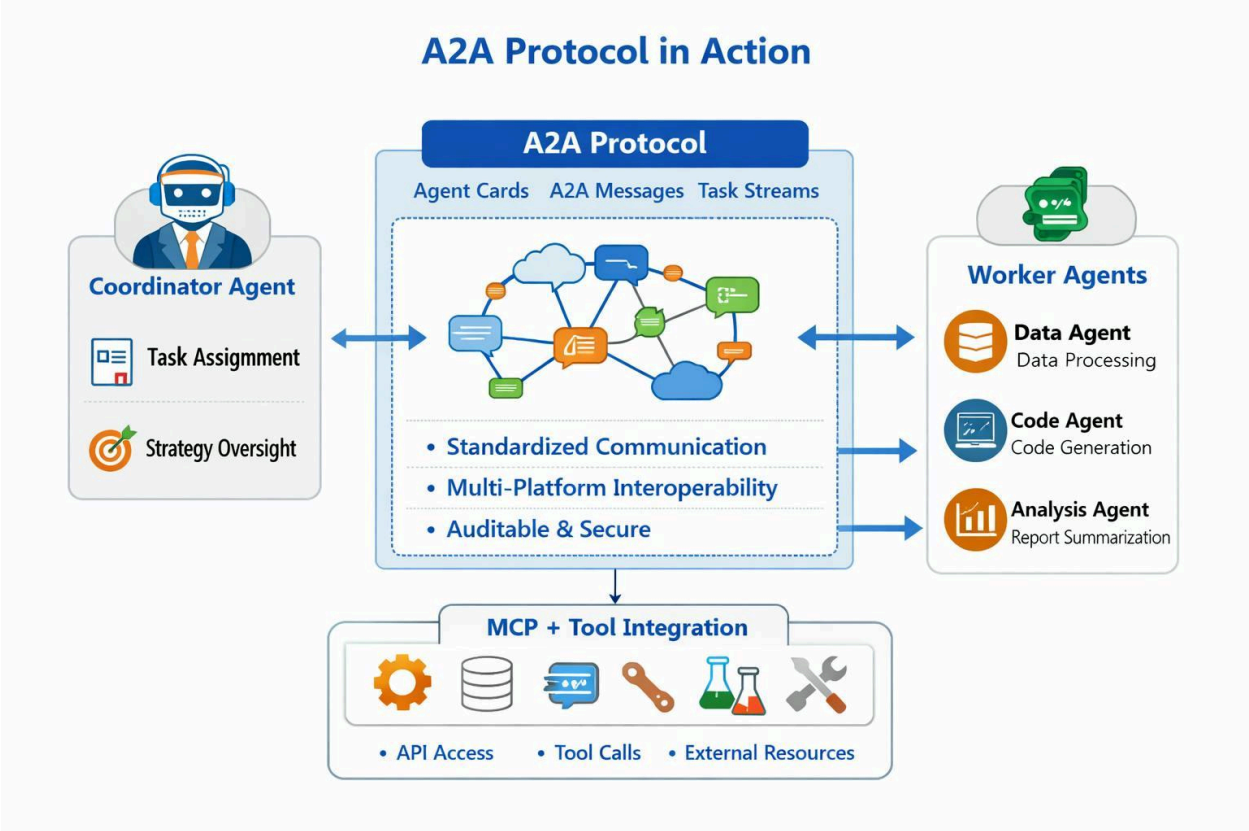
Agent-to-Agent (A2A) Protocol: The Future of Autonomous Multi-Agent Systems

The Agent-to-Agent (A2A) protocol is an open, vendor-neutral standard proposed by Google to enable seamless collaboration between independent AI agents. A2A lets agents *discover* each other via **Agent Cards** (standard JSON metadata), exchange **structured Tasks** and **messages** over JSON-RPC/HTTP, and stream updates or push notifications for long-running work. In essence, A2A establishes a “universal language” for agents so they can securely share information and coordinate workflows across different organizations and platforms.



A2A protocol workflow overview

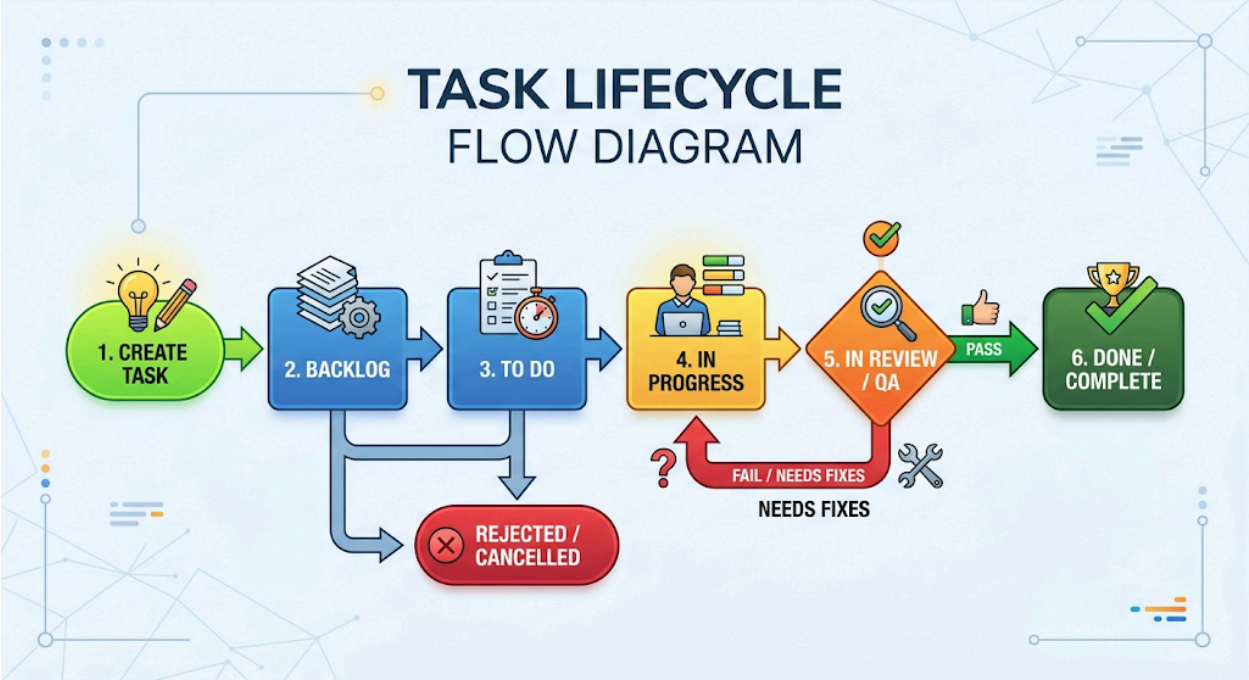
Unlike Anthropic’s Claude Code/MCP approach – where a single agent can call tools (MCP) or spawn subagents within one environment – A2A is designed for *agent-to-agent* interoperability. It focuses on **horizontal integration** between peer agents, whereas MCP focuses on **vertical integration** of an agent with external data

sources and tools. In practice, modern autonomous systems will use **both**: MCP (Model Context Protocol) connects agents to databases, APIs or on-prem systems, while A2A coordinates *multiple* agents (each possibly using MCP) to complete complex workflows. When combined, A2A+MCP form a future-proof, interoperable ecosystem that supports scalable, enterprise-grade autonomous processes.
Below we explain the technical building blocks of A2A (Agent Cards, Tasks, streams, etc.), compare it feature-by-feature with Claude Code + MCP, illustrate several real-world multi-agent workflows (with diagrams), and discuss limitations, security, governance, and architectural recommendations for adoption.

How A2A Works: Agents, Agent Cards, and Tasks

Agents and Agent Cards
In A2A, each “agent” is a running service (or LLM-powered assistant) that exposes *capabilities* and endpoints to others. Agents announce themselves by hosting a standardized JSON document called an **Agent Card** (typically at `/.well-known/agent.json` on their service URL). An Agent Card includes metadata like the agent’s name, description, endpoints, supported input/output modalities, and authentication methods (e.g. OAuth2 schemes). For example, an Agent Card might list skills like “FlightSearch” and “HotelBooking” and say it accepts text or image inputs. Other agents (or an agent registry) can discover agents by URL or DNS, and read their Agent Cards to know **who** they are, **what** tasks they can perform, and **how** to authenticate with them. Agents publish their metadata and capabilities via a standardized JSON document called an Agent Card… the discovery can be handled through DNS, registries, marketplaces, or private catalogs.
Tasks and Messages
The **Task** is the core unit of work in A2A. A client agent sends a new task to a remote agent via a `task/send` request. Each task has a unique `taskId` and may include a **Message** (the user’s request or task description) and optionally any **Parts** of data (like text, files, or JSON blobs) needed to perform it. A task’s JSON object is stateful and has a lifecycle. Upon receiving a task, the remote agent processes it and can respond in various ways: immediate **results** can be returned as an **Artifact** (e.g. a report PDF), intermediate **messages** (requests for clarification), or it can set the task’s state to `input-required` if it needs more input.
For example, a “ScheduleMeeting” task might have an initial message with proposed times. The agent would work on it, possibly ask follow-up questions, and finally produce artifacts like a confirmed meeting invitation. At any time the agent can update the client by emitting messages or artifacts. Importantly, tasks support **streaming** and **push updates**: if the task is long-running, the agent can stream partial results over Server-Sent Events, and even send a push notification (webhook) when done.

Task Lifecycle



A2A tasks progress through well-defined states. When created, a task is `submitted`, then moves to `working` while being processed. If the agent needs more data, it transitions to `input-required` (prompting the client to send more). Finally, it reaches `completed` (or an error state). At each state change, the agent can send **TaskStatusUpdate** events. A compliant implementation must deliver events in order, support multiple concurrent streams for a task, and allow clients to poll or subscribe for updates. In practice, this means a client can periodically poll `task/get`, or open an SSE stream to receive live updates as the agent works.

Summary of Key Objects. In A2A the main protocol objects are:

- **Agent Card:** JSON metadata published by each agent describing its identity, endpoints, supported inputs/outputs, authentication, and capabilities (skills/tags).
- **Task:** A structured JSON object representing an atomic unit of work with a unique ID and lifecycle states (submitted, working, input-required, completed).
- **Message:** Part of a Task containing the conversational payload (e.g. user prompt or agent clarification).
- **Artifact:** An immutable result produced by the agent (e.g. a file or data blob).
- **Part:** A data chunk (text, JSON, image, etc.) embedded in a Message or Artifact.

Together, these allow agents to talk in **structured natural language or multi-modal messages** without sharing their internal memory or tools.

Technical Details: Protocol and Security

A2A is designed on open web standards: it typically uses **JSON-RPC over HTTP** for sending tasks and retrieving status. For streaming, it uses server-sent events (SSE) to push incremental updates, and standard HTTP webhooks for push notifications. It uses OAuth2/OIDC for authorization: agents authenticate peers and verify that the sending agent is allowed to submit tasks. Agent Cards can be signed with JWT to prevent tampering. By default, A2A expects **HTTPS** transport and follows enterprise-grade security practices (OAuth2 flows, PKCE, scopes).
This makes A2A secure-by-design: agents do *not* share passwords or API keys. Instead, a client agent obtains an OAuth token and includes it when sending a task. The remote agent verifies the token against its policy (often checking client ID, scopes, and an optional audience field in the Agent Card). In short, *no agent can talk to another without authorization*, and all data in transit is encrypted (HTTPS).
In practice, each enterprise implementing A2A will define a governance framework: which agents are allowed to call which others, what scopes they need, and what data they can exchange. The A2A spec encourages clear audit logging and role separation. For example, an enterprise might require that only a “Client Agent” with a certain OAuth scope can create tasks for a “Payment Agent”. By design, the protocol is **multi-tenant and cross-domain**: an agent owned by Company A can delegate to an agent owned by Company B as long as both trust each other’s PKI/signatures. This cross-organization capability is a key difference from single-tenant solutions.

Comparison: A2A Protocol vs Claude Code + MCP

To illustrate the fundamental architectural divergence, consider a routine travel scenario: the simultaneous booking of a flight and hotel accommodation.

- **A2A Protocol Approach:** You interact with distinct, specialized service providers as peers. You would first *discover* and dispatch a **Task** to a “Flight Agent” (operated by an airline), then repeat the process with a “Hotel Agent” (operated by a booking platform). These agents function as **independent black boxes**; they are unaware of each other's internal state and communicate solely through standardized A2A messages.

- **Claude Code + MCP Approach:** Coordination is *centralized*. A primary Claude instance orchestrates the entire workflow within a single conversational context, potentially invoking sub-functions or tools as defined by the Model Context Protocol. Unlike A2A peers, these “subagents” or tools lack autonomous, discoverable identities and are instead treated as **attached resources** under the

master agent's control.

This comparison underscores the core distinction: A2A enables **horizontal integration** through peer-to-peer protocols between discoverable agents, whereas Claude Code + MCP focuses on **vertical integration** between a single orchestrator and its subordinate toolchain.

Below, we examine these technical differentiators in greater detail:

Feature	A2A Protocol	Claude Code + MCP
Discovery	Agents are discovered via **Agent Cards** (hosted JSON files), DNS/registry lookups, or catalogs.	Subagents in Claude Code are pre-configured in the conversation or project; there is no standard external discovery mechanism.
Communication	**Direct agent-to-agent:** JSON-RPC tasks/messages exchanged between distinct services. No shared memory or tools.	**Main-agent mediated:** A single main Claude agent runs subagents internally. Subagents do not talk peer-to-peer; communication is via the main agent's chat interface.
Auth & Security	Standard OAuth2/JWT. Agents authenticate each other per request, using scopes (e.g. via an identity provider).	Anthropic handles auth under-the-hood. Subagents inherit the main agent's user auth; no open OAuth handshake between agents. Tools (like databases) have separate creds.
Cross-Org Support	Yes. A2A is designed for multi-tenant, cross-company collaboration. Agents can be published publicly or privately.	No. Claude Code subagents run in one user's workspace or org; they cannot directly integrate with agents owned by another company.
Tool Access	**None.** Agents do not directly call APIs or tools within the protocol. Task handling is agent-to-agent only.	**Built-in.** Claude Code subagents can call any integrated tool or MCP server (e.g., file system, APIs). They inherit all tools from the main conversation.
Task Lifecycle	Fully supported. Tasks have states (submitted, working, etc.) and	Limited. Claude Code tasks are just conversation threads. There's no

Feature	A2A Protocol	Claude Code + MCP
	updates (polling, streaming, webhooks).	standardized task object with lifecycle; subagents finish and return answers in one shot.
Streaming/Updates	Built-in. A2A natively supports SSE streaming and webhooks for real-time updates.	Not native. Claude Code conversations can stream chat responses but there's no separate SSE/push system for agent-to-agent updates.
Auditability	High. Tasks and agent calls can be logged by both parties, with unique IDs for traceability. Task histories persist in logs.	Basic. The chat history shows when subagents were invoked, but there is no formal audit log or task ID for cross-agent workflows.

A concise way to view it: **A2A** creates a formal, HTTP-based interface between black-box agents, while **Claude Code + MCP** is an internal toolchain where one "master" agent orchestrates subordinate agents and tools under one user's session. A2A focuses on *peer-to-peer agent protocols*, whereas Claude Code/MCP focus on *agent-to-tool integrations*.

- *Discovery & Extensibility:* A2A's use of Agent Cards and registries allows plugging in new agents across organizations without code changes. Claude Code's subagents, by contrast, must be defined ahead of time (via YAML or CLI) and cannot be dynamically discovered.
- *Security & Governance:* A2A's OAuth2/PKCE-based approach means each interaction is explicitly authorized. Claude Code subagents rely on the main user's login and workspace permissions; a subagent can access any tool granted to the main agent, including injected MCP servers. There is no standardized "scope" hand-off between agents.
- *Agent Interaction:* In A2A, **agents act as peers**, each deciding how to handle a task. In Claude Code, subagents do **one-shot tasks** (typically answering a question or performing a simple function) and then disappear. They don't talk to each other – all coordination is done by the main agent. This means multi-agent workflows are *decentralized* in A2A but *centralized* in Claude Code.
- *Tool vs Agent:* If a step in the workflow is calling a known API or database, an A2A system would typically do that by invoking an agent that is an MCP "tool provider" (like a Google Drive agent). In Claude Code, the agent would call MCP servers directly. In short, A2A treats every component (tools or people) as an "agent"; MCP treats tools as attached resources.

Real-World Multi-Agent Workflows

Below are three illustrative examples of end-to-end workflows achieved with A2A (and, where relevant, MCP). They show how humans can step back while agents autonomously collaborate.

1. Autonomous IT Helpdesk Ticket (A2A-Only)

Scenario: A user reports a laptop issue (e.g. “Laptop won’t turn on after update”). An AI-driven helpdesk orchestrator (Client Agent) breaks this into subtasks handled by specialized agents:

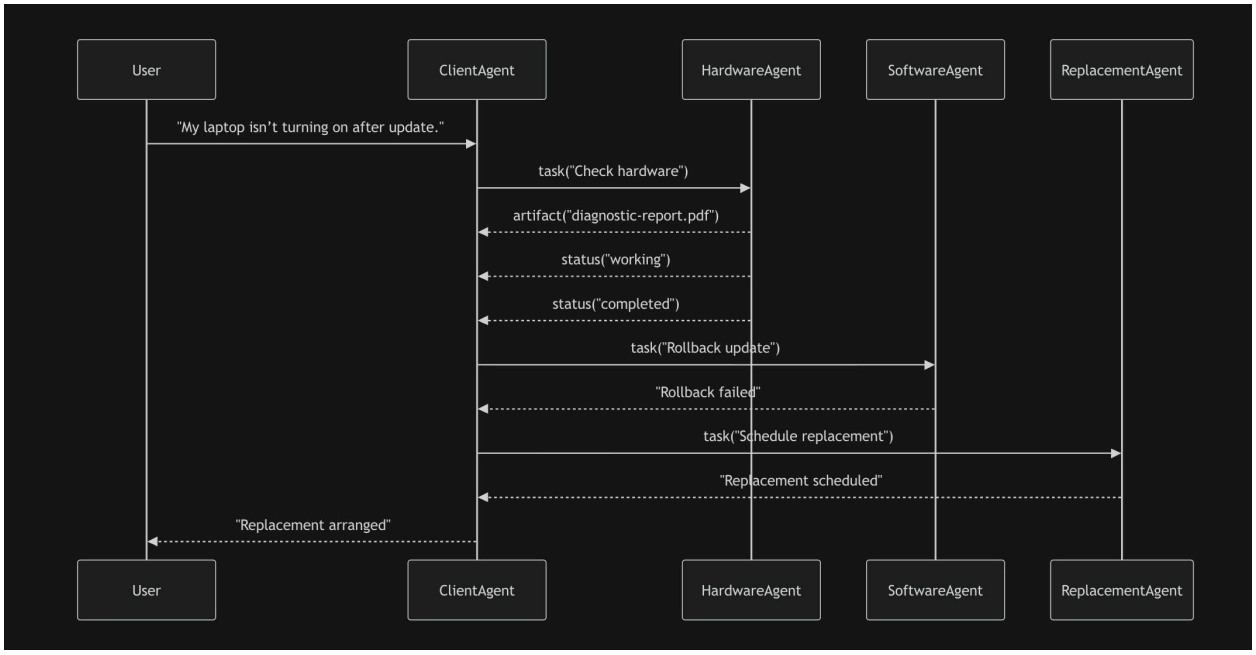
Step 1: *Hardware Diagnostic Agent* – The client agent creates a task `CheckHardware` and sends it via A2A to the hardware agent. The hardware agent runs diagnostics (streaming logs or data back as parts/artifacts).

Step 2: *Software Rollback Agent* – If hardware is OK, the client agent then tasks a rollback agent to undo the recent update. That agent replies with a success/failure.

Step 3: *Replacement Agent* – If rollback fails, the client agent tasks a device-replacement agent to schedule hardware replacement. The replacement agent may contact inventory systems (potentially via its own sub-flow) and complete the service ticket.

Step 4: *Completion:* Once a repair plan is set, the client agent updates the user (human) or ticketing system that the issue is resolved.

Each agent publishes an Agent Card with its capabilities (e.g. “diagnose_laptop”, “rollback_update”) so the client agent can discover and invoke it. All communication is via A2A tasks and messages; for example, the hardware agent might stream diagnostic results as a running log artifact. At no point does the client share internal code or credentials – just well-defined tasks. This is a **pure A2A use case**: *“No structured tools like APIs or OCR engines are involved. All coordination happens between opaque, collaborating agents”*. The agents act as true peers with decision logic: e.g. the hardware agent independently decides if the device is OK, and the software agent decides whether rollback is feasible.



(Above sequence: Each arrow is an A2A call/task or response. The client agent decides flow; agents reply or stream artifacts.)

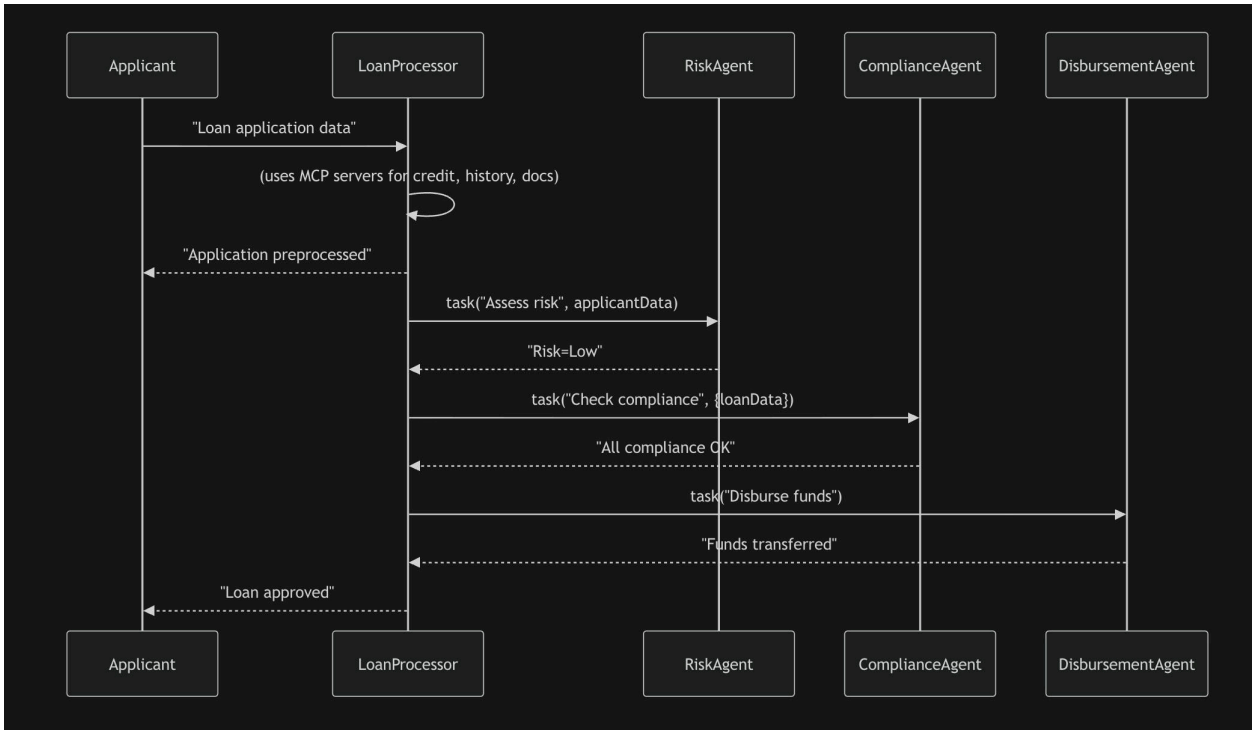
2. Loan Approval Workflow (A2A + MCP)

Scenario: A bank automates loan approvals with agents. This example uses *both* MCP and A2A.

Preprocessing (MCP): A *LoanProcessor* agent receives a new loan application (forms, credit data). It uses MCP to fetch data: calling a credit score API, querying transaction history from a secure database, and OCR-processing uploaded documents. (This step involves one agent calling tools; it could also be broken into subagents but here we treat it as one agent gathering data.)

Decision (A2A): Once structured data is ready, the LoanProcessor creates an A2A task for a **RiskAssessment Agent** to analyze borrower risk. After the risk agent completes, it tasks a **Compliance Agent** via A2A to check regulations. Each agent returns an approval or objections. If all clear, the LoanProcessor sends an A2A task to a **Disbursement Agent** to schedule the fund transfer.

The flow might look like: LoanProcessor (client) → RiskAgent (task: assess risk) → ComplianceAgent (task: verify regulations) → DisbursementAgent (task: disburse funds). All of these communication steps are A2A tasks using the agent cards of those specialist agents. Meanwhile, the LoanProcessor uses MCP internally to handle APIs and data prep.



(Above: The LoanProcessor uses MCP internally to gather data, then coordinates three specialist agents via A2A.)

3. Healthcare Referral Pipeline (A2A-Only)

Scenario: A patient asks for a health check on symptoms. An AI system orchestrates a referral pipeline:

The **Triage Agent** first receives the patient's description. It uses natural language to decide the likely issue (say, "possible migraine"). It then tasks a **Diagnosis Agent** with relevant tests.

The Diagnosis Agent, for example, asks the patient to do a quick vision test (in-streamed as instructions) and reports results back as an artifact. Based on results, it generates a report PDF.

If treatment is required, the Triage Agent then uses A2A to contact a **Medication Agent** and a **Billing Agent** to prescribe drugs and submit insurance claims.

Each specialist agent returns results asynchronously. The Triage Agent may pause ("input-required" state) if it needs a patient's follow-up or consent.

All agents expose their capabilities (e.g. "triage_symptoms", "generate_report") via cards. This autonomous pipeline requires no human to connect the dots: A2A handles hand-offs between triage, diagnosis, pharmacy, and billing agents. (In reality, the human patient or doctor could always monitor or approve steps.)

These examples show agents passing tasks like colleagues: *the focus is on "what to do next" (task content), not on building the internal workflow*. The protocol handles delivery, state, and logging.

4. Autonomous Global Travel & Visa Concierge (A2A + MCP)

Scenario: A user plans a complex international business trip requiring a visa and currency conversion. A primary **Travel Orchestrator Agent** (Client Agent) manages the end-to-end workflow by dynamically discovering and hiring specialized peer agents.

- **Step 1:** *Dynamic Discovery (Agent Cards)* – The orchestrator queries an agent registry to discover a **Visa Processing Agent** and **Currency Exchange Agent** via their `/.well-known/agent.json` Agent Cards, which define their endpoints, authentication, and skills.
- **Step 2:** *Visa Application (A2A)* – The orchestrator initiates a task via `task/send`. The Visa Agent transitions to `input-required` for a passport scan, then moves to `completed` to return a secure visa approval artifact.
- **Step 3:** *Currency Exchange (A2A + MCP)* – The orchestrator creates an A2A task for the Currency agent, which uses MCP internally to query a secure banking database, lock in a quote, and stream the live rate back.
- **Step 4:** *Secure Autonomous Settlement* – Using the Agent Payments Protocol (AP2) or A2A transactional tokens, the orchestrator aggregates fees and presents a unified cart for human approval (Human-in-the-loop) before cryptographically signing payment mandates.

Sample Agent Card and Task JSON

For concreteness, here are illustrative JSON snippets following the A2A specification: **Agent Card (JSON):** Describing an example travel-booking agent. It includes name, description, endpoints, supported formats, capabilities, and security scheme.

JavaScript

```
{
  "name": "TravelBookingAgent",
  "description": "Agent for searching flights and hotels.",
  "provider": "TravelCorp Inc",
  "service": { "url": "https://api.travelcorp.com/a2a" },
  "supportedInput": ["text/plain", "application/json"],
  "supportedOutput": ["application/json"],
  "capabilities": { "streaming": true, "pushNotifications": true },
```

```
  "skills": ["FlightSearch", "HotelBooking"],
  "security": {
    "OAuth2": {
      "flow": "authorizationCode",
      "authorizationUrl": "https://auth.travelcorp.com/authorize",
      "tokenUrl": "https://auth.travelcorp.com/token",
      "scopes": ["travel.read", "travel.write"]
    }
  }
}
```

This card tells clients that *TravelBookingAgent* can take text or JSON requests, stream results, and uses OAuth2 security. It lists two skills so orchestrators can choose it for flight/hotel tasks.

Task JSON: An example of a Task object in progress. It shows the task ID, current status, a message from the agent, and an artifact reference.

JSON

```
{
  "taskId": "task-1234",
  "sessionId": "sess-5678",
  "status": "working",
  "messages": [
    {
      "sender": "agent",
      "timestamp": "2026-03-15T15:34:00Z",
      "parts": [
        { "type": "text/plain", "data": "Looking up available flights..." }
      ]
    }
  ],
  "artifacts": [
    {
      "name": "flight-options",
      "type": "application/json",
      "url": "https://api.travelcorp.com/artifacts/flight-options-1234.json"
    }
  ]
}
```

Here the client agent receives back an updating task: status is `working`, there's a message from the agent, and an artifact link for results. The protocol would allow the agent to update `status` to `completed` and send final data when done.

Limitations, Security, and Human Oversight

A2A unlocks powerful autonomy, but it also introduces new challenges:

Complexity & Maturity: A2A is new. Implementations must handle networking issues, retries, and error states gracefully. Agents must manage partial failures (what if one agent fails?). For now, best practice is *"start small"* and test thoroughly. Many real-world use cases are prototypes; enterprise adoption will grow as the ecosystem matures.

Security Risks: While A2A is secure by default, it assumes agents will validate inputs. A malicious agent could attempt to flood others with bad data or consume resources. Enterprises should implement monitoring, rate limits, and whitelists/blacklists. The spec's OAuth2 approach means every call is authenticated, but governance is vital: define which agents are allowed which scopes.

Data Privacy: Agents only exchange data explicitly included in tasks. However, sensitive data traveling across networks may need extra encryption (beyond HTTPS) or anonymization. Logging all tasks (for audit) also requires compliance with privacy laws. Industries like healthcare must ensure PHI is handled with care.

Trust & Verification: In a multi-org setup, how does one agent trust another? The spec encourages **signed Agent Cards** and well-known public keys so you can verify an agent's identity. Enterprises may require agents to use certificates from recognized CAs. Additionally, agents should only perform permitted actions; tasks should be scoped (agents shouldn't grant each other full access to corporate systems).

Scalability & Performance: The decentralized architecture of A2A, while robust, introduces unique considerations for large-scale deployments.

- **Network Latency:** Every peer-to-peer interaction necessitates a network request/response cycle. In complex, multi-organization workflows, the accumulation of these "network hops" across disparate systems can result in significant latency for sequential task execution.
- **Identity Overhead:** While A2A is secure-by-design, the requirement for OAuth2/OIDC handshakes and cryptographic token validation between

numerous agents adds computational overhead. Architects must account for the load on identity providers to ensure real-time responsiveness.
    - **Discovery Bottlenecks:** Heavy reliance on DNS or centralized registries for agent discovery can create performance bottlenecks. Implementing efficient caching of **Agent Cards** and utilizing high-availability registry services are critical architectural recommendations.

Reliability & LLM Hallucinations: The non-deterministic nature of LLM-powered agents remains a primary technical limitation.

- **Semantic Misalignment:** Agents may suffer from "hallucinations" or semantic drift, misinterpreting the **Message** history or associated **Artifacts**. Mitigating this requires clear data schemas and the use of structured data within **Parts** to ensure robust validation.
- **Error Cascades:** A failure or hallucination in one peer can propagate through the entire agent chain, complicating debuggability. Comprehensive audit logging and end-to-end tracing of **Task** IDs are essential for maintaining system integrity.

Human-in-the-Loop: Despite automation, certain decisions require human review. A2A supports human oversight: for instance, tasks can be paused (`input-required`) awaiting a human input or approval. Audit logs allow humans to review agent interactions. In practice, regulated industries will still have human checks at key points (e.g. a manager must approve a large fund transfer). A2A doesn't eliminate humans – it takes care of routine coordination so humans can focus on strategy.

Limitations of Language: Agents rely on LLMs or fixed logic; they can misunderstand instructions. In a multi-agent setup, semantic misalignment is a risk (one agent's output might be misinterpreted by another). Clear data schemas (enforced via JSON parts) and test conversations help mitigate this. The A2A protocol itself enforces "passing complete context" (via the Task message history), which is better than ad-hoc integrations.

To summarize, A2A provides *structure and standards*, but organizations must still govern agent behavior. The protocol's design (e.g. explicit tasks, OAuth, event streams) facilitates logging and control. As one industry blog notes, **"define agent roles, responsibilities, and permissions, establishing a clear governance model to ensure auditability of multi-agent systems."**. In other words, treat agent networks like microservices: with service accounts, monitoring, and fail-safes.

Recommendations for Architects

For system architects exploring A2A, consider the following roadmap (adapted from A2A.how guidelines):

- **Pilot Projects:** Start with a limited pilot. Choose a single workflow (e.g. an internal helpdesk or a known supply-chain problem) and integrate a few agents. Validate the value and iron out kinks. Keep the data scope small.
- **Security First:** Build the identity/auth layer early. Set up an OAuth2 provider (or use a cloud identity service) for service-to-service auth. Use scoped credentials so agents only see what they need. Encrypt all traffic and consider agent whitelisting.
- **Define Governance:** Create an "agent registry" of approved Agent Cards. Catalog what each agent can do and which tasks it can accept. Log every task exchanged. Assign humans to oversee critical agent interactions.
- **Hybrid A2A+MCP Strategy:** Design workflows that leverage both protocols. Use MCP to attach agents to data (e.g. a DatabaseAgent via MCP), and use A2A to glue agents together. For example, as in the loan workflow, one agent could prepare data via MCP, then hand off to specialist agents via A2A. A2A.how suggests explicitly **combining** both for maximum benefit.
- **Monitoring & Audit:** Instrument agents to emit logs/traces (e.g. using distributed tracing or custom logging on each task ID). Use unique Task IDs to track the end-to-end flow. Monitor agent health and task queues. If possible, deploy dashboards showing real-time task status via streaming updates.
- **Incremental Deployment:** Architect systems so existing tools can be wrapped by an agent. For instance, wrap a legacy CRM API in an "Agent API" that speaks A2A. This makes adding agents non-disruptive. Or start by having one agent speak to others via A2A while still falling back to manual processes if needed.
- **Fail-Safe and Rollback:** Always plan for failure. Agents should have timeouts, and clients should handle exceptions (e.g. revert a transaction if a downstream agent fails). Transactional patterns (like two-phase commits) may be needed for critical operations.
- **Engage the Community:** A2A is evolving. Contribute to open specs, join working groups, and share lessons. Many organizations are experimenting (MCP Market, Google's ADK, LangGraph, etc. support A2A). Keep systems modular so you can adapt as standards stabilize.

Following these patterns, architects can gradually migrate from point-to-point integrations to a more modular agent ecosystem. The **migration path** might be: encapsulate each function as an agent (even if it's your own service), deploy an agent directory, and then convert chatbots or automation scripts to use A2A under the hood. Over time, this will pay off in flexibility and scalability.

Conclusion

A2A represents a significant step toward making multi-agent AI systems practical and reliable. By establishing a common protocol for discovery and task-based communication, it allows diverse agents (from startups to cloud services) to *"talk to each other"* as peers. Combined with MCP's tool integrations, it enables fully autonomous workflows: agents can call out to data sources, then coordinate among themselves to get complex jobs done without humans babysitting every step.

For enterprise architects, the key is to embrace standards early and build on them. A2A adoption will not only streamline current processes but also future-proof systems against fragmentation. As one expert puts it, *"A2A is the missing link in large-scale multi-agent systems"*, offering **structured, discoverable, and scalable** coordination. In the coming years, expect to see agent teams spanning companies (vendors, partners, regulators) all working together over A2A.

However, we should be sober about the journey. Robust implementations will require careful attention to **security, governance, and human oversight**. The goal isn't to eliminate humans, but to offload repetitive decisions so people can focus on strategy. By adopting A2A + MCP wisely (via pilots, clear policies, and monitoring), organizations can achieve *smarter* automation – one where AI agents handle the mechanical work and humans steer the ship.

References

- **A2A Specification v1.0** (Google/A2A community) – core protocol definition
- **A2A Use Cases & Guidance (A2A.how)** – industry scenarios and recommendations
- **Clarifai Blog:** MCP vs A2A Explained – clear exposition of horizontal vs vertical agent protocols
- **Claude Code Docs** (Anthropic) – MCP usage for tools
- **DataCamp A2A Guide** – examples of multi-agent flows and comparison
- **Google Developers Blog:** Introducing A2A – official announcement and goals (Nov 2024)