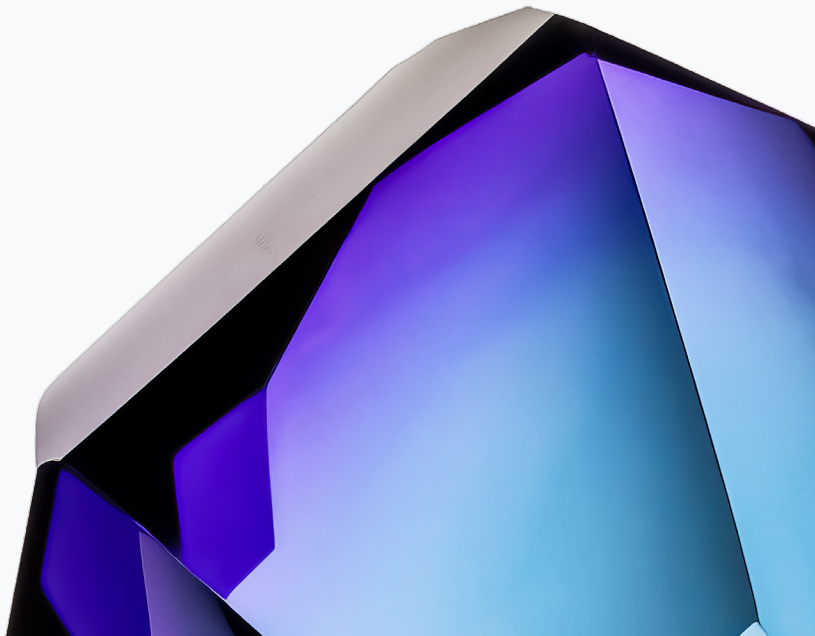
Agent Tools & Interoperability

Authors: Kanchana Patlolla, Łukasz Olejniczak, and Pier Paolo Ippolito

Google



Acknowledgements

Content contributors

Alan Blount

Mike Smith

Anant Nawalgaria

Lukas Geiger

Curators and editors

Anant Nawalgaria

Designer

Michael Lanning

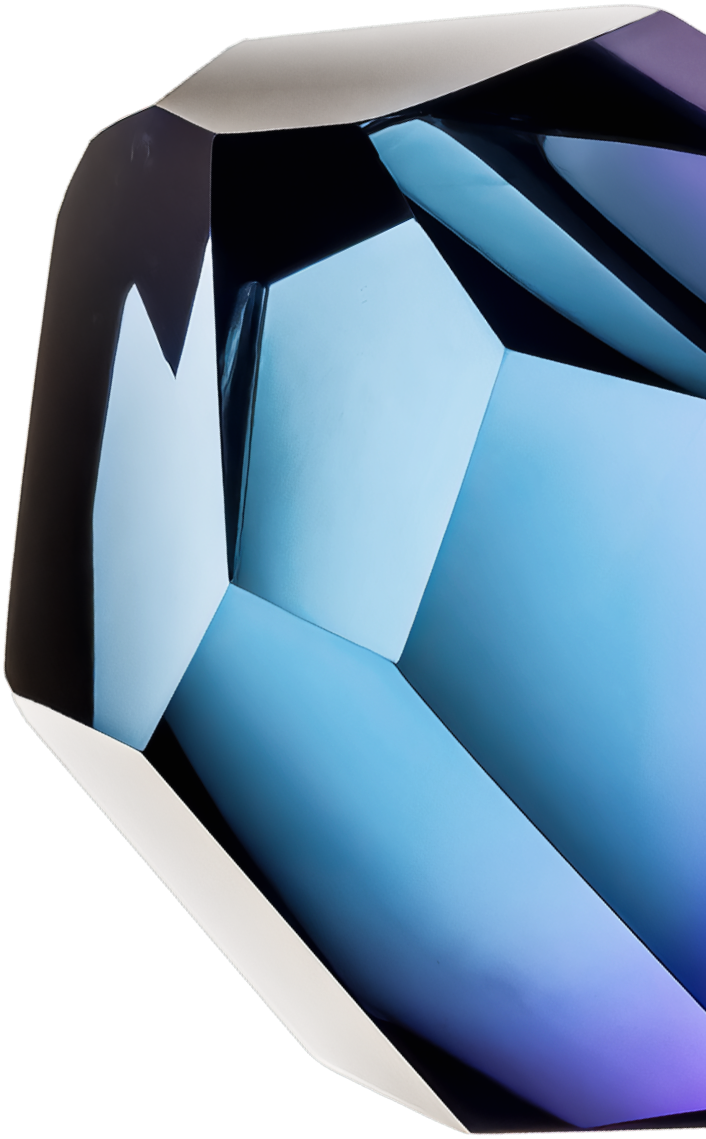


Table of contents

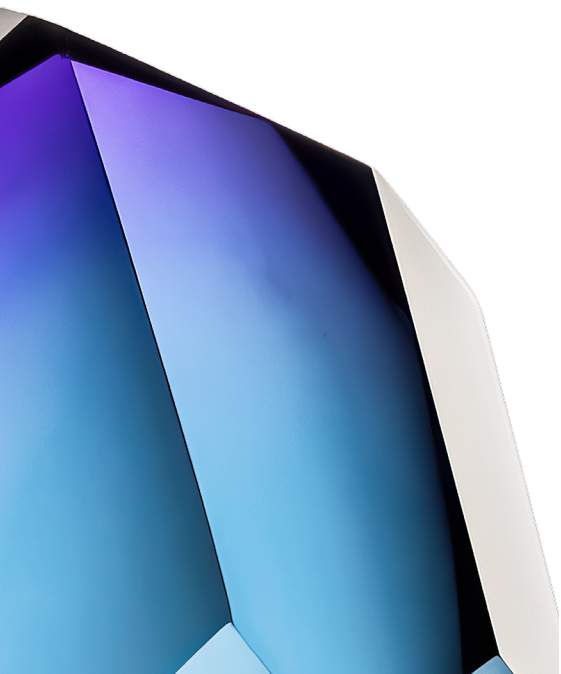


Table of contents

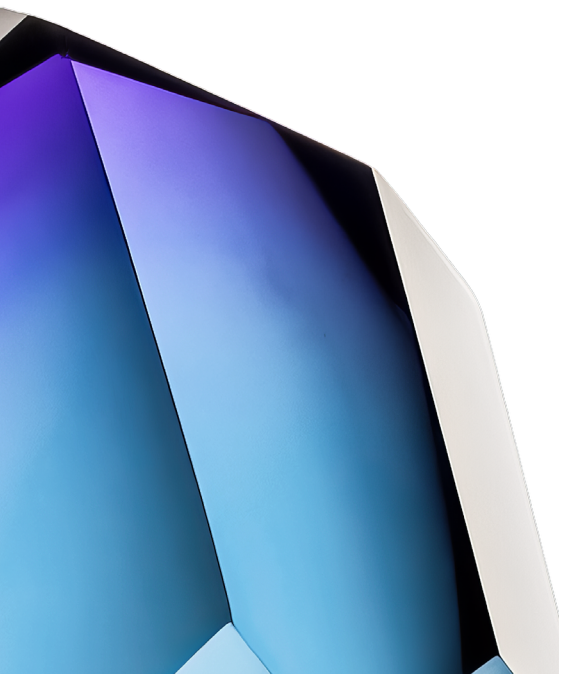
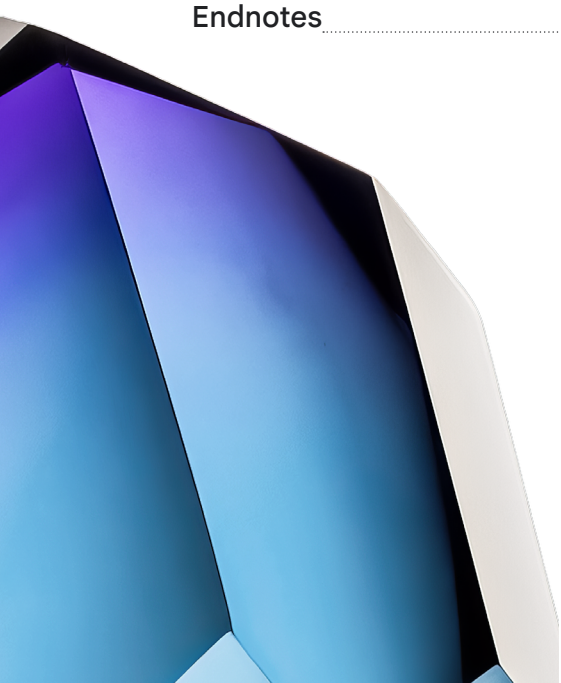


Table of contents



> Software's next evolution isn't written: it's orchestrated by interoperable agents.

Introduction

In Day 1, we outlined the paradigm shift from traditional software development to Agentic Engineering, introducing the Factory Model where your primary output as a developer is no longer raw syntax, but the system that produces code. We defined the core architecture of this system:

Agent = Model + Harness

If Agentic Engineering represents the factory floor you are orchestrating, then **MCP**, **A2A**, **A2UI**, **AP2**, and **UCP** are the **Industry Standards**—the uniform nuts and bolts and screw sizes, data formats, and communication channels—that allow your machinery to safely interact with the rest of the world.

Without these open protocols, every agent you build exists as an isolated "custom machine" in a garage. You are forced to spend your hours and tokens writing fragile, bespoke wrappers for every single tool and API connection, trapping you in a low-leverage **Conductor** role.

By adopting standardized interoperability layers, you transform your agent's **Harness** into a modular, plug-and-play platform. You spend less time debugging custom JSON payloads and more time directing high-level intent as an **Orchestrator**.

- **OpenResponses & Interactions API** are both **“Power Plugs”**, modern API approaches to LLM inference which support long running tasks. These blur the line between a stateless single turn and a stateful agent.
- **MCP (Model Context Protocol)** acts as the **"USB-C"** within your agent's harness, instantly connecting models to databases, filesystems, and web APIs.
- **Skills** are **“Playbooks”**, very simple markdown instructions and scripts or tools which can be used in a sandbox environment like a terminal.
- **A2A (Agent-to-Agent)** serves as the **"Factory Radio"**, allowing specialized agents to negotiate, brain-storm, and delegate tasks to each other.
- **A2UI (Agent-to-User Interface)** behaves like a **"Generative Display Window"**, turning raw, complex JSON outputs into safe, interactive visual components for human operators.
- **AP2 and UCP** act as the **"Global Supply Chain & Transaction Network"**, allowing agents to securely negotiate and execute autonomous commercial transactions.

This paper focuses on how vibe coders can rapidly utilize some of these protocols to construct a virtual data and execution team in a single afternoon.

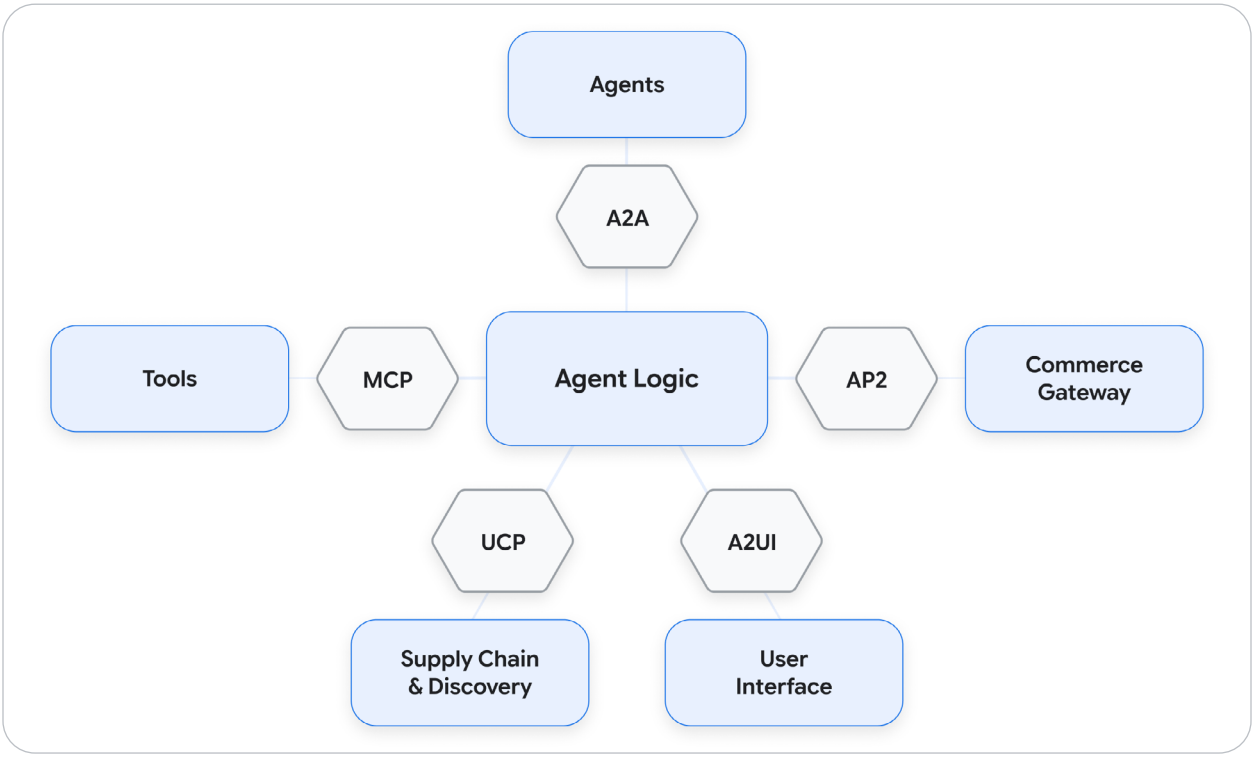


Figure 1: Ecosystem of Agent Protocols

Why this paper, why now

In the era of vibecoding with less structure, harnesses and protocols help build trust in the agent development process. While velocity of outcome remains the primary driver for practitioners and developers, standardized protocols allow us to expand much further in achieving complex goals by transforming isolated "custom machines" into modular, interoperable platforms. Without agreed upon open standards, developers are creating tech

debt, each API is a standard-of-one. These are low-leverage tasks, writing fragile, bespoke wrappers for every tool, maintaining them over time and adapting to other's needs. Adopting these layers allows for a shift from being a mere builder to a high-level orchestrator.

Who this paper is for

This paper is specifically designed for software engineers, engineering managers, architects, and technical leaders who recognize that the shift toward Agentic Engineering requires strict adherence to protocols to maintain the fidelity and reliability of outcomes. It serves as a practical guide for "vibe coders" who prioritize speed and visual results, demonstrating how to construct a virtual data and execution team.

💡 Applied Tip:

In the Day 1 whitepaper, we recommended using an `AGENTS.MD` file to provide standard guidance for coding agents.

Always begin by thinking deeply before you code—explicitly stating your assumptions, surfacing tradeoffs, and halting to ask for clarification the moment you encounter ambiguity rather than guessing silently.

Write only the absolute minimum amount of code required to solve the immediate problem, strictly avoiding speculative features, unrequested abstractions, or predictive configurations. When editing existing code, make highly surgical changes by restricting your updates only to the exact lines necessary to fulfill the request, maintaining the existing style perfectly, and leaving adjacent, unbroken code completely untouched unless your changes directly orphaned an import or variable.

Finally, approach every task through goal-driven execution by breaking it down into a clear, step-by-step plan with strong success criteria, such as writing a reproducing or failing test first and independently looping through verification until that specific goal is strictly met.

In our previous whitepaper, Agent Tools & Interoperability with Model Context Protocol (MCP), we laid out the enterprise architecture of MCP, detailing host-client-server topologies, custom server creation, and security governance.

For the vibe coder, however, the priority is **consumption over creation**. You do not want to spend hours writing custom server configurations. You want to hook into existing public and private registries to instantly give your agent "plug-and-play" superpowers. This section covers how to consume MCP servers efficiently, how to bypass the NxM integration crisis, and how to debug transport layers when things break.

Note: we will have a deeper look at Agent Skills in the next whitepaper and Security in the following.

The Vibe Coder's View of MCP: Discovery, Configuration, & Connection

The traditional enterprise way of bringing tools to an LLM is **bespoke hardwiring**. Every new connection requires custom REST wrappers, API keys, token refresh policies, and custom JSON parsers. MCP replaces this friction with a standardized socket. Setting up your coding agent involves three simple steps:

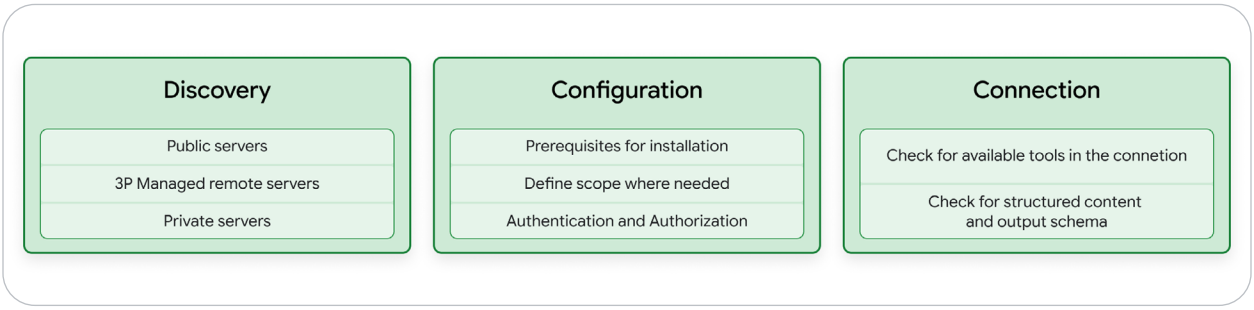


Figure 2: Steps for Onboarding an MCP Server

Discovery

Vibe coders do not write custom connectors; they locate pre-built MCP servers from three primary sources:

- **Public MCP Registries:** Publicly available registries host hundreds of pre-built servers (e.g., registry.modelcontextprotocol.io, github.com/mcp). These are excellent for rapid local prototyping but are unvetted and used at your own risk.
- **Third-Party (3P) Remote MCP Servers:** Vetted platforms expose hosted endpoints. Discover managed pre-vetted MCP servers from official sources. For instance, official Google-published MCP servers (e.g., Google Maps, BigQuery, Google Docs) let your agent securely interact with managed services.
- **Internal Registries:** Inside an organization, internal tools are exposed as MCP servers and cataloged in a secure registry. The technical implementation of this registry is typically managed via an API gateway, GCP Agent Registry, or a private microservice portal.

> 💡 **Applied Tip:**
>
> When making a decision always consider security as a first priority, look for official instructions where possible.
>
> If using public community servers do not pass credentials, consider using services like Model Armor to avoid security issues.

Configuration

Before connecting, specify standard parameters detailing the server's scope and permissions. This includes setting up environment files to handle API credentials (such as Personal Access Tokens or OAuth secrets) and defining read/write permissions to protect your filesystem.

💡 **Applied Tip:**

- ✅ Check for pre-requisites
- ✅ Identify scope, access criterias
- ✅ Include the specifications in the coding agent
- ✅ Authentication

Connection

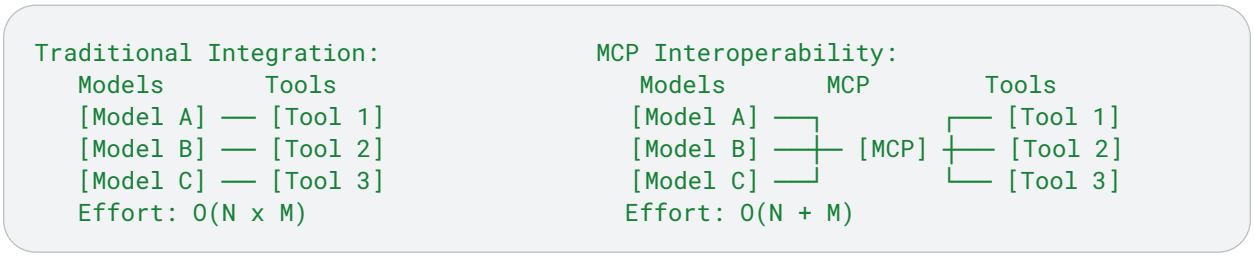
Once configured, the client establishes connection with the server. To verify the connection, the host client runs a basic handshake request to list the available tools and validate the output schema.

Bypassing the NxM Prototyping Problem

To appreciate why MCP is crucial for rapid prototyping, look at the integration math. If you are experimenting with N different LLMs (Gemini 3.1 Pro, Gemini 3 Flash, Gemini 3.5 Flash or a local open-source model) and want to connect them to M different external tools (Jira, BigQuery, GitHub, Google Drive), traditional ad-hoc development requires writing custom integration code for every single model-tool intersection:

$$\{\text{Integration Complexity}\} = O(N \times M)$$

If you have 5 models and 10 tools, you must maintain **50 bespoke integration points**. If a single tool API changes, multiple parser loops break without a protocol in place.



Snippet 1: MCP Complexity

MCP reduces this complexity to linear scale:

$$\{\text{Integration Complexity}\} = O(N + M)$$

Why is this important?

By standardizing tool definitions, you connect standard transports directly into your agent's Harness without writing integration code:

- **stdio (Standard Input/Output):** Most often used with Local and Prototyping efforts. Your coding agent can run without a complex network connection setup, as it treats the tool as a local process. The host client launches the MCP server as a local background subprocess, passing JSON-RPC 2.0 messages over stdin and stdout.
- **SSE (Server-Sent Events) over HTTP:** The host client (could be Local or a deployed agentic application) connects to a remote MCP endpoint over standard web protocols, streaming data to the agent in real-time. This has many advantages over the studio approach, fewer dependencies, always up to date, smaller footprint, and simpler lifecycle, but is a higher burden on the cloud hosted MCP server.

Both options enable the vibecoders to adopt the platform without multiple custom integration layers.

Debugging Issues with MCP Servers

When your agent hallucinates parameters, calls the wrong tool, or fails to parse a payload, don't waste time blindly modifying your system instructions. Debug the transport pipes directly:

- **MCP Inspector:** A native developer tool that runs a local web panel. It lets you manually query any local or remote MCP server, view the active tool schemas, manually test payload inputs, and inspect the raw JSON-RPC 2.0 packets without initiating your main agent workflow.
- **Chrome DevTools:** Perfect when running web-based development environments or debugging SSE connections, allowing you to trace incoming web streams and check for server latencies.

💡 Vibe Coder Toolkit: Best Practices for MCP Consumption

To maintain speed without sacrificing stability, vibe coders should adhere to the following guidelines when consuming MCP servers:

✅ Do's (Best Practices)

- Do audit public servers before connection: Always review the code of publicly available, open-source MCP servers before attaching them to an agent that has access to your local file system or credentials.
- Do use RAG for tools: Keep your agent's context window clean. Dynamically load tools from a registry only when needed, and drop them from context when the task is complete to prevent attention dilution.

- Do leverage internal API Gateways and registries: If possible, rely on internal tool registries. This ensures you are consuming approved, governed data schemas rather than reinventing the wheel or running unvetted code.
- Do use the MCP Inspector: When an agent hallucinates a tool call with their arguments, use the MCP Inspector or Chrome DevTools to look at the raw transport data rather than blindly tweaking the agent's system prompt.
- Do include HITL: Show tool inputs to the user before calling the server, to avoid malicious or accidental data exfiltration
- Do auditing needs: Log tool usage for audit purposes

❌ Don'ts (Common Pitfalls)

- Don't build if you can consume: Resist the urge to write custom REST API wrappers. Search for an existing MCP server first to maintain the "universal outlet" philosophy.
- Don't use public, unverified MCPs in production: Open-source MCP servers are fantastic for weekend prototypes, but they carry security and reliability risks. Do not tie core business logic to unverified public endpoints or API wrapper code.
- Don't hardcode credentials: During the Configuration phase, never paste API keys or auth tokens directly into your agent's prompt or local scripts. Rely on environment variables to pass credentials to the MCP server.
- Don't connect to production: Use the MCP server with a development project, not production. LLMs are great at helping design and test applications, so leverage them in a safe environment without exposing real data. Be sure that your development environment contains non-production data (or obfuscated data).
- Don’t use it for updates: If you must connect to real data, set the server to read-only mode, which executes all queries as a read-only.

- Don’t provide wide access to all the Projects: Scope your MCP server to a specific project, limiting access to only that project's resources where necessary and possible.

Now we have learnt how to enable vibecoders to leverage the MCP tools for their AI Agent building, but AI agent building goes beyond tools, it is to enable users to bring interoperability across the ecosystem of applications. We will review how Agent interoperability works in the next section.

Agent-to-Agent (A2A) Interoperability

As AI systems transition into distributed networks of specialized domain experts, standardized communication becomes essential for scaling. This section explores the Agent-to-Agent (A2A) protocol as the foundational layer that resolves ecosystem fragmentation, enabling developers to discover, orchestrate, and monetize a globally interoperable virtual workforce.

The Evolution of Agentic Architectures

To understand where **Vibe Coding** fits into the evolution of agentic architectures, we can observe a recurring pattern in technology: the shift from manual, low-level configuration to high-level, intent-based, declarative orchestration when users don’t say HOW to build something but WHAT they need.

A parallel trend is found in the history of IT infrastructure where we transitioned from manual configurations to "Infrastructure as Code" declarative scripts. A similar trend happened in the history of Machine Learning. The visionary premise from Google behind Automated Machine Learning (AutoML) was to democratize AI by automating the "drudge work" of model creation:

> *"One way we hope to make AI more accessible is by simplifying the creation of machine learning models called neural networks. Today, designing neural nets is extremely time intensive... That's why we've created an approach called AutoML, showing that it's possible for neural nets to design neural nets. We hope AutoML will take an ability that a few PhDs have today ...."* [1]

Today, technology enables us to vibe code entire applications. However, the trajectory of agentic architectures used to vibe code these applications is currently mirroring one of the most significant shifts in software history: the transition from **Monolithic to Microservices architectures**.

Just as early web applications were built as single, massive codebases where every function (billing, UI, database) was tightly coupled, many initial vibe coding projects relied on a **"Swiss Army knife" Single Agent Monolith**. *This involves relying on a single, highly sophisticated prompt where one agent wears multiple hats with many connected tools to handle everything from database queries to UI rendering and testing.* While this allows a developer to wire together a prototype in a weekend, it eventually hits the same "Monolithic Ceiling" that plagued early web apps:

- **Scaling Friction:** You cannot optimize just the "Database logic" without potentially confusing the "UI logic" residing in the same prompt. Also, the more tools a single agent has access to, the worse its decision-making becomes. *The search space for its next action is simply too large, leading to hallucinated parameters or triggering the wrong tools.*

- **Contextual Overload:** *Shoving overarching system instructions, dozens of complex tool schemas, and ongoing conversation history into a single prompt quickly maxes out the model's working memory.*
- **The Single Point of Failure:** A bug in one "tool" or instruction can cause the entire agent to hallucinate or crash. Irrelevant or corrupted data stored in context gets carried over, breaking future reasoning.

To overcome these bottlenecks one can follow a similar blueprint laid out by ML engineers and software engineers years ago. When early AutoML models successfully proved their business value, teams rarely left those auto-generated black boxes running the core business. To reach production scale they had to break this black box apart into observable, maintainable stages—data versioning, feature stores, drift detection. By specializing the components they applied a fundamental law of system design: **specialization as a scaling mechanism**.

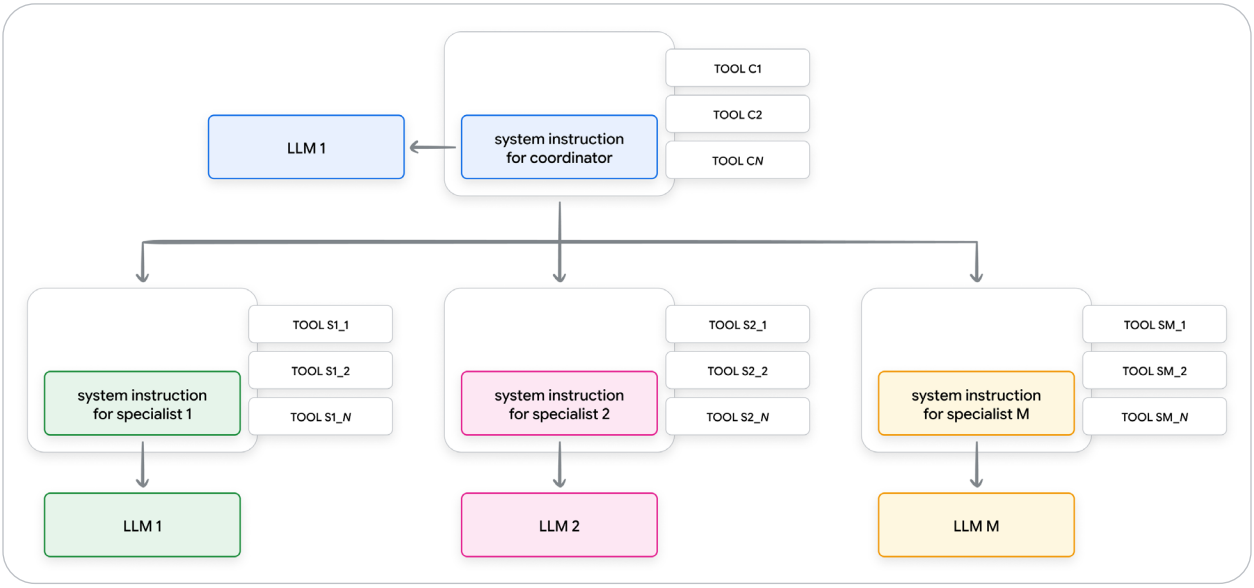


Figure 3: Monolithic Multi-agent Architecture

Internal specialization works by logically partitioning a monolithic agent into distinct, purpose-built sub-agents. Each agent is governed by a highly focused system prompt and a relevant subset of tools. It is still monolith because these internal sub-agents do not communicate across network boundaries. Instead, they share the same runtime and underlying memory.

This logical partitioning allows developers to modularize complex definitions while maintaining the low-latency execution and simplified state management inherent to a single-process application. At the same time this modularity addresses the inherent limitations of single-agent systems through several key technical improvements:

- **Reduction of Search Space:** By restricting a sub-agent's tools and skills (e.g., giving a DB agent only query tools), we drastically reduce tool-call errors and hallucinations.
- **Mitigation of Attention Dilution:** Specialization allows the underlying LLM to focus on a single domain prompt, leading to sharper reasoning.
- **Optimization of Contextual Load:** Because the orchestrator routes tasks instead of processing the entire logic tree, sub-agents maintain a high signal-to-noise ratio in their context windows.

As agentic architectures mature, a significant ecosystem shift based on distributed multi-agent architectures is underway. Industry leaders—including Google, Salesforce, ServiceNow, and Workday—are moving beyond the provision of APIs. These organizations are deploying domain-specific AI agents designed to navigate their proprietary ecosystems with native precision.

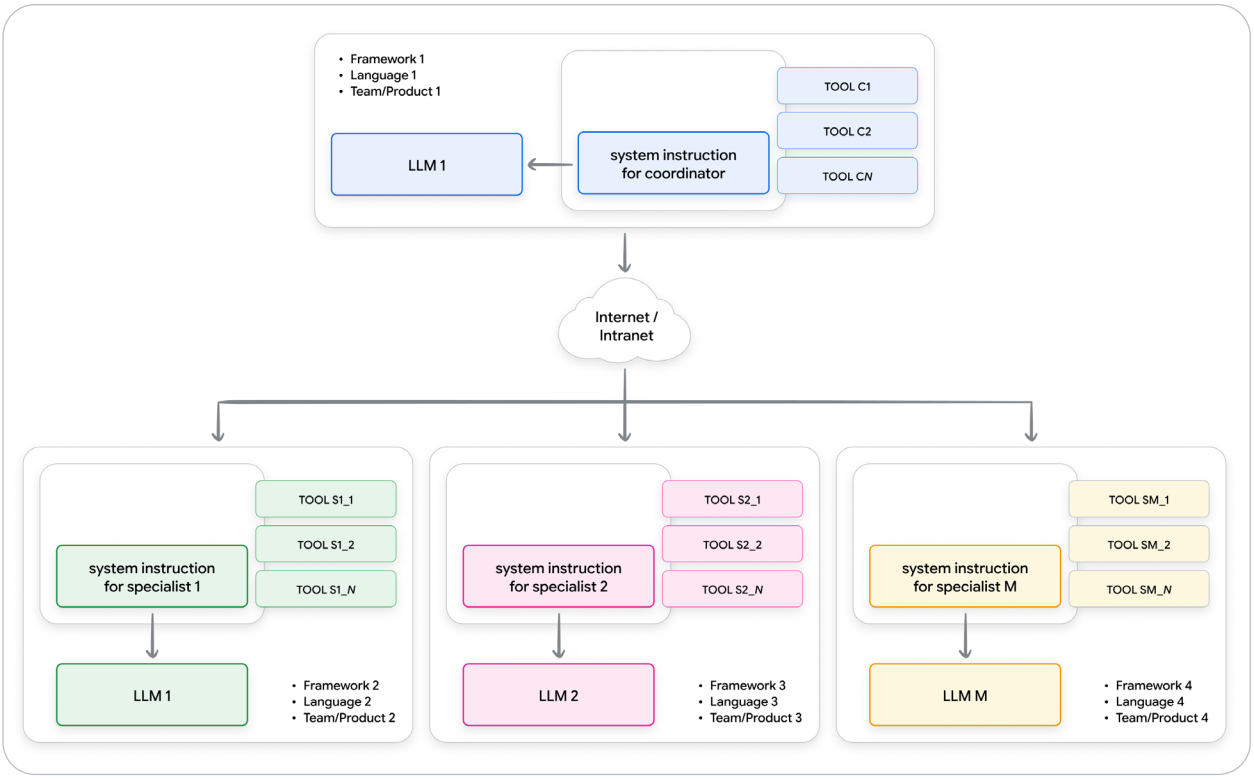


Figure 4: Distributed Multi-Agent Architecture. An orchestrator delegates tasks across network boundaries to remote, domain-specific agents.

This shift presents a strategic opportunity for developers. However, to reach the next tier of scale, technical and business leaders must ruthlessly prioritize their focus through a modern "Build vs. Buy" lens.

While it is technically feasible to construct a multi-agent system using custom-built sub-agents for third-party platforms (e.g. custom subagent for ServiceNow or custom subagent for Salesforce etc), this approach introduces a significant Maintenance Tax. By opting for bespoke development of domain specialists, the developer assumes full responsibility for updating prompt logic and tool definitions reflecting upstream product updates and API schema changes.

A better architectural strategy for building high-value applications would be to leverage official specialist agents. These agents are maintained by teams with deep domain expertise, ensuring maximum reliability and performance. By offloading the maintenance of specialist domain logic to these official entities, the **Orchestrator**—and by extension, the developers—can focus entirely on the unique user value and core innovation of their application.

However, attempting to orchestrate this virtual team of distributed AI specialists introduces a new bottleneck: **fragmentation**. Every one of these specialist agents can be built by a different team, using different technologies. Google's agent might be written in Python, Go or Java using ADK, Salesforce's agent might run on a LangChain framework, and Workday's might use something completely bespoke. They exist outside our network boundaries, they speak different languages, expect completely different payload structures, process conversational state differently, and rely on varying transport layers.

If a developer has to write custom integration code and handle bespoke error-correction loops for every single specialist they want to hire, the "Virtual Team" concept instantly turns into an integration endeavor. The maintenance tax of keeping all those custom bridges from collapsing would consume the entire project.

This chaos of fragmentation is exactly what the **Agent-to-Agent (A2A) protocol** [A2A] is designed to standardize. A2A, originally developed by Google and now donated to the Linux Foundation, introduces a universal layer of communication for agentic systems. It acts as the lingua franca for the AI ecosystem, abstracting away networking transport nuances, the underlying frameworks, programming languages, and payload disparities.

It ensures that the central Orchestrator can discover, onboard, and collaborate with any specialist agent in the ecosystem, completely agnostic to how that specialist was built under the hood. Just as HTTP standardized the web, **A2A standardizes the virtual workforce**.

> *"Does the caller need a result, or does the caller need another participant to take responsibility?" — that's the cleanest framing I've seen for this decision. The smell test of a central agent prompt growing into an accidental workflow engine is exactly how most teams discover they needed collaboration semantics three months too late.*[19]

But this raises the next architectural question: *If we are just delegating to external specialists, why can't we just treat them as standard tools?*

The nature of the engagement with specialists and tools is fundamentally different. Imagine a homeowner renovating a kitchen. They face a choice: *purchase the individual tools and manuals to attempt the build themselves, or delegate the project to a domain specialist who builds kitchens for a living*.

Tools are just passive instruments; a specialist is a collaborative partner. When you hire that specialist, you don't just hand them a single blueprint and walk away expecting a perfect kitchen to magically appear. They will hit edge cases or an oversight in your original design requiring them to pause, consult you on trade-offs, and resume.

Bounded vs. Unbounded Domains

This is exactly one of the reasons why treating a specialist AI agent as a simple tool does not scale. A standard tool or API operates on a **fire-and-forget** mechanism. It expects a single, perfectly formatted request (the blueprint payload) and returns a response. Real-world software environments often contain the digital equivalent of crooked walls: ambiguous data structures, misleading requirements, and conflicting user preferences.

yAn agent, however, operates in an *unbounded* problem-solving space. Real-world software environments are full of ambiguous data structures, misleading requirements, and conflicting user preferences. You rarely can specify all necessary details in a complex workflow without multi-turn clarification. The digital equivalent of *crooked walls*.

The GOTO Problem in Agentic Architecture

Because an agent's domain is unbounded, trying to force an agent into a standard tool wrapper introduces the architectural equivalent of a `GOTO` statement into your orchestrator.

When you call a collaborative agent, the control flow leaves the expected, structured context. The agent might hit an interrupted state, request more information, and potentially never return the expected output to the original caller if the user changes their mind or abandons the prompt.

You need a paradigm that isolates this messy, multi-turn state. You need a protocol that allows the domain agent to pause its execution, reach back out to the Orchestrator, negotiate a solution, and then resume its work without losing its conversational state.

That is exactly the gap the **A2A protocol** fills. By isolating this collaborative routing to the A2A layer, we keep the tool layer (MCP) clean, predictable, and strictly structured.

Building the Virtual Workforce

The introduction of the A2A protocol and the shift toward specialization does more than just solve technical bottlenecks—it creates the foundations for new marketplaces for expertise which represents a new model in how value is delivered.

Without A2A, a developer might build an application and struggle to maintain its growing complexity. In the A2A era, that same developer can focus on perfecting a high-value niche—such as an agent that specializes in "Real-time Regulatory Compliance". Whether built as a sophisticated multi-agent monolith or a distributed app, these systems can now be exposed to the world as A2A-compliant agents. This means that a specialist vibe coded in one part of the world can be discovered and "hired" by another Orchestrator across the globe.

The Agent Card

To facilitate this discovery, every agent is defined by an Agent Card. This is the standardized "CV" of the AI world, providing a machine-readable identity that outlines:

- **Capabilities:** What tasks the agent can perform.
- **Security & Compliance:** Its data handling policies and permission requirements.
- **Interaction Schemas:** How other agents should communicate with it via the A2A protocol.

Registries: From Public Marketplaces to Private Enterprises

Once an agent is equipped with its Agent Card, it can be registered within an Agent Registry whose goal is to turn Agents into discoverable services. This offers two primary paths for developers:

1. **Public Registries (Marketplaces):** Like a global talent agency, public registries allow developers to list their specialist agents for the world to find. This paves the way for building specialist agents for a specific industry and licensing its expertise to thousands of other orchestrators.

2. **Private Registries:** Within the enterprise, Agent Registries provide a secure, governed environment. Developers inside a large corporation can build a specialist agent that automates an internal workflow and register it for use by other departments, ensuring that expertise is shared across corporate boundaries without compromising security.

Ultimately, the A2A protocol transforms the act of building isolated agentic applications into building the **foundational members of a global, interoperable digital workforce**.

Implementing A2A Protocols

The practical implementation of the A2A architecture enables two distinct development motions:

- exposing AI agent as A2A agent (the supply side)
- orchestrating remote A2A agents (the demand side)

Exposing A2A Agent

Packaging an agent for the A2A ecosystem involves three core steps:

- **Defining the Agent Card:** Formal agent specification.
- **Implementing the Agent Executor (The Translation Layer):** It translates incoming A2A requests and responses into the specific calls required by the underlying agentic frameworks (e.g., ADK, LangGraph, or bespoke enterprise code).
- **Establishing the A2A Endpoint:** Executor must be exposed as an A2A-compliant endpoint.

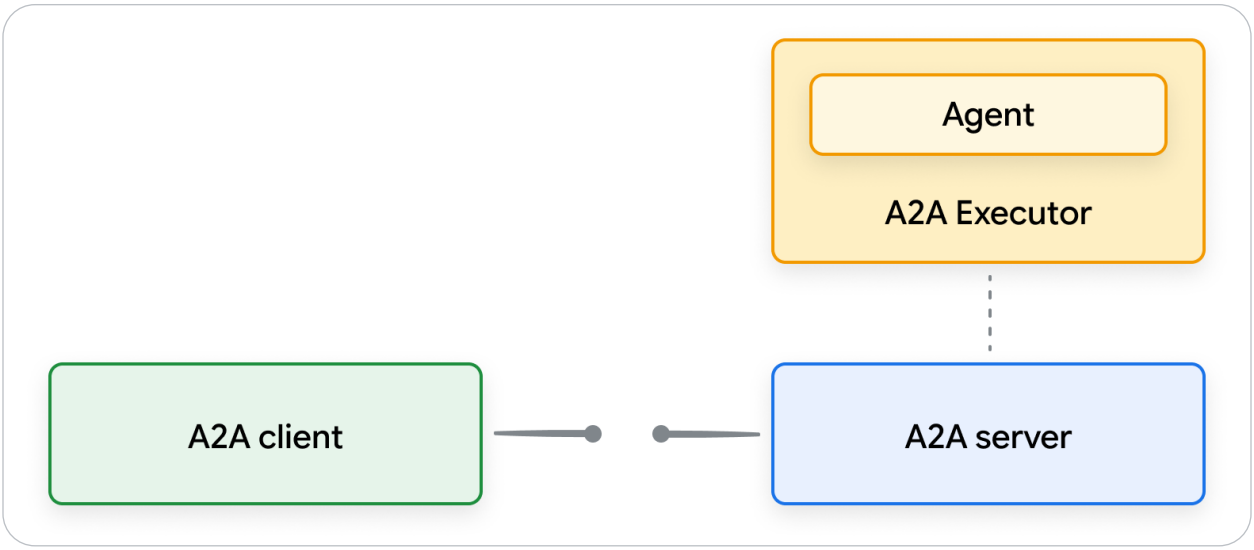


Figure 5: The supply-side exposure of a native AI agent as an A2A Server and its demand-side consumption via an A2A Client.

Connecting Remote A2A Agents

In a mature ecosystem, an application does not natively possess deep knowledge of every domain it touches. Instead, it acts as an **Orchestrator**—a central hub whose primary cognitive load is dedicated to understanding user intent, managing the overarching workflow, and delegating specific tasks to specialized, remote A2A agents.

Remote A2A agents operate as an autonomous, domain-bound contractor which communicates over the A2A protocol. Connecting to these remote agents generally follows one of two architectural patterns (code sample from Google ADK): direct point-to-point integration:

Python

```
# 1. Direct Instantiation via Hardcoded Endpoint
# Ideal for specific vendor integrations or private agents
billing_specialist = RemoteA2aAgent(
    name="billing_agent",
    endpoint="https://api.vendor.com/v1/billing/a2a"
)
```

Snippet 2: Direct remote A2A agent instantiation via a hardcoded endpoint.

or by using agent registry to discover agents:

Python

```
# 2. InDirect Instantiation via Agent Registry
registry = AgentRegistry(project_id=project_id, location=location)

#The registry resolves resource names and handles authentication validation
agent_name = f"projects/{project_id}/locations/{location}/agents/YOUR_AGENT_ID"
my_remote_agent = registry.get_remote_a2a_agent(agent_name=agent_name)
```

Snippet 3: Indirect remote A2A agent discovery and instantiation via an Agent Registry.

The Extensibility Layer: A2A as the Foundation for UI and Commerce

By resolving the fragmentation inherent in the early AI ecosystem, A2A protocol establishes a unified communication layer which in turn enables developers to build atop this standardized substrate. While the core A2A protocol functions as a transport and negotiation backbone, realizing rich, transactional applications very often demand highly specialized capabilities.

This requirement is addressed through the mechanism of A2A Extensions[1]. It provides a standardized pattern for agents to securely advertise, negotiate, and execute optional, higher-order functionalities that transcend basic message passing.

Three foundational frameworks that operate as native extensions atop the core A2A foundation include the **Agent-to-User Interface** (A2UI), designed to generate dynamic, stateful user experiences; the **Universal Commerce Protocol** (UCP), engineered to facilitate secure, autonomous agentic commerce; and the **Agent Payments Protocol** (AP2) which lays the groundwork for trusted, verifiable agentic payments. All three are discussed in more detail in the following sections.

Monetizing A2A Agents

Making an agent work is only half the battle; the other half is ensuring long-term commercial sustainability. Following the success of the Software-as-a-Service (SaaS) paradigm, the A2A protocol naturally enables an **Agent-as-a-Service (AaaS)** model. This allows specialized agents to be offered in a consumption-based model through various sales channels.

One example is utilizing the **Google Cloud Marketplace as a monetization engine**. Independent agent/software vendors and developers can list their A2A agents on the marketplace to instantly leverage the existing base of Google Cloud enterprise customers. Customers can procure these specialists and at the same time utilize their existing GCP financial commitments. Win-win for everyone.

The marketplace infrastructure automatically handles the "hard part"—complex billing—by providing native support for hybrid pricing, such as the **"Flat fee with usage"** model. This allows vendors to charge a predictable base fee while monetizing compute or token-based overages.

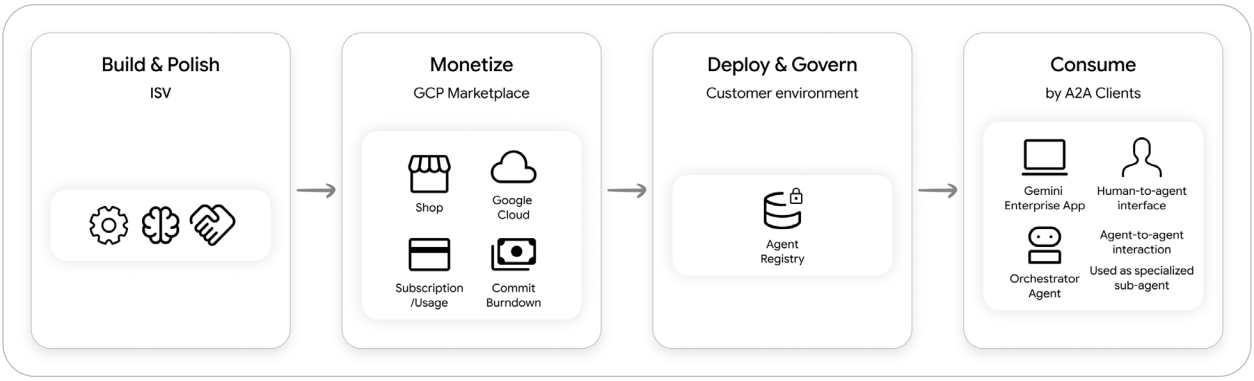


Figure 6: The Agent-as-a-Service (AaaS) Lifecycle.

Agents registered in the Google Cloud Marketplace are directly accessible to **Gemini Enterprise** app users. Gemini Enterprise is an advanced agentic platform that brings the best of Google AI to every employee for every workflow. It empowers teams to discover, create, share, and run AI agents within a single secure environment. By integrating with Agent Registries and operating as a native **A2A Client**, the platform augments the human experience by giving employees access to a broad ecosystem of specialized workers. At the same time Gemini Enterprise is a clear example of both an Agent as a Service platform (via its Assistant API), and it is also a host for remote agents.

While cloud marketplaces and traditional payment gateways will handle the majority of Agent-as-a-Service (AaaS) billing, the A2A protocol also supports permissionless, machine-to-machine microtransactions for developers who want to avoid managing user accounts entirely.

By utilizing the A2A Extensions framework `[A2AEXT]`, an agent's server can implement the **x402** (or L402) standard. In this pattern, the server intercepts a request and if “unpaid” returns an `HTTP 402 Payment Required` status code, bundled with a machine-readable

invoice. The calling agent executes the payment autonomously and retries the request with a cryptographic proof-of-payment token. This provides a standardized option for pay-per-call endpoints that require strictly stateless, automated billing.

The Extensibility Layer: A2A as the Foundation for UI and Commerce

By resolving the fragmentation inherent in the early AI ecosystem, A2A protocol establishes a unified communication layer which in turn enables developers to build atop this standardized substrate. While the core A2A protocol functions as a transport and negotiation backbone, realizing rich, transactional applications very often demand highly specialized capabilities. This requirement is addressed through the mechanism of A2A Extensions [A2AEXT]. It provides a standardized pattern for agents to securely advertise, negotiate, and execute optional, higher-order functionalities that transcend basic message passing.

Two foundational frameworks that operate as native extensions atop the core A2A foundation include the **Agent-to-User Interface (A2UI)**, designed to generate dynamic, stateful user experiences, and the **Universal Commerce Protocol (UCP)**, engineered to facilitate secure, autonomous agentic commerce. Both are discussed in more detail in the following sections.

Agent-to-UI (A2UI) Interoperability

The Communication Gap

When you ask a colleague, "How did Q4 perform by region?" they don't hand you a spreadsheet. They sketch a bar chart on the whiteboard, circle the standout regions, and add context. This is how humans share insights: through visuals and interaction.

Yet our agents return raw JSON. You then build the chart yourself. Import libraries, configure axes, manage state. This context-switching between asking for insights and doing frontend work breaks the vibe coding flow.

Agent-to-UI (A2UI) interoperability resolves this enabling Agents to generate complete, interactive user interfaces as outputs: not just JSON blobs.

Generative UI & A2UI

What is Generative UI?

Generative UI is the concept of LLMs dynamically creating user interfaces at runtime based on user intent and context. Instead of developers hard-coding every possible UI state, the model generates appropriate interfaces on demand. When you ask "Compare Q4 sales by region", the system doesn't just retrieve data: it composes an interactive layout, cards, filters, and controls, tailored to that specific request.

This represents a shift from static, pre-built interfaces to dynamic, context-aware UIs. The challenge is doing this safely. Running arbitrary UI code generated by an LLM can pose security risks: code injection, XSS attacks, and uncontrolled side effects.

A2UI: A Secure Implementation

A2UI is **a framework-agnostic standard for declaring UI intent** — Google's open-source way of letting agents describe interfaces in a portable, declarative format instead of streaming raw data or shipping arbitrary code.

To see why a *format* matters here, think about how a composer ships their work. They don't hand musicians a recording: they hand them sheet music. The same score plays on a piano, an orchestra, or a synthesizer; each instrument interprets the notation through its own voice. A2UI is sheet music for UI: the agent writes the intent (what to render and how it composes), and any renderer (e.g. React, Angular, Lit, Flutter, Jetpack Compose, SwiftUI, etc) performs it natively on the device the user actually has. The agent doesn't need to know whether you're targeting web, mobile, wearable, or appliance, it just knows the catalog of available components and any examples you want to give.

That separation of concerns is what makes A2UI safe. The agent doesn't generate executable code (a security nightmare) and it doesn't ship pre-rendered pixels (which can't reflow or stay interactive). Instead it requests components from a trusted catalog (e.g. buttons, text fields, cards, your own charts) and the client renders them using its own component library. The catalog defines what's available, the agent decides how to arrange them, the client assembles the final structure. Compositional, like LEGO blocks, but the blocks are UI components from *your* design system.

The Basic Catalog (and Bringing Your Own)

A2UI ships a basic catalog of 18 ready-to-use components for prototyping and demos:

Category	Components
Layout	Row, Column, List
Display	Text, Image, Icon, Divider
Containers	Card, Modal, Tabs
Media	Video, AudioPlayer
Interactive	Button, TextField, CheckBox, Slider, DateTimeInput, ChoicePicker

Table 1: A2UI Basic Catalog

This set was called "standard" in v0.8 and renamed to "basic" in v0.9, a deliberate signal that production frontends should bring their own catalog, mapping their existing components (your design-system buttons, your charts, your maps) to A2UI types. The agent doesn't change; only the renderer's catalog mapping does (`ChoicePicker` was `MultipleChoice` in v0.8.)

Here's what an A2UI message looks like (v0.9 format):

JSON

```
{
  "version": "v0.9",
  "updateComponents": {
    "surfaceId": "main",
    "components": [
      {"id": "root", "component": "Column", "children": ["title",
"summary", "export"]},
      {"id": "title", "component": "Text", "text": "Q4 Sales", "variant": "h1"},
      {"id": "summary", "component": "Text", "text": "Revenue grew 12% QoQ"},
      {"id": "export", "component": "Button", "child": "export-label",
       "action": {"event": {"name": "export_csv"}}},
      {"id": "export-label", "component": "Text", "text": "Export CSV"}
    ]
  }
}
```

Snippet 4: Example A2UI message

Components form a flat adjacency list referenced by id, which makes the structure easy for an LLM to generate incrementally and easy for the client to update without re-rendering. A separate `createSurface` message (not shown) tells the client which id is the root. The client then renders this as a complete, interactive interface: no React code required.

Generating A2UI: Two Patterns

There are two ways to produce A2UI in practice, and the choice is mostly about where the layout decision lives.

The **default** is to let the LLM emit A2UI directly: the model owns the layout, adapts it to user intent, and the same agent handles "compare these regions" and "show me trends" with different interfaces. Production code for this pattern uses the official `a2ui-agent-sdk` and lives in 4.4.

The **specialization** is to have a tool return a fixed A2UI structure: one tool call, no LLM tokens spent on UI generation, fully predictable output. This is the right call when the layout is deterministic from inputs: every region's sales dashboard looks the same, every booking form has the same fields. Effectively, the tool is a server-side template.

A clean tool-as-template tool does two things in its body: build the A2UI structure with **data bindings** (path references, not f-string interpolation) and return it. The framework's `A2uiPartConverter` (from `a2ui-agent-sdk`) intercepts the tool's response and routes it to the client as an A2UI part, so the tool itself stays a plain Python function:

Python

```python
from google.adk.agents import LlmAgent
from google.adk.models import Gemini

def get_sales_dashboard(region: str) -> dict:
    """Build a data-bound sales dashboard for `region`."""
    data = fetch_sales(region)
    return {
        "version": "v0.9",
        "updateComponents": {
            "surfaceId": "sales",
            "components": [
                {"id": "root", "component": "Column",
                 "children": ["title", "total", "drill"]},
                {"id": "title", "component": "Text",
                 "text": {"path": "/title"}, "variant": "h1"},
```

Continues next page...

```
                {"id": "total", "component": "Text", "text", {"path": "/total"}},
                {"id": "drill", "component": "Button", "child": "drill-label",
                 "action": {"event": {"name": "expand_details"}}},
                {"id": "drill-label", "component": "Text", "text": "Drill Down"},
            ],
        },
    }

agent = LlmAgent(
    name="sales_agent",
    model=Gemini(model="gemini-flash-latest"),
    tools=[get_sales_dashboard],
)
# Wire the converter at executor setup so this tool's response becomes an
A2UI part:
#   from a2ui.adk.send_a2ui_to_client_toolset import A2uiPartConverter
#   A2aAgentExecutorConfig(event_converter=A2uiPartConverter(catalog,
bypass_tool_check=True))
```

Snippet 5: LLM-generates-UI pattern

Data values flow through a parallel `updateDataModel` message that resolves the `{path: "/title"}` references, so clients can re-render on data updates without re-sending the structure. The LLM only sees a structured tool response (not the rendered UI) so its context stays focused on what to do next, not on the UI it just built.

When to Use Each Pattern

Query	Return
"What's the average?"	Data (text)
"Compare these regions"	UI generated by the LLM (§4.4)
"Show me my dashboard"	UI built by a tool (this section)
API-to-API	Data (JSON)

Table 2: Query to Output Mapping

Use A2UI when interaction or visualization adds value beyond the raw data, and pick the pattern by who owns the layout decision: the LLM (intent-driven) or a deterministic template (input-driven).

Interactive Artifacts & The Canvas

Traditional chat interfaces are linear. Each response is static. The canvas approach is different: it creates a persistent workspace where both the agent and user can edit.

Instead of scrolling through chat history, you have a living document. The agent can modify sections. You can edit manually. Both changes are reflected in real-time.

Canvas + A2UI

Combining canvas persistence with A2UI interactivity creates useful workflows:

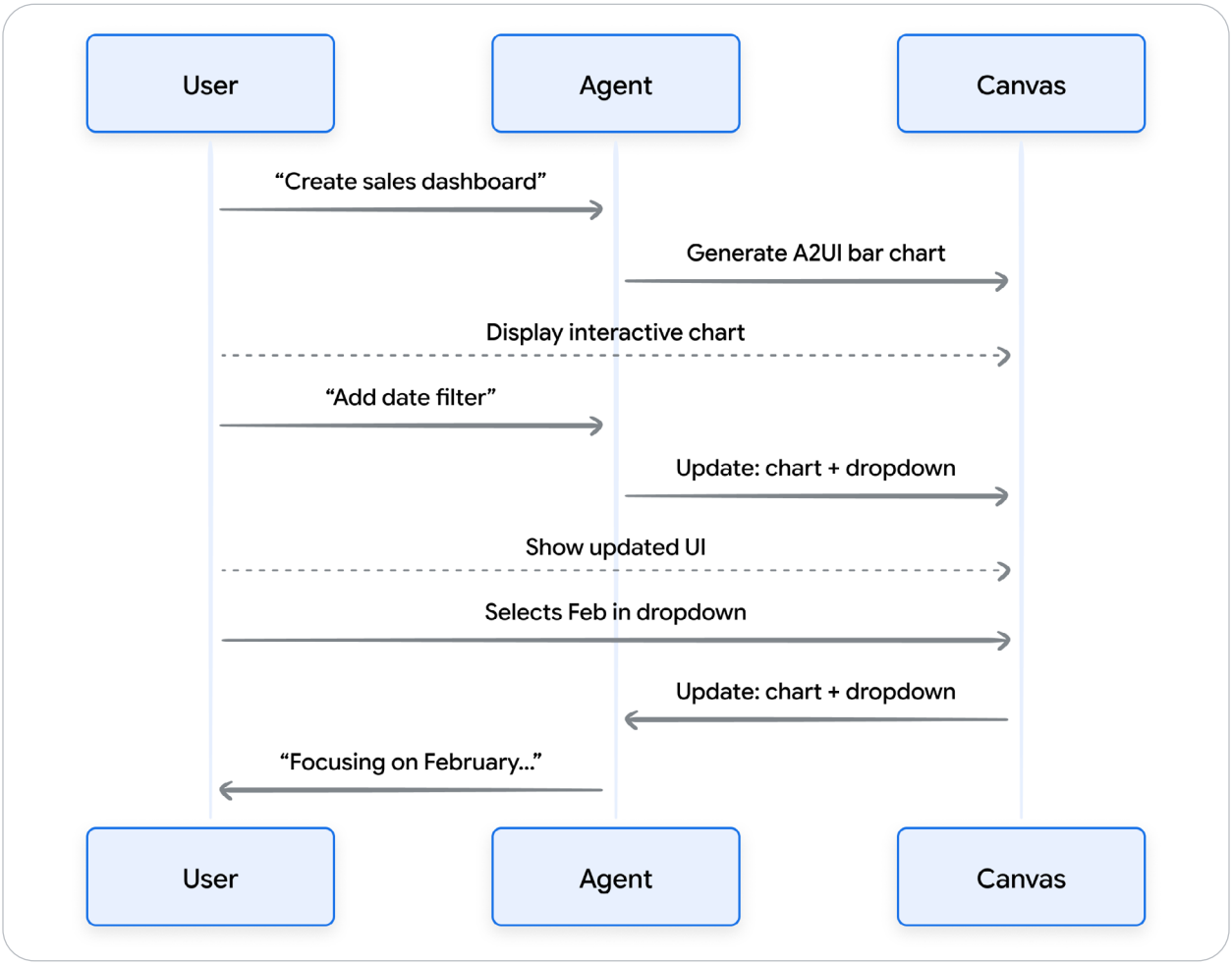


Figure 7: User, Agent, Canvas interaction flow

The UI isn't just rendered: it's a communication medium. The agent observes your interactions and responds accordingly.

Best Practices

Let LLMs Generate A2UI

For the LLM-generates-UI pattern (the default introduced in 4.2), hand-coding A2UI JSON is tedious. Use the official `a2ui-agent-sdk` (`pip install a2ui-agent-sdk`): the `A2uiSchemaManager` builds a system prompt that already embeds the catalog schema and worked examples, the catalog ships its own JSON-Schema validator, and the same SDK provides a parser for the `<a2ui-json>` blocks the agent emits. Then validate and retry on schema errors.

Python

```
# pip install a2ui-agent-sdk google-adk
import asyncio, json
import jsonschema
from a2ui.schema.manager import A2uiSchemaManager
from a2ui.basic_catalog.provider import BasicCatalog
from a2ui.schema.constants import VERSION_0_9
from a2ui.parser.parser import parse_response
from google.adk.agents import LlmAgent
from google.adk.models import Gemini
from google.adk.runners import InMemoryRunner
from google.genai import types

# 1. The SDK loads the catalog, builds the validator, and renders a complete
#    system prompt with the schema and worked examples baked in.
schema_manager = A2uiSchemaManager(
    version=VERSION_0_9,
    catalogs=[BasicCatalog.get_config(version=VERSION_0_9)],
)
catalog = schema_manager.get_selected_catalog()
```

Continues next page...

```
agent = LlmAgent(
    model=Gemini(model="gemini-flash-latest"),
    name="ui_agent",
    instruction=schema_manager.generate_system_prompt(
        role_description="You generate interactive UIs as A2UI v0.9 messages.",
        ui_description="Use Cards, Lists, ChoicePickers, and Buttons to
present data.",
        include_schema=True,
        include_examples=True,
    ),
)

# 2. Run the agent. It emits text containing one or
more <a2ui-json>...</a2ui-json>
#    blocks; parse_response() extracts the parsed JSON. Validate, then retry
on errors.
async def create_ui(intent: str, data: dict, max_retries: int = 3) -> list:
    runner = InMemoryRunner(agent=agent, app_name="ui_demo")
    session = await runner.session_service.create_session(
        app_name="ui_demo", user_id="u",
        state={"expression": "{expression}"},  # escapes the SDK's
templating placeholder
    )
    query = f"{intent}\n\nData: {json.dumps(data)}"
    last_error = None
    for _ in range(max_retries + 1):
        chunks = []
        msg = types.Content(role="user", parts=[types.Part(text=query)])
        async for ev in runner.run_async(user_id="u",
session_id=session.id, new_message=msg):
            if ev.content and ev.content.parts:
                chunks.extend(p.text for p in ev.content.parts if p.text)
        blocks = [rp.a2ui_json for rp in parse_response("".join(chunks))
if rp.a2ui_json]
        try:
            for b in blocks:
                for m in (b if isinstance(b, list) else [b]):
                    catalog.validator.validate(m)
            return blocks
        except jsonschema.ValidationError as e:
```

Continues next page...

```
            last_error = e
            path = "/".join(str(p) for p in e.absolute_path)
            query = (
                f"Your previous response failed schema validation at {path}: "
                f"{e.message[:200]}\nFix this and retry:
{intent}\nData: {json.dumps(data)}"
            )
    raise ValueError(f"Schema validation failed after {max_retries}
retries: {last_error}")
```

Snippet 6: A2uiSchemaManager implementation

In production, wrap `create_ui()` in try/except and fall back to a text response on schema-validation failure. LLM output is stochastic, and the renderer should never see a malformed payload.

Hybrid Output for Flexibility

Provide both data and UI so consumers can choose:

JSON

```
{
  "data": {"sales": [...]},
  "ui": {"version": "v0.9", "updateComponents": {"surfaceId": "main",
"components": [...]}},
  "ui_available": true
}
```

Snippet 7: Hybrid Output Example Schema

API clients ignore the `ui` field and use `data`. Human-facing clients render the A2UI message.

Summary

Generative UI lets LLMs create user interfaces at runtime based on user intent. A2UI is Google's open-source standard for doing this safely: a framework-agnostic format for declaring UI intent, so the same agent message renders natively in Lit, Flutter, React, or your own design system.

The security model matters: agents can't inject arbitrary code. They can only request components the renderer's catalog already trusts. That's what makes A2UI safe while still enabling dynamic, generative interfaces.

Agents and Commerce (AP2 and UCP)

Autonomous Procurement with a workflow example

Vibe coders learn checkout, catalog , order capability and mandate with their agents.

Key characteristics and benefits of Protocols

Typed schemas, Security and open source (Integration debt and vendor neutrality)

Lab recommendations:
https://codelabs.developers.google.com/next26/adk-agent-commerce#0

While earlier sections focused on how coding agents interact with tools, other agents, and user interfaces within an Agentic framework, those interactions are often limited to "read" operations using protocols like MCP, A2A, and A2UI. As these systems evolve, agents must transition toward executing "actions" that carry real-world financial implications.

Prioritizing commerce protocols and building a robust operational harness ensures your agents adhere strictly to industry standards when managing transactions.

To understand how to implement these systems, we must first explore the nature of these protocols and their complementary roles.

What is AP2 and UCP?

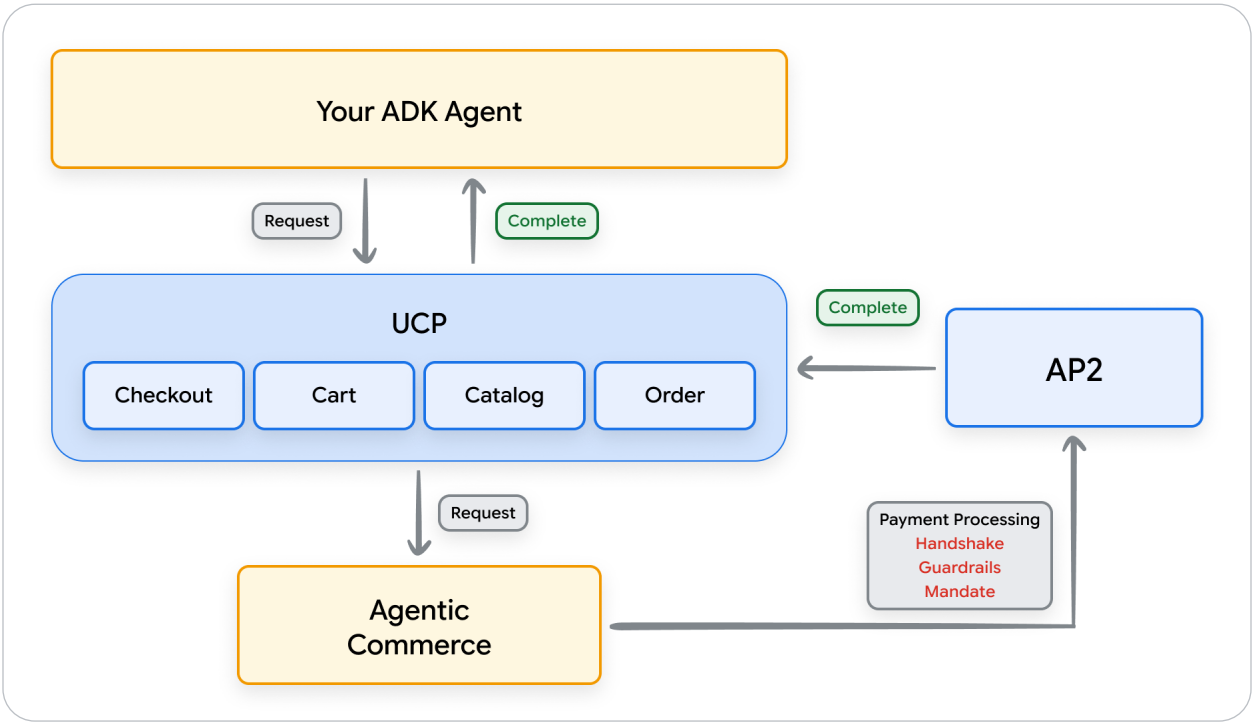


Figure 8: AP2 and UCP Ecosystem

Imagine you and your roommates are starving at 2:00 AM. You are too busy studying to stop, so you deploy your AI Smart Assistant to get food for the apartment.Here is how UCP (Universal Commerce Protocol) and AP2 (Agent Payments Protocol) split the workload to get that burrito to your dorm lobby.

UCP (Universal Commerce Protocol): The Ultimate Food Delivery App

In 2024, your AI would have to manually open Chrome, go to the local burrito place's poorly designed website, try to click the "extra guac" checkbox, and hope the site doesn't crash. Instead, UCP acts like a universal translator. Every restaurant on campus publishes their menu, hours, and customization options in a clean, standard machine language.

Your AI uses UCP to ask the restaurant’s system, "Are you still open? Do you have vegetarian burritos?". To build an order one could ask: "One chicken burrito, no onions, add sour cream, and a side of chips" and the restaurant responds back with the tax, delivery fee, and expected arrival time.

In short: UCP is how your AI talks to the store, looks at the options, and builds the perfect order. Prior to UCP, your application would have been able to discover and get information. However the interactions between these different providers need to be orchestrated individually. By standardizing the interaction, we make it easier for non-humans to interact with each other.

AP2 (Agent Payments Protocol): *The Parent's Credit Card with Strict Rules*

Now the food is in the cart, but your AI needs to pay. You obviously aren't going to type your actual debit card number into an AI prompt and say "Go wild."

AP2 is the **open, shared protocol** that provides a common language for secure, compliant transactions between agents and merchants that lets your AI pay for the food only within rules you set.

- **The Guardrails (The Mandate):** Before you send the AI off, you approve a digital rule: *"You can spend up to $25 at Taco Bell."*
- **The Handshake:** When the AI checks out, it doesn't show your card number. It shows a digital, encrypted "promissory note" signed by you that says: *"My human approved this $18.50 order."* The restaurant's bank instantly verifies this digital signature.
- **No Hidden Fees:** If the restaurant tries to sneakily charge you $50 instead of $18.50, the AP2 protocol blocks it instantly because it violates the rules you signed off on.

In short: AP2 is the **secure lockbox** that lets your AI pay for things using your money, but ensures it can never buy a $1,000 TV by mistake.

UCP	AP2
Brain that decides what to buy, handles the menu, and puts the food in the cart.	Is the wallet that securely handles how to pay for it without you getting scammed.
Integrates with any business provider	Integrates with payment
Unified integration: Shared language: Extensible architecture Security-first approach	Authorization & Auditability Authenticity of Intent Agent Error and Hallucination Accountability

Table 3: Comparison for UCP and AP2

- If you are a developer building your UCP agent for a merchant or the payment processor, follow the instructions here.
- If you are a developer building your AP2 agent for a merchant or the payment processor, follow the instructions here.
- If you are a developer building an agent that consumes an AP2 agent such as a shopper, follow the instructions here.

Conclusion

By adopting foundational standards like MCP, A2A, A2UI, AP2 and UCP, organizations can eliminate the crushing technical debt of bespoke integrations and focus entirely on orchestrating high-value business logic. This paradigm shift fundamentally elevates developers from mere mechanics wiring fragile APIs into true architects of a global, autonomous workforce. As these standardized communication layers mature, they will unlock entirely new economies of scale, transforming how enterprise software is built, consumed, and monetized.

Endnotes

1. A2A Community (2026) A2A Documentation: Extensions, *A2A Protocol*, https://a2a-protocol.org/latest/topics/extensions/
2. Antigravity Team (2026) Antigravity Editor: MCP Integration, *Antigravity Docs*, https://antigravity.google/docs/mcp
3. Fowler M, Lewis J (2014) Microservices, *MartinFowler*, https://martinfowler.com/articles/microservices.html
4. Google A2A Team (2025) Announcing the Agent2Agent Protocol (A2A), *Google Developers Blog*, https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
5. Google A2UI Team (2025) Introducing A2UI: An Open Project for Agent-Driven Interfaces, Google Developers Blog, https://developers.googleblog.com/introducing-a2ui-an-open-project-for-agent-driven-interfaces/
6. Google A2UI Team (2026) A2UI v0.9: The New Standard for Portable, Framework-Agnostic Generative UI, *Google Developers Blog*, https://developers.googleblog.com/a2ui-v0-9-generative-ui/
7. Google A2UI Team (2026) A2UI [Computer Software], GitHub, https://github.com/google/A2UI
8. Google Cloud Databases Team (2025) MCP Toolbox for Databases: Simplify AI Agent Access to Enterprise Data, *Google Cloud Blog*, https://cloud.google.com/blog/products/ai-machine-learning/mcp-toolbox-for-databases-now-supports-model-context-protocol
9. Google Cloud Team (2026) Announcing Agents-to-Payments (AP2) Protocol, *Google Cloud Blog*, https://cloud.google.com/blog/products/ai-machine-learning/announcing-agents-to-payments-ap2-protocol?e=48754805
10. Google Cloud Team (2026) Configure MCP in an AI Application, *Cloud Docs*, https://docs.cloud.google.com/mcp/configure-mcp-ai-application#antigravity
11. Google Stitch Team (2026) Google Stitch Documentation: MCP Setup, *Stitch with Google Docs*, https://stitch.withgoogle.com/docs/mcp/setup
12. MCP Team (2026) Specification for Model Context Protocol, *MCP Docs*, https://modelcontextprotocol.io/specification/2025-11-25

13. Muchandi V (2026) Rad-skills: Gemini Skills for Rapid Agent Development, *GitHub*, https://github.com/VeerMuchandi/rad-skills

14. Hotz H, (2026) The Agentic Commerce Revolution, *O'Reilly Radar*, https://www.oreilly.com/radar/the-agentic-commerce-revolution/

15. Pichai S (2017) Making AI Work for Everyone, *Google Blog*, https://blog.google/innovation-and-ai/products/making-ai-work-for-everyone/

16. Saboo S, Overholt K (2026) Developer's Guide to AI Agent Protocols, *Google Developers Blog*, https://developers.googleblog.com/developers-guide-to-ai-agent-protocols/

17. Styer M, Patlolla K, Mohan M, Diaz S (2025) Agent Tools & Interoperability with Model Context Protocol (MCP), *Kaggle*, https://www.kaggle.com/whitepaper-agent-tools-and-interoperability-with-mcp

18. The Linux Foundation (2026) A2A Protocol Surpasses 150 Organizations, Lands in Major Cloud Platforms, and Sees Enterprise Production Use in First Year, *Linux Foundation Press*, https://www.linuxfoundation.org/press/a2a-protocol-surpasses-150-organizations-lands-in-major-cloud-platforms-and-sees-enterprise-production-use-in-first-year

19. Common AI Catalog and Registry Standard Documentation (2026), https://github.com/Agent-Card/ai-catalog

20. Ricardo Cataldi (2026), MCP vs A2A: Tools, Agents, and Where Each Protocol Belongs, https://levelup.gitconnected.com/mcp-vs-a2a-tools-agents-and-where-each-protocol-belongs-53e1f9ab9765

21. Lukas Geiger (2026), Monetizing AI Agents: Ensuring Profitability https://medium.com/google-cloud/monetizing-ai-agents-ensuring-profitability-92efa535ebb3