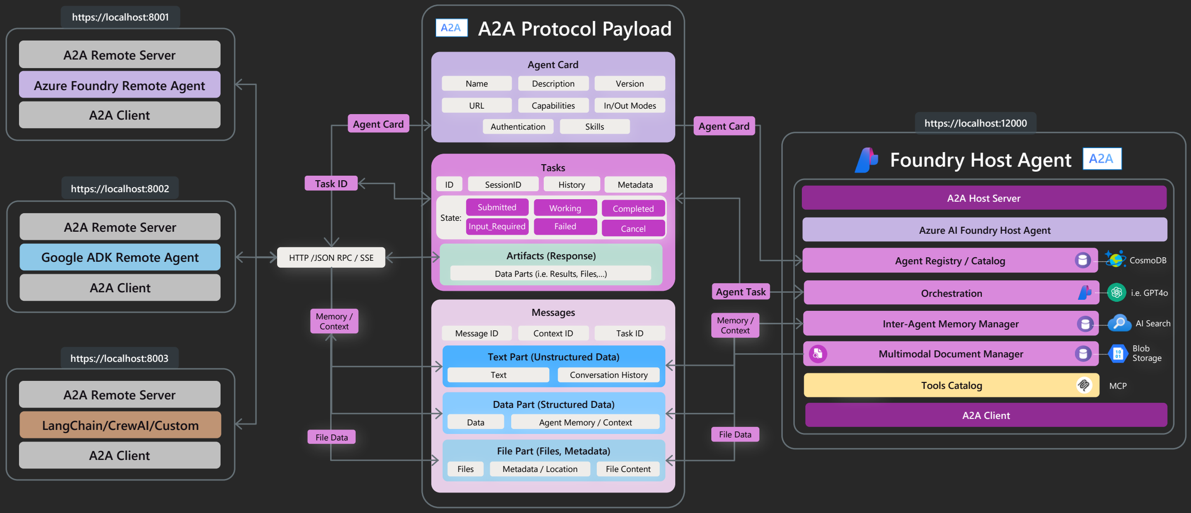


Figure 9. Tools & MCP Protocol

Rather than hard-coding integrations inside each agent, the Host Orchestrator becomes the tool routing layer, exposing capabilities to agents in a standardized, discoverable way through A2A and MCP (Model Context Protocol).

Tool Invocation via Azure AI Foundry

Azure AI Foundry's Agent Service is built to make tool integration easy and modular. You don't have to hardcode agent logic, tools can be registered declaratively (e.g. via OpenAPI or function signatures), making them discoverable and usable by any agent.

Some of the built-in tools that Azure AI Foundry supports out-of-the-box include:

- Grounding with Bing Search — for pulling in fresh web-based information and context.

- Azure AI Search / File Search — for querying private or document-based knowledge sources.
- Azure Logic Apps — to orchestrate broader workflow logic in a low-code/no-code fashion.
- Azure Functions (custom action tools) — to run bespoke computations or integrations as needed.
- Browser Automation — letting agents manipulate or interact with web UIs as needed.
- Deep Research (preview) — for advanced, multi-step research pipelines that synthesize data from across the web.
- Code Interpreter / Function Calling — agents can call structured functions or run limited code in sandboxed environments.
- Microsoft Fabric tool — to interact with data in Fabric through conversational agents (recently introduced)

You can attach tools at different levels (agent-level, thread-level, or run-level) so that more specific overrides can be applied when needed and you can easily create your own tools as needed.

In practice, when an agent detects an actionable intent (e.g., "look up this record," "trigger a reporting workflow," or "scrape this web page"), it doesn't "guess" via natural language. Instead, it chooses the appropriate registered tool and invokes it through a structured payload, ensuring consistency, observability, and extensibility.

MCP: A Unified, Open Tool Invocation Layer

As agent ecosystems grow, bespoke API integrations don't scale. Each new service or capability would traditionally require a custom wrapper, hardcoded logic, and per-agent integration work. To break away from that pattern, we can adopt the Model Context Protocol (MCP), an emerging open standard that defines a consistent way for agents to discover, describe, and invoke external tools using a structured JSON-RPC interface.

MCP is rapidly gaining traction across the industry as a vendor-neutral protocol for tool invocation, enabling agents from different platforms, runtimes, or vendors to interoperate around a shared tool definition model. Instead of agents being tightly coupled to platform-specific plugins or SDKs, MCP servers expose tools with self-describing metadata, allowing agents to dynamically understand what tools exist, what they do, what inputs they require, and how to call them, all without bespoke integration code.

Native MCP Support in Azure AI Foundry Agent Service

Azure AI Foundry Agents include first-class support for the Model Context Protocol (MCP), meaning MCP servers and tools can be registered directly in the Azure AI Foundry environment without custom integration code. Once registered, Azure AI Foundry automatically exposes these MCP tools through a standardized discovery layer and wraps tool calls in a protocol-aligned invocation format.

This means any enterprise system, API, or workflow that exposes an MCP endpoint instantly becomes available across the entire multi-agent network, without modifying agent code. MCP effectively becomes a universal tool substrate, with Azure AI Foundry handling tool lifecycle, schema enforcement, invocation transport, and response packaging natively. In our Github repository we show an example on how to use MCP with our customer support remote agent.

You can read more about Azure AI Foundry Agent service native MCP tool invocation here:
https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools/model-context-protocol

Optional: To extend governance and visibility across the enterprise, Azure API Management (APIM) and API Center can act as the catalog layer for MCP servers, enabling discovery, RBAC-scoped visibility, and policy-based access control over which agents or teams can invoke specific MCP endpoints. This allows MCP tools to be published, secured, and monitored just like any other managed API asset, while remaining natively callable within Foundry’s agentic runtime.

You can learn more about MCP Servers and API Management here:
Inventory and Discover MCP Servers in Your API Center - Azure API Center | Microsoft Learn
Overview of MCP servers in Azure API Management | Microsoft Learn

Observability and Telemetry

In a multi-agent system, agents operate autonomously, make delegated decisions, and trigger downstream actions across external systems. Without observability, this becomes a black box, tasks disappear into remote agents, tool calls fire across services, and memory is updated silently. When something goes wrong or when something goes right, there is no traceable record of why.

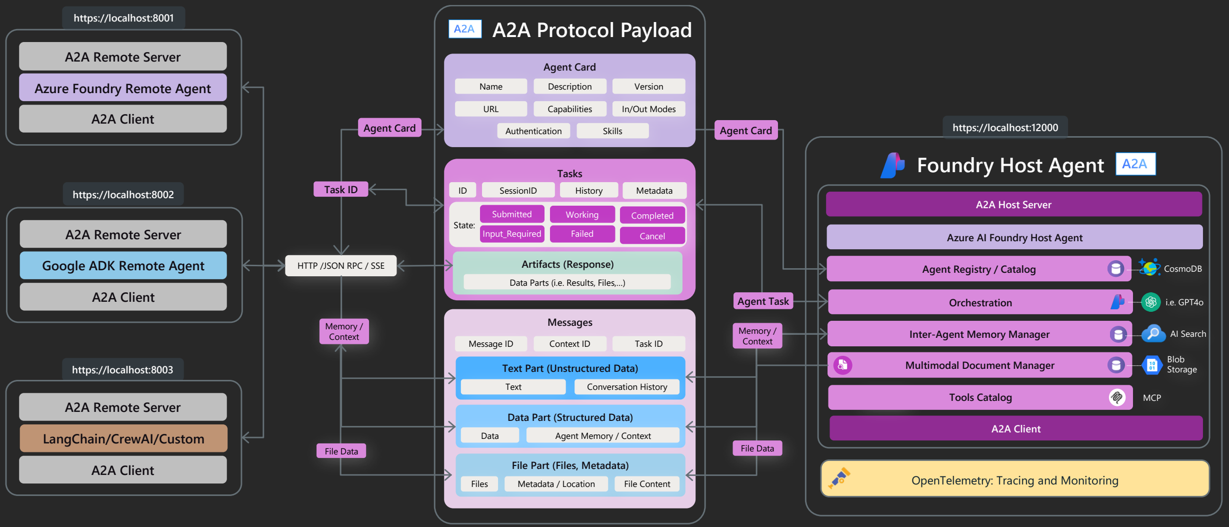


Figure 10. Tracing and Monitoring

A proper observability framework makes agent behavior visible, explainable, and auditable across the full lifecycle of an interaction. Rather than just logging events, agent observability captures intent, decisions, execution traces, performance metrics, and memory updates as structured telemetry that can be analyzed and inspected over time.

For multi-agent systems, observability becomes even more critical because:

- Agents make decisions autonomously and need to justify their reasoning trail.
- Root cause analysis requires tracing failures back through delegation chains and tool executions.
- Memory mutations, tool usage, and delegation events must be traceable back to who triggered what and why.
- Cross-agent coordination needs to be visualized as a distributed flow, not a sequence of isolated calls.

Azure AI Foundry: Built-in Observability Layer

With the Azure AI Foundry Agent SDK, tracing occurs in the context of agent runs rather than generic API calls. Each span can be enriched with agent identity, tool metadata, and A2A delegation context, allowing telemetry to follow the full chain of reasoning and handoffs across agents instead of fragmenting into isolated logs. This makes traces natively aware of A2A workflows, every tool usage, memory fetch, and delegation step appears as a correlated span in a single distributed trace.

Azure AI Foundry provides native observability hooks for agent-driven architectures. Each agent execution is automatically instrumented through tracing and telemetry pipeline, producing structured telemetry that captures:

- Agent runs, including duration, outcome status, and response metadata.
- Tool calls and responses, logged as discrete trace entries.
- Streaming and multi-modal interactions, tracked as message-level events.
- Latency metrics and failure points, including retries and exception traces.

Under the hood, this observability layer is powered by OpenTelemetry, an open standard for collecting and exporting traces, logs, and metrics. Azure exposes this through:

- Distributed trace views in Application Insights
- Agent-level dashboards in Azure Monitor
- End-to-end call flow visualization across tools, memory retrieval, and remote agents

Because our Host Orchestrator is built on top of Azure AI Foundry's agent runtime, we inherit this telemetry and observability fabric automatically, meaning every agent run, task delegation, and tool invocation is traceable without requiring custom instrumentation.

OpenTelemetry Integration in Our Architecture

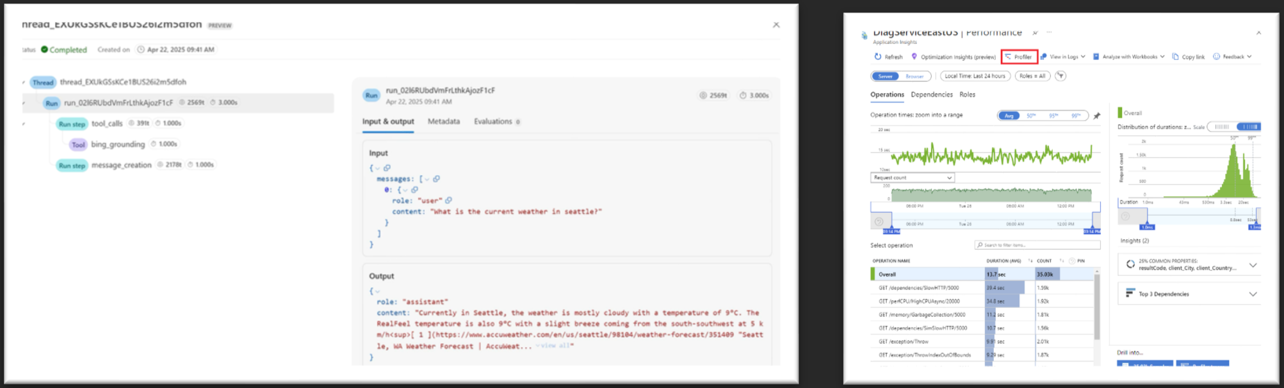
To extend this default observability, we add custom OpenTelemetry tags and spans at key decision points in our Host Orchestrator

- Before and after A2A delegation
- Memory retrieval operations

- Multimodal extraction events
- Tool invocations
- Routing decisions

Each of these spans is emitted into Azure's OpenTelemetry pipeline, allowing us to correlate agent reasoning steps with system-level metrics, e.g., Did memory retrieval impact latency? Which agents produce the highest success/error ratios? Are certain file types triggering additional retries?

Dashboards become real-time intelligence panels, showing agent throughput, success rates, error spikes, and cross-agent dependencies.



hread_EXUkGSsKCe1BUS26i2m5dfoh PREVIEW

Status ● Completed Created on Apr 22, 2025 09:41 AM

- Thread thread_EXUkGSsKCe1BUS26i2m5dfoh
  - Run run_02l6RUbdVmFrLthkAjozF1cF 2569t 3.000s
    - Run step tool_calls 391t 1.000s
      - Tool bing_grounding 1.000s
    - Run step message_creation 2178t 1.000s

Run run_02l6RUbdVmFrLthkAjozF1cF 2569t 3.000s
Apr 22, 2025 09:41 AM

Input & output | Metadata | Evaluations 0

Input

```
{
  messages: [
    0: {
      role: "user"
      content: "What is the current weather in seattle?"
    }
  ]
}
```

Output

```
{
  role: "assistant"
  content: "Currently in Seattle, the weather is mostly cloudy with a temperature of 9°C. The
  RealFeel temperature is also 9°C with a slight breeze coming from the south-southwest at 5 k
  m/h<sup>[ 1 ](https://www.accuweather.com/en/us/seattle/98104/weather-forecast/351409 "Seatt
  le, WA Weather Forecast | AccuWeat... view all"
}
```

DiagServiceEastUS | Performance
Application Insights

Server | Local Time: Last 24 hours | Roles = All

Operations | Dependencies | Roles

Select operation

OPERATION NAME	DURATION (AVG)	COUNT
Overall	13.7 sec	35.03k
GET /dependencies/SlowHTTP/5000	39.4 sec	1.59k
GET /perfCPU/HighCPUAsync/20000	34.8 sec	1.92k
GET /memory/GarbageCollection/5000	11.2 sec	1.81k
GET /dependencies/SimSlowHTTP/5000	10.7 sec	1.56k
GET /exception/Throw	9.91 sec	2.01k
GET /exception/ThrowIndexOutOfBounds	9.29 sec	1.87k

Insights (2)

25% COMMON PROPERTIES:
resultCode, client_City, client_Country...

Top 3 Dependencies

Figure 11. Azure AI Foundry Agent Tracing Figure and Application Insights

You can learn more about telemetry and observability for Azure AI Foundry Agent service here : https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/develop/trace-agents-sdk

Identity and Trust

In a multi-agent system, communication alone is not enough, every agent must have a verifiable identity. Without identity, agents cannot be trusted, audited, or granted controlled access to enterprise systems. Authentication becomes guesswork, authorization becomes manual, and malicious or misconfigured agents cannot be distinguished from legitimate ones.

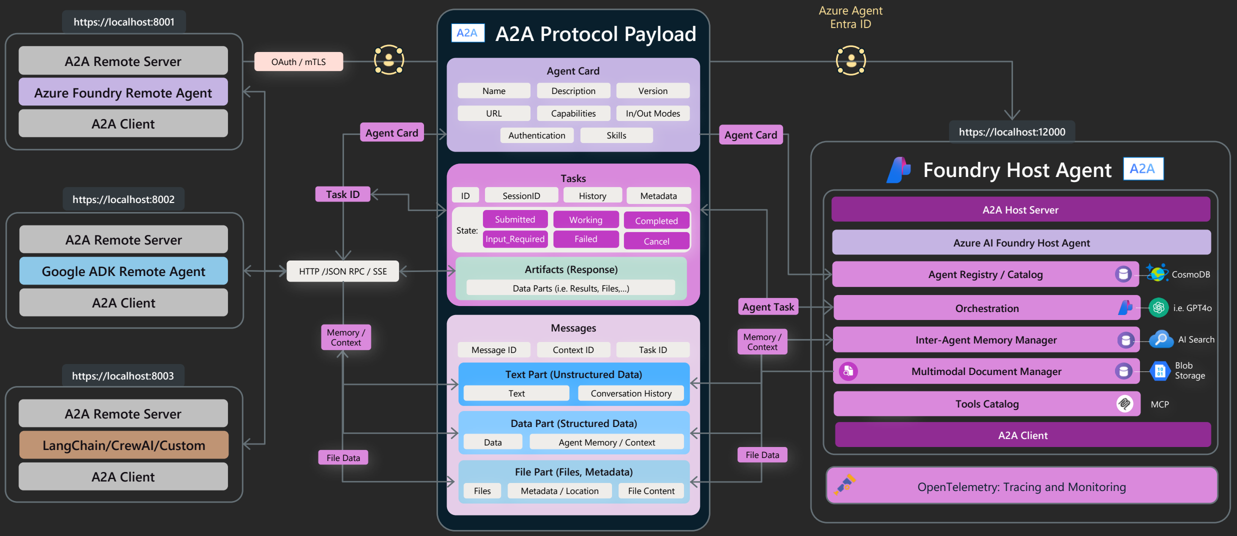


Figure 12. Agent Identity

Agent Identity establishes a foundation where every agent, human or autonomous, participates in the ecosystem with proof of origin, permissions, and accountability. This matters at scale because:

- Agents act on behalf of users, systems, and departments, the origin of an action must be attributable.
- Security boundaries must persist even in agent-to-agent conversations, delegation should never bypass access controls.
- Audit trails must tie back to identity, memory logs, A2A messages, and tool calls should reference who (which agent identity) initiated the action.
- Cross-organization agent collaboration requires federated identity, not hardcoded trust.

Azure Entra Agent Identity

Microsoft is introducing Azure Entra Agent ID, a framework that extends Entra ID (formerly Azure Active Directory) to autonomous agents as first-class identity objects. Just as users, applications, and services have Entra identities, agents will receive their own identity principals, capable of:

- Authenticating via OAuth / mTLS
- Receiving least-privilege access scopes (e.g., this agent can call ServiceNow APIs but cannot modify HR records)
- Being assigned to Entra Groups and Conditional Access Policies
- Participating in SIEM/Audit pipelines like any other identity
- This transforms agents from unbounded processes into governed digital entities inside Azure's security perimeter.

How This Would Work in Our Architecture

While not yet implemented in our solution, the identity model would integrate at the A2A message layer:

- Each agent participating in A2A would carry an Entra Agent Access Token in its message headers.
- The Host Orchestrator would validate the agent's identity via Entra Agent ID, enforcing trust boundaries before accepting a task or memory contribution.
- Tool calls (Foundry functions, MCP endpoints, external APIs) would run under the agent's own Entra Agent identity, not a generic service identity, enabling fine-grained policy enforcement and audit trails.
- Every memory write, retrieval event, or artifact annotation stored in Azure AI Search could be tagged with identity.agent_id metadata, allowing full traceability across time.
- Cross-tenant agents (e.g., an insurance carrier agent collaborating with a bank's fraud agent) could use federated Entra Agent ID trust, making cross-organization A2A networks secure and compliant without static credential sharing.

Identity as a Future Layer

By introducing Agent Identity via Azure Agent Entra, our architecture evolves from open interoperability to trusted interoperability, where communication, delegation, memory, and tool usage are all identity-bound and policy-enforced. This becomes essential for:

- Regulatory-grade audit logs (who initiated which retrieval or modification)
- Zero-trust agent networks where no agent is implicitly trusted by topology
- Cross-department or cross-enterprise collaboration through A2A, with full identity trace and revocation controls

Our Host Orchestrator is already structured in a way that would allow Entra Agent Identity to be inserted at the protocol level, making it a natural next extension of the platform.

To learn more about Microsoft Entra Agent ID please follow the link here: https://learn.microsoft.com/en-us/entra/security-copilot/entra-agents

Evaluation and Governance

As agents gain autonomy and begin making decisions, evaluation and governance become essential. In traditional software systems, governance is enforced through static rules, unit tests, and access controls. In multi-agent systems, governance must evolve; agents reason, adapt, and coordinate in dynamic environments, which means their behavior must be continuously evaluated against safety, accuracy, compliance, and business policy expectations.

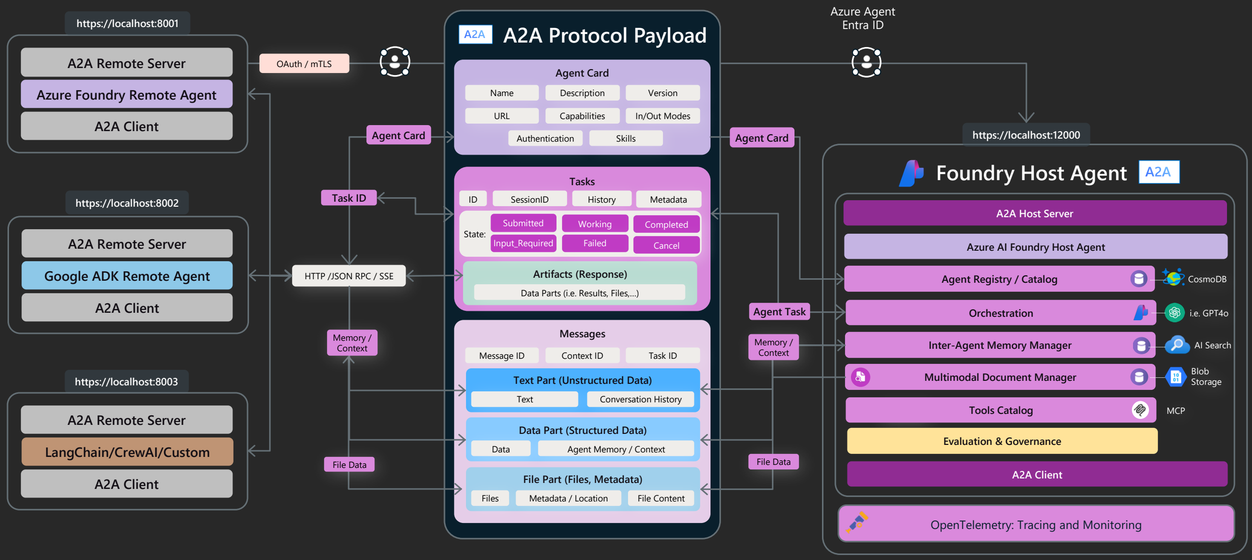


Figure 13. Evaluation and Governance

What Evaluation Means in Multi-Agent Systems

Evaluation in this context is not just model scoring, it is behavioral validation of the entire agent interaction flow, including:

- **Task-level correctness** — Did the agent produce the right action or insight?
- **Protocol compliance** — Was the A2A exchange valid, well-structured, and safe?
- **Memory and context usage** — Did the agent retrieve and apply relevant historical context, or hallucinate?
- **Tool governance** — Were function calls authorized, safe, and within policy boundaries?
- **Chain-of-thought / trace auditability** — Can the decision be reconstructed and justified?

Azure AI Foundry: Evaluators

Azure AI Foundry provides a built-in evaluation pipeline designed to assess both individual model outputs and full agent workflows. Evaluation is integrated into the lifecycle of agents, not bolted on afterwards, allowing organizations to define metrics, run scenario-based tests, and continuously monitor agent quality and safety using structured telemetry.

Azure AI Foundry ships with a set of standard evaluators that can be applied to raw model outputs or end-to-end pipelines. These include quality evaluators that measure coherence, completeness, and relevance; safety evaluators that flag harmful or off-policy content; and RAG evaluators that assess grounding, context utilization, and retrieval coverage when enterprise knowledge is involved. These evaluators can be composed into repeatable test suites or wired into continuous evaluation runs, producing structured telemetry that feeds directly into Foundry’s observability layer.

While these evaluators focus on what a model produces, Azure also introduces a new class of evaluators designed specifically to score how agents behave during multi-step workflows.

Azure AI Foundry: Evaluators for Agents

Azure AI Foundry supports evaluation across agent flows and model/tool invocations. On the agent side, Agent Evaluators (preview) introduce LLM-based scoring for behaviors such as:

- Intent Resolution — measuring whether the agent correctly interprets user goals or asks for clarification when appropriate.
- Tool Call Accuracy — verifying that the correct tools were invoked with appropriate parameters, without unnecessary or missing calls.
- Task Adherence — ensuring that the agent follows instructions and does not drift into irrelevant reasoning steps.

These evaluators are complemented by quality and safety metrics, including Relevance, Coherence, Fluency, Groundedness, Code Vulnerability Detection, Violence, Self-harm, and Hate content checks. Evaluations can be executed locally via the Azure AI Evaluation SDK or remotely in cloud mode (preview), with results automatically logged and surfaced in Azure AI Foundry’s monitoring views.

To learn more about Azure AI Foundry Agent Evaluators please visit the link here: https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/evaluation-evaluators/agent-evaluators

Azure AI Foundry: Continuous Evalations for Agents

Azure AI Foundry also introduces Continuous Evaluation for Agents (preview), a powerful capability that extends evaluation beyond static test runs. When enabled, Foundry can sample live agent interactions in production, automatically scoring a configurable percentage of real conversations against quality, safety, and compliance criteria. These evaluations run in the background and feed results directly into Foundry's observability dashboards, providing near real-time insight into agent behavior drift, tool misuse, intent misalignment, or emerging safety risks.

Because our A2A Host Orchestrator and Remote Agents exchange strictly structured protocol messages, every task, response, tool invocation, and memory reference naturally aligns with Foundry's evaluation schema. Rather than parsing unstructured transcripts, evaluation can operate directly on protocol-shaped interaction logs, enabling precise scoring of agent performance, behavioral compliance, and decision efficiency across the entire multi-agent network.

To learn more about Azure AI Foundry Continuous Agent Evaluation please follow this link: https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/continuous-evaluation-agents?

Governance Matters

Without governance, a multi-agent system can drift, produce inconsistent outcomes, or trigger unintended downstream effects in enterprise systems. Evaluation and governance ensure that:

- Agents remain aligned with policy and compliance frameworks
- Autonomous behavior remains observable, contestable, and correctable
- Teams can iteratively improve agents using structured feedback loops
- Agents can be promoted from sandbox to production safely

From a compliance and audit perspective, governance is the mechanism that makes agentic workloads reconstructable, defensible, and testable. High-risk use cases will increasingly face statutory duties for traceability and record-keeping (for example, the European Union Artificial

Intelligence Act requires event logs across the system lifecycle so actions can be traced and explained), which in practice means rigorous versioning of models/tools/prompts, data lineage, and tamper-evident interaction logs for every autonomous decision. Beyond regulation, leading frameworks converge on the same operational disciplines: NIST AI Risk Management Framework calls for systematic documentation and measurable controls to bolster transparency and accountability; ISO/IEC 42001 elevates this into an auditable management system that embeds policies, roles, and continuous improvement for AI across its lifecycle; and in regulated industries, model risk management expectations (e.g., SR 11-7) demand validation, use-governance, and documentation sufficient for independent parties to understand limitations, assumptions, and control efficacy.

In short, without governance that enforces logging, evidence collection, and control testing, enterprise agents cannot meet audit standards or prove compliance at scale.

Operationalizing Compliance and Security Controls

While Azure AI Foundry's evaluation framework provides the foundation for measuring agent behavior, enterprise deployments require these insights to trigger concrete security responses and compliance workflows. Evaluation telemetry alone cannot block threats in real time, correlate incidents across cloud infrastructure, or automatically generate audit evidence for regulatory frameworks. To close this loop, organizations need an integrated control plane that unifies runtime protection, security operations, and compliance management around the same AI workloads that AI Foundry evaluates.

Microsoft provides this through a layered approach that combines real-time threat prevention, security event correlation, and automated compliance tracking, ensuring that the behavioral insights surfaced by Foundry translate directly into actionable governance outcomes.

To operationalize safety and compliance alongside evaluation, Azure pairs Azure AI Content Safety with Microsoft Defender for Cloud and Microsoft Purview Compliance Manager as a closed-loop control plane.

Azure AI Content Safety sits on the invocation path to detect and block jailbreaks and other adversarial inputs—including document-borne prompt injections—before tools or data are touched, with results surfaced in Foundry and available to downstream security systems.

Microsoft Defender for Cloud then ingests those safety outcomes and emits AI-specific security alerts (for example, "Jailbreak attempt … was blocked by Azure AI Content Safety Prompt Shields"), enabling SOC workflows, correlation, and incident response across cloud resources.

Microsoft Purview Compliance Manager maps these Defender signals to automated improvement actions and assessments, updating your compliance score without manual evidence collection, while Purview Data Security Posture Management (DSPM) for AI provides an AI-centric lens to discover usage, apply data controls, and continuously monitor risk across multicloud estates.

Looking ahead, the Azure AI Red Teaming Agent can be introduced as a future extension of our A2A ecosystem—acting as an autonomous adversarial tester wired into the same orchestration fabric. Using PyRIT-based attack strategies, it can continuously probe agent endpoints, score Attack Success Rate, and feed those results into the existing evaluation layer. By treating red teaming as just another agent role rather than a separate security exercise, we set the foundation for a system that doesn't just operate and get evaluated, but actively stress-tests itself as part of the workflow.

This integrated control plane transforms governance from a bottleneck into an enabler of scale. By embedding real-time protection, automated compliance tracking, and unified security observability directly into the agent runtime, enterprises can deploy multi-agent networks with confidence, knowing that every interaction is protected, every incident is surfaced, and every regulatory obligation is continuously tracked.

Rather than forcing teams to choose between velocity and control, this approach ensures that agentic workloads can expand across business units, geographies, and use cases while maintaining the trust, auditability, and risk management that enterprise stakeholders demand.

As agent networks grow more distributed and autonomous, Azure AI Content Safety, Microsoft Defender for Cloud, and Microsoft Purview Compliance Manager provide the operational guardrails that keep innovation aligned with organizational policy, delivering on the blueprint's promise of agents that are not only intelligent and collaborative, but also secure, compliant, and enterprise-ready.

Learn more about Azure AI Content Safety Prompt Shield here: https://learn.microsoft.com/en-us/azure/ai-services/content-safety/concepts/jailbreak-detection

Learn more about Alerts and AI threat protection here: https://learn.microsoft.com/en-us/azure/defender-for-cloud/alerts-ai-workloads#ai-services-alerts
https://learn.microsoft.com/en-us/azure/defender-for-cloud/ai-threat-protection

Learn more about Microsoft Purview Compliance Manager here: https://learn.microsoft.com/en-mpliance-manager-improvement-actions#microsoft-defender-for-cloud-automation

You can learn more about Azure AI Red Teaming Agent here: https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/ai-red-teaming-agent

Front End

While agents operate autonomously via A2A, the Host Orchestrator also serves as the primary entry point for human interaction and control (UX/UI). In our implementation, a Next.js–based frontend provides a unified interface where users can register and discover agents and humans into their session, trigger multi-agent workflows, inspect agent capabilities, submit multimodal inputs, and collaborate alongside agents in real time through a chat interface.

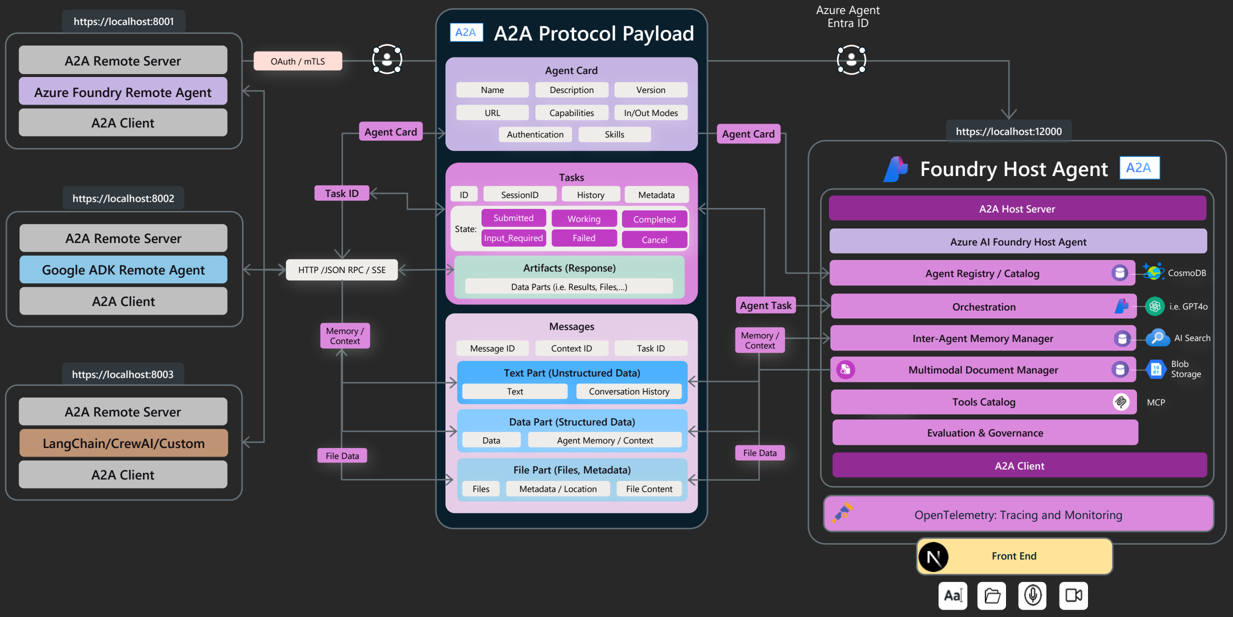


Figure 14. Fronted UI

The frontend exposes several core capabilities:

- **Agent Catalog & Agent Cards** — Users can browse A2A-registered agents, inspect their capabilities, and register and initiate tasks directly against them.
- **A2A Task Interface** — The UI visualizes active tasks, agents involved, task states, artifacts generated, and memory context flowing between agents.
- **Multimodal Input Panel** — Users can upload PDFs, images, spreadsheets, or code files directly through the chat interface. These are packaged into A2A FilePart and DataPart payloads and routed through the Host using the same protocol used for autonomous agents.
- **Human-in-the-Loop Control** — Users can pause, override, or redirect agent flows,

making the UI a collaborative command surface where human intent and agent execution align through A2A.

- **Live Observability** — The frontend can surface agent trace snapshots, reasoning timestamps, and task diagnostics directly in the interface through it’s thinking box.

Behind the scenes, a persistent WebSocket channel connects the frontend and backend, streaming real-time agent events, task updates, and user actions. Every A2A message — from task delegation to file exchange — flows through this channel, making the UI a live reflection of agent reasoning and orchestration state. This bi-directional link allows the frontend not only to observe but also to participate — dispatching new tasks, uploading files, or interrupting workflows with minimal latency.

By exposing the same A2A protocol used for agent-to-agent communication, the frontend becomes a collaborative control surface rather than a passive interface. Users can inspect agents, artifacts, and memory in real time — and step into the loop to correct or guide execution without breaking protocol.

The Next.js implementation offers a lightweight foundation for this interaction layer, providing task visualization, multimodal uploads, and human-in-the-loop controls. While minimal by design, it can be easily extended with richer workflows, customized UI patterns, or domain-specific branding.

Closing Note

This work is a starting point, not a destination. By grounding the system in open protocols (A2A, MCP) and Azure services, we've shown a path to compose agents that can discover, coordinate, use tools, share memory, and remain observable and governable. The intent is reusable patterns, not a fixed stack: replace components, extend the catalog, adjust orchestration, tighten identity, and deepen evaluation as your requirements evolve. Azure gives us the rails; the blueprint shows how to ride them at scale; the rest is invention and innovation.

Adoption can be incremental. Begin with a small workflow, introduce memory and tool routing, then layer in continuous evaluation and policy. As capabilities grow, the same protocol shapes carry across teams and clouds, letting agents interoperate without bespoke bridges. Over time, this shifts from individual automations to a network of agents that collaborate reliably under shared standards and controls.

If you adapt this blueprint, contribute back what you learn, new remote agent, better evaluators, improved routing and orchestration, federated agent catalogs, better memory and adaptability. The value here is the pattern: a practical foundation for building and scaling multi-agent systems that are interoperable by design and accountable in operation.

The future of enterprise AI will not be defined by one model or platform, but by connected systems where networks of agents and humans collaborate through shared protocols and open standards. What you have here is the first step toward that network.

References

2. - Google A2A Github Repository: https://github.com/a2aproject/A2A

3. - Azure AI Foundry Agent service: https://learn.microsoft.com/en-us/azure/ai-foundry/agents/overview

4. - Azure CosmosDB service: https://azure.microsoft.com/en-us/products/cosmos-db

5. – Microsoft Agent Framework: https://azure.microsoft.com/en-us/blog/introducing-microsoft-agent-framework/

6. - Azure AI Search: https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search

7. - Azure Content Understanding: https://azure.microsoft.com/en-us/products/ai-services/ai-content-understanding

8. - Azure Blob Storage: https://azure.microsoft.com/en-us/products/storage/blobs

9. - Azure AI Agent Service MCP support: https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools/model-context-protocol

10. - Azure AI Foundry Tracing and Telemetry: https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/develop/trace-agents-sdk

11. - Microsoft Entra Agent ID: https://learn.microsoft.com/en-us/entra/security-copilot/entra-agents

- Inventory and Discovery MCP Servers API Center: https://learn.microsoft.com/en-us/azure/api-center/register-discover-mcp-server

- Azure API Management MCP Server - https://learn.microsoft.com/en-us/azure/api-management/mcp-server-overview

12. - Azure AI Foundry Agent Evaluation: https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/evaluation-evaluators/agent-evaluators

13. - Azure Continuous Agent Evaluations: https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/continuous-evaluation-agents

14. - Azure AI Foundry Content Safety Jailbreak Detection: https://learn.microsoft.com/en-us/azure/ai-services/content-safety/concepts/jailbreak-detection

15. - Azure Defender for Cloud – AI Alerts: https://learn.microsoft.com/en-us/azure/defender-for-cloud/alerts-ai-workloads#ai-services-alerts

16. - Azure Defender for Cloud: - Threat Protection https://learn.microsoft.com/en-us/azure/defender-for-cloud/ai-threat-protection

17. - Microsoft Defender for Cloud Automation: https://learn.microsoft.com/en- mpliance-manager-improvement-actions#microsoft-defender-for-cloud-automation

18. - Azure AI Foundry Red-teaming Agent: https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/ai-red-teaming-agent

19.

20.

21. -

-