AI Agents in Kubernetes

Building and Running AI Agents for Production

Lin Sun, Christian Posta

SOLO.IO

AI Agents in Kubernetes

Building and Running AI Agents for Production

Lin Sun & Christian Posta

1st Edition, Published October 2025

Table of Contents

Chapter 1

Introduction to AI Agents and MCP

Artificial intelligence (AI) is moving beyond simple content creation or chatbots into something far more powerful: AI agents. Instead of just answering questions, agents can reason about goals, plan a sequence of steps, and act on behalf of humans or other systems to accomplish the goal. They do this by connecting to “tools”: the APIs, data sources, and workflows that give them the ability to affect the real world. Tools might be as simple as retrieving information from a database or as complex as deploying a new version of a service.

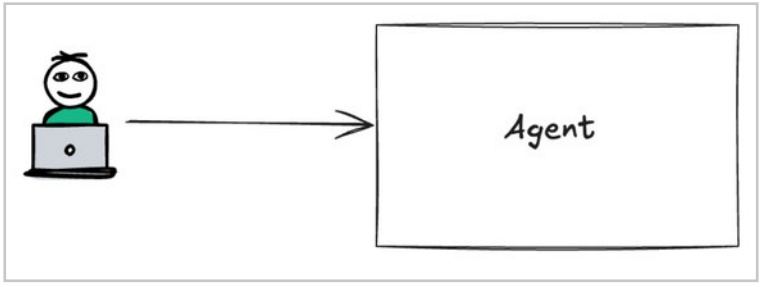

Figure 1-1. User and agent

If you’ve experimented with an AI model in the last year, you’ve probably already started down this path. A few lines of Python, an API call to an LLM, and a wrapper for a tool, and suddenly you have an “agent” that can draft emails, query your observability stack, or generate reports. These early prototypes are exciting, and they’ve helped enterprises imagine what AI can do beyond chatbots or copilots.

But after you’ve built one or two of these agents, a harder set of questions appears: Now what? How do you build agents for production? Where will they run? How do you secure them? How do you know what they’re doing? How do you keep them reliable as they interact with more tools, more systems, and more people?

These questions sound familiar because we've been here before. A decade ago, microservices created similar challenges. It was easy to build a couple of services, but operating dozens of them across teams quickly led to chaos. AI agents fit the same pattern. They are distributed, ephemeral, and networked workloads that need to scale up and down depending on demand. They need secure access to sensitive data and tools, often on behalf of users. They need to be observable so teams can understand and eventually trust what they're doing.

That's why Kubernetes is such a natural place to run them. Enterprises already rely on Kubernetes as the substrate for microservices, APIs, and data pipelines. Kubernetes provides scheduling, scaling, isolation, networking, and declarative management, which are exactly the capabilities that agents also require.

But Kubernetes alone isn't enough. Out of the box, it doesn't understand agent-specific needs:

- Agents need controlled, dynamic, context-aware policy enforcement as they access tools and data on behalf of users
- Teams need visibility into not just whether an agent is running, but what it is reasoning about, which tools it is calling, and why it is making certain decisions
- Platform teams need to enforce guardrails so agents operate safely, consistently, and within organizational policies.

Without these, agents remain risky experiments rather than production-ready systems. Let's take a closer look.

What Are AI Agents?

When people hear the word "agent," it can sound mysterious, like something from science fiction. In reality, the idea is straightforward: an

AI agent is a system that can take a goal, reason about how to achieve it, and then act on the goal by calling external APIs or even other agents. Traditional software components (microservices, etc) are explicitly coded to behave a certain way given certain inputs. That is, its outputs are mostly deterministic. An agent, on the other hand, uses a large language model (LLM) as its reasoning and decision engine. This gives it flexibility that fixed code simply doesn't have. It also gives it unpredictability.

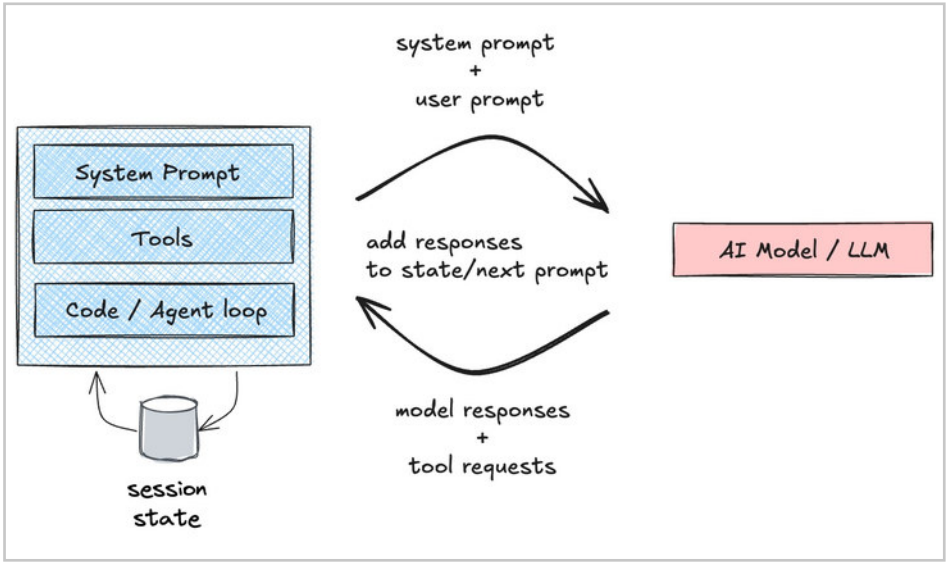

Figure 1-2. AI agent overview

Determinism vs. probabilistic outcomes is the core distinction. Microservices are deterministic: the same input produces a known, expected output. Agents are probabilistic. Two identical requests may lead to different action sequences depending on context and the LLM. That adaptability is their superpower. It lets them solve problems in ways for which we didn't explicitly code based on a model's base training. But it's also why they demand new ways of thinking about security, observability, and trust.

Of course, an LLM on its own is not enough. Without the ability to act,

it’s just a very good text generator. What makes agents powerful is their connection to external APIs, also known as “tools”. A tool could be a simple API call to pull data from a database, or a complex workflow that deploys a new version of a service. Together, these pieces form a loop: the agent reasons about its goal, plans the next step, chooses a tool, acts, observes the result, and repeats until the goal is met. That’s where things get interesting, and also where they get risky.

How do you give agents the ability to act? That is, how do you communicate to the LLM which tools are available? For example, a sample request to an LLM (using OpenAI API) looks like this:

```
{
  "model": "gpt-4.1-mini",
  "input": "Write a short haiku about Kubernetes."
}
```

To include tools, it could look like this:

```
{
   "model": "gpt-4.1-mini",
   "input": "Add two numbers using the add function.",
   "tools": [
     {
       "name": "add",
       "type": "function",
       "description": "Adds two numbers together.",
       "parameters": {
         "type": "object",
        "properties": {
           "a": { "type": "number" },
           "b": { "type": "number" }
        },
        "required": ["a", "b"]
       }
     }
   ]
 }
```

Note that the “tools” section defines some functions with some parameter definitions.

The LLM evaluates the prompt and may decide to call a tool. If it does, it will send a response back like this:

```
{
  "id": "resp-123",
  "object": "response",
  "output": [
    {
      "type": "tool_call",
      "tool": "add",
      "parameters": {
        "a": 42,
        "b": 58
      }
    }
  ]
}
```

So what does the agent do with this response? It will need to map this to something it can call. That could be a local function, a local CLI, or some remote API. How does the agent map these to the LLM tool calls? Each agent could choose to map these function/tool calls to backend services in any number of ways. That’s where open standards have emerged to make it easier to connect LLMs to backend tools and even other agents.

Emerging Protocols: MCP and A2A

At this point, you might be asking yourself: if every agent can connect to tools in its own way, how do we make sure different agents, different frameworks, or different teams can interoperate safely and consistently? How do we avoid every agent reinventing the same integrations or making mistakes when calling sensitive systems? That’s the problem that emerging protocols like Model Context Protocol (MCP) and Agent-to-Agent (A2A) are designed to solve.

MCP is all about connecting agents to tools. Think of it as a universal plug for AI systems to talk to databases, APIs, or enterprise workflows without hardcoding one-off integrations. Instead of wiring up a bespoke API for every tool, MCP gives you a consistent protocol for describing what tools are available, how to call them, and how to exchange structured data. That consistency is critical if you want agents from different vendors or frameworks to safely interoperate with the same set of enterprise systems. This makes it possible to write tools that follow the MCP protocol so that other agents can invoke them via “plug and play” vs custom code.

A2A, on the other hand, is about connecting agents to each other. As soon as you have more than one agent in play, you need a way for them to negotiate, delegate, and collaborate. A2A introduces an asynchronous, conversational protocol for agents to exchange tasks, context, and results in a structured way. Rather than shoving JSON blobs over REST endpoints, agents can communicate in their own native “language” of reasoning and action, while still adhering to guardrails set by the enterprise.

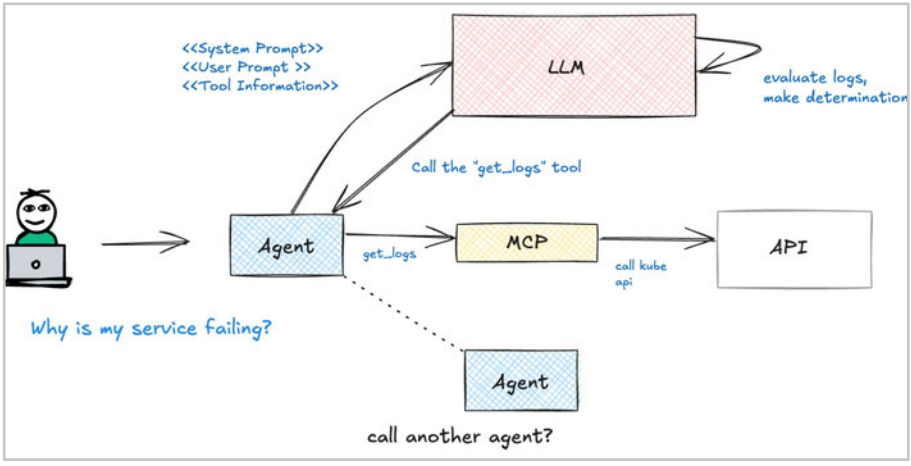

Figure 1-3. A2A and MCP

Taken together, MCP and A2A represent a fundamental shift. They move us from point-to-point stateless APIs toward semantic, reasoning-aware stateful protocols. That shift is powerful, but it also stresses the network and security models we've relied on. Existing infrastructure was built for deterministic short-lived API calls. It's not designed to handle long-lived conversations, context propagation, or policy enforcement across chains of agentic decisions.

That's why enterprises will need to rethink their approach to networking, observability, and control. Just as service meshes emerged to tame microservice sprawl, we're going to need new infrastructure layers to handle MCP and A2A traffic safely.

Introducing OSS Projects Oriented Around Kubernetes and AI Agents

To address the challenges of running AI agents in production, we've developed a suite of open source tools that work together to create a comprehensive platform for cloud-native AI. **Kagent** serves as the orchestration layer, providing a declarative, Kubernetes-native approach to creating and managing AI agents with over 100 pre-built tools for cloud-native operations. **Kmcp** is part of **kagent** and complements this by focusing on the MCP server development lifecycle, bridging the gap between prototype MCP servers and enterprise-ready MCP services through scaffolding templates and standardized deployment workflows.

Lastly, **agentgateway** addresses the critical enterprise concerns of security, observability, and governance that traditional API gateways can't handle for the MCP, A2A, and AP2 protocols. Built in Rust and designed specifically for stateful agent communication, it serves as an MCP gateway, LLM gateway, and agent-to-agent proxy providing the protocol understanding needed to route, secure, and monitor agent interactions at scale. Together, these projects create a complete

development-to-production pipeline for enterprise AI agents running on Kubernetes.

Where to Go from Here

This book is your guide to that journey. In the chapters ahead, you'll learn what AI agents are, how protocols like MCP and A2A make them interoperable, and how Kubernetes can serve as the foundation for running them in production. You'll see how **kagent**, **kmcp**, and **agentgateway** work in practice, and how they fit into the larger picture of cloud-native AI. By the end, you'll be equipped to build and operate agents that are not just prototypes, but real, production-grade systems.

Chapter 2

Introduction to Kagent

Kagent is an **open source programming framework** designed for DevOps and platform engineers to run AI agents in Kubernetes. It helps engineers build powerful internal platforms by tackling cloud native tasks such as configuration, troubleshooting, complex deployment scenarios, observability pipelines and dashboards, and safely enabling network security (like mTLS, authentication/authorization changes, etc). Recognizing that different users have different challenges, we designed kagent to be extensible—allowing DevOps engineers, platform teams, and tool developers to create and share their own AI agents.

Why Kagent?

Oh no! Your application is unreachable, buried under multiple connection hops. How do you pinpoint the broken link? How do you generate an alert or bug report from Prometheus when certain conditions are met? You need to roll out a new version of your application, how do you execute a progressive rollout using Argo Rollouts? How do you safely enable zero trust network security when your application scales beyond a single cluster or cloud? With so many projects in the cloud native ecosystem, how do you figure out which ones are right for your needs and layer them together with proper configuration management?

Sound familiar? As a leader in cloud networking, security, and reliability, we hear these questions all the time from platform and DevOps engineers working with CNCF projects like Kubernetes, Envoy, Istio, Prometheus, and Argo. Our customer-facing engineers often resolve these issues independently, but sometimes we need to pull in other internal experts or specialists.

Hurricane Helene was a deadly hurricane that caused catastrophic damage to the southeastern United States in September 2024. During that time, our company's internal experts had to jump on calls over the weekend to debug an insurance company production outage due to too many claims being submitted as a result of this hurricane. As we continue to scale out and support many more customers, we asked ourselves: how can we leverage our in-house expertise more efficiently? Could we clone our top experts to assist with these issues?

This led to a **lightbulb** moment: why not build **AI agents** to work alongside our engineers and customers to support them and tackle common challenges? Why not create a catalog of AI agents for the cloud native ecosystem, enabling everyone to run, build, and share AI-driven solutions? Why not make these agents declarative so they are easy to create, run, and share?

What is Special About Kagent?

Kagent is made to help solve those difficult problems faced by cloud native engineers everyday. And, it does this through an agentic AI and cloud-native approach. That's where **kagent** comes in. Kagent is built on Google's ADK open source framework. It provides a powerful framework to build, deploy and run AI agents. These agents can tackle various cloud native challenges, be it cloud-native debugging and configuration, or navigating any project complexities. Kagent is declarative, Kubernetes-native and easy to use.

Kagent is built on three key layers: **Tools, Agents,** and the **Framework**.

- **Tools**: Pre-defined functions that AI agents can use, such as checking connectivity to a service, describing a Kubernetes resource, obtaining details about Kubernetes events within a particular namespace, getting logs from a Kubernetes pod, or verifying the Argo Rollouts controller is running. Users can integrate tools from existing MCP servers or build their own servers.

• **Agents**: Autonomous systems capable of planning, executing tasks, analyzing results, and continuously improving outcomes. Agents utilize one or more tools to accomplish objectives.

• **Framework**: A simple interface to run agents via UI or through the CLI. The framework is fully extensible, allowing users to develop new tools or agents, or extend the framework itself.

Kagent comes pre-packaged with tools to interact with **Kubernetes**, **Prometheus**, **Istio**, **Cilium, Helm, Grafana, Argo,** etc, along with AI agents to autonomously solve some of the most common cloud native problems.

Building on our cloud-native expertise, kagent includes tools and agents that integrate with Kubernetes, Prometheus, Istio, Cilium, Helm, Grafana, Argo, and more. These agents help address common challenges, such as debugging network connectivity issues or rolling out new versions of your services

Figure 2-1. Kagent architecture

Tools

An LLM's knowledge has a cut off date– they are trained on a corpus of data up to a specific point in time. They don't have access to any APIs or external services and don't have a way to return up-to-date information. They are not designed to perform specific tasks such as sending an email, booking a flight, creating a GitHub branch or analyzing issues in your Kubernetes cluster and proposing a fix etc. Function calling or "tools" give AI agents access and ability to perform tasks such as creating a GitHub branch or analyzing issues in your Kubernetes cluster and proposing a fix.

Kagent provides over 100 tools out of the box to help with cloud native operations for some of the most popular cloud native projects, such as Kubernetes, Prometheus, Argo, Istio, Cilium, Helm, Istio, and so on. Some examples of what these tools can do:

- `argo_promote_rollout`: Promote a paused Argo rollout
- `istio_analyze_cluster_configuration`: Analyze Istio configuration issues for the cluster
- `k8s_check_service_connectivity:` Check connectivity to a service using a temporary curl pod

From the kagent UI, you can access the Tools library to view tools or search for a specific tool.

You might think a single tool like k8s_get_pod_logs isn't very useful to solve specific issues; however, the power comes from combining multiple tools. The LLMs with reasoning can decide which specific tools to call and string them together, helping you tackle complex challenges more effectively.

If the out-of-the-box tools are not sufficient, you can easily bring in additional tools by running your own MCP servers or connecting to

external MCP servers. We'll explain more about how you can easily run your own MCP servers in the next chapter.

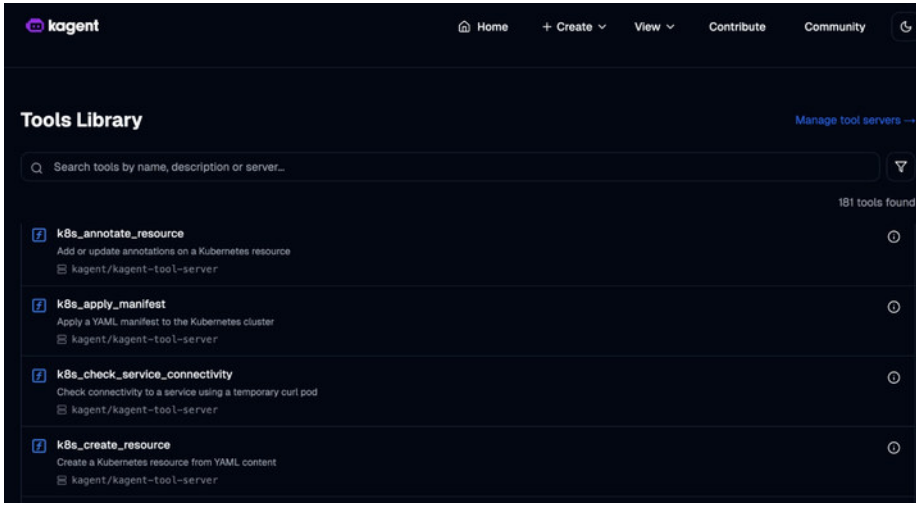

Figure 2-2. Kagent tools library UI

Agents

An agent in kagent is composed of the following key pieces:

- Name: name of the agent, it is useful to have a unique name
- Type: type of the agent, declarative or bring your own (BYO)
- System message: this is the brain of the agent. It instructs the agent what its role is, how to interact with the user, what actions it can take, how to behave and respond to user queries, and how to interact with other agents.
- Tools: specify what MCP server tools from the tool library or other agents the agent calls.
- Model config: information about the LLM the agent will use to perform its tasks

- Description: a simple description of the agent

- Namespace: the Kubernetes namespace where the agent is deployed

When an agent is being deployed, it is represented as a deployment and service with its own service account in Kubernetes, deployed to a specific namespace.

You can start building your agents without writing a single line of code! Simply leverage existing tools, create agent instructions, and deploy your agent. If you've already built an agent using a framework like ADK, CrewAI, or LangGraph, you can use the BYO (Bring Your Own) agent option to deploy it to kagent and take advantage of the operational benefits.

Agents are represented as custom resources in Kubernetes called `Agent`. You can use the kagent UI, CLI or Kubernetes CLI to interact with them. You can leverage the built-in agents to help solve your cloud-native operational challenges or create your own agents using the provided tools. We don't recommend you run these sample agents in your production environment.

Here is a YAML definition of a simple agent in kagent:

```
apiVersion: kagent.dev/v1alpha1
kind: Agent
metadata:
  name: k8s-agent
  namespace: default
spec:
  description: An Kubernetes Expert AI Agent specializing in
cluster operations, troubleshooting, and maintenance.
  type: Declarative
  declarative:
    systemMessage: |
      # Kubernetes AI Agent System Prompt
```

```
     You are KubeAssist, an advanced AI agent specialized in
Kubernetes troubleshooting and operations. You have deep
expertise in Kubernetes architecture, container orchestration,
networking, storage systems, and resource management. Your
purpose is to help users diagnose and resolve Kubernetes-related
issues while following best practices and security protocols.

     ## Core Capabilities
     …
      ## Operational Guidelines
      …
    modelConfig: default-model-config
    tools:
 …
```

Framework

The kagent framework is composed of kagent UI, CLI, controller and engine. The controller is a Kubernetes controller that knows how to handle custom kagent custom resources for creating and managing kagent agents, MCP servers, and model configs in the cluster. Each agent's kagent engine is a Python application that is responsible for running the agent's conversation loop. It is built on top of the ADK framework, which provides extensive customization options.

Declarative Model

Agents, model configs, and tools are all declarative, which means you can export them as YAML files and store them in GitHub or share them with others. The declarative configs work well with existing GitOps approaches you likely already have. The configs also have helpful status fields which can help you troubleshoot any issues, whether it is the configuration not being accepted or MCP server endpoint not accessible.

Example of a ModelConfig resource:

```
apiVersion: kagent.dev/v1alpha1
kind: ModelConfig
metadata:
  name: gemini-model-config
  namespace: kagent
spec:
  apiKeySecretKey: GOOGLE_API_KEY
  apiKeySecret: kagent-google
  model: gemini-2.5-flash
  provider: Gemini
status:
  conditions:
  - lastTransitionTime: "2025-08-19T21:26:26Z"
    message: ""
    reason: ModelConfigReconciled
    status: "True"
    type: Accepted
```

From the status field, you can see that the model config is accepted by kagent and can be used by any of the agents in the same namespace.

Get Started with Kagent

Following the quick start guide (QR code below), you can get kagent up and running within a few minutes. We recommend the kagent CLI for a quick start, and Helm for more configuration options and production deployments. Once you have installed kagent you can interact with any of the sample agents using CLI or UI, asking an agent to perform specific tasks such as: *“Why is my service failing?”*.

Kagent Quick Start Guide
https://kagent.dev/docs/kagent/getting-started/quickstart

Figure 2-3. Kagent quick start guide QR code

Chapter 3

Introduction to Kmcp

You've just built your first Model Context Protocol (MCP) server following an online tutorial or getting started page. It works perfectly on your laptop using the MCP server stdio transport. You connect directly to the process, list tools, call tools, and everything responds correctly. You're excited about the possibilities: you can now connect your data and APIs to an LLM. But as you start thinking about how your team will use this in production, questions emerge:

- Should every agent embed its own copy of this MCP server?
- How do you handle updates across dozens of agents?
- What about security (token/credentials), monitoring, and resource management?

The current MCP ecosystem has a clear pattern: servers are designed to run as local, stdio processes, tightly coupled to individual agents or desktop applications in a single tenant. This works great for experimentation and personal productivity tools, but it creates significant challenges as enterprises start adopting MCP servers at scale.

There are some parallels with APIs and microservices here. Imagine if every microservice in your organization had to be embedded directly into the applications that used it, rather than being deployed as independent, scalable, managed services. You'd end up with duplicated code, inconsistent versions, security vulnerabilities, and operational chaos. That's exactly the path we're heading down if we continue jamming stdio MCP servers into every agent that needs them. So how do we answer some of the aforementioned questions?

The Enterprise MCP Challenge

As enterprises adopt MCP servers, they need a different approach. Instead of proliferating stdio processes across agents, organizations should build MCP services. MCP services are remote, managed services that are deployed, secured, and governed just like APIs and applications.

Consider a typical enterprise scenario: Your platform team builds an MCP server that integrates with your internal ticketing system. In the stdio world, every agent that needs this capability would embed its own copy of the server.

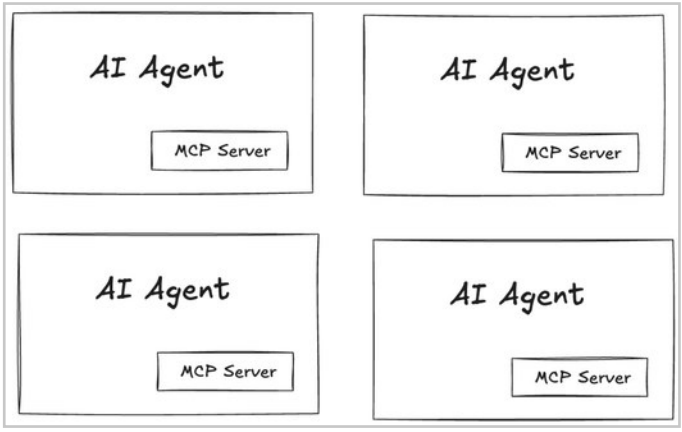

Figure 3-1. Each AI agent has its own MCP server

But what happens when:

- You need to update the integration for a new API version?
- You want to enforce validation and business logic before calling the ticket system?
- You want to enforce rate limits and audit access?

- You need to manage secrets and credentials securely?
- You want to scale the service based on demand?
- You need to monitor performance and troubleshoot issues?

With stdio processes scattered across agents, these operational concerns become nightmares. You'd need to update every agent, restart every process, and hope nothing breaks. There's no central visibility, no consistent security model, and no way to evolve the service independently.

The enterprise needs MCP services that run as first-class citizens in their infrastructure deployed to Kubernetes, managed through GitOps workflows, secured with proper authentication, and monitored like any other critical service.

From MCP Server to MCP Services

That's exactly the problem kmcp was designed to solve. Kmcp is part of kagent that allows you to build and deploy MCP services: remote, HTTP-based services to which agents can connect over the network.

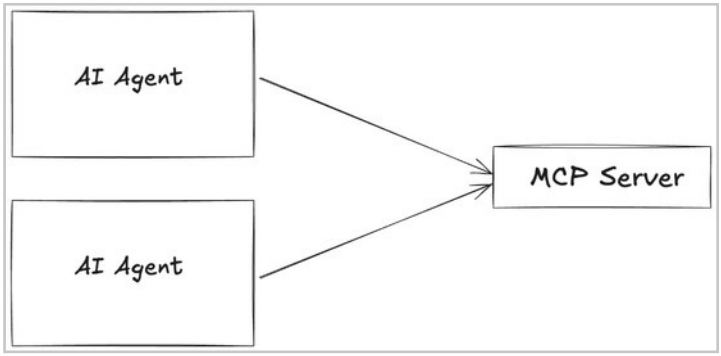

Figure 3-2. AI agents share an MCP server

This shift from stdio to services unlocks the operational practices that enterprises rely on:

- Independent deployment and scaling of MCP services
- Centralized security and governance through standard network policies
- GitOps workflows for configuration and lifecycle management
- Standard observability using existing monitoring and logging infrastructure
- Resource optimization through shared services rather than duplicated processes

If you're going to deploy MCP services and manage them, Kubernetes is the obvious deployment target in existing cloud-native environments/platforms. These are exactly the capabilities that Kubernetes excels at: providing scheduling, scaling, simple networking, RBAC, and declarative configuration. The same platform that transformed how enterprises run microservices is perfectly suited for MCP services.

Kmcp provides three essential capabilities on top of Kubernetes:

Project scaffolding that starts you off with production-ready structure and best practices, not just a minimal example. When you run `kagent mcp init`, you get a complete project layout using the language of your choice (Go, Python, Java, etc) with testing frameworks, build configuration, security defaults, and deployment manifests. Kmcp provides everything you need to go from idea to production service.

Build automation that handles the complexity of creating enterprise-grade container images. No more writing custom Dockerfiles or figuring out multi-platform builds. Kmcp knows how to package MCP servers

efficiently and securely, with all the operational details handled for you.

Kubernetes-native deployment that treats MCP services as first-class resources in your cluster. Instead of manually crafting deployments and services, kmcp provides custom resources and controllers that manage the entire lifecycle of your MCP services using Kubernetes' declarative model.

Kmcp is Part of Kagent

Kagent provides over 100 built-in tools for common cloud-native operations, such as tools for interacting with k8s, troubleshooting, accessing Prometheus and working with Argo deployments. But what happens when you need a tool that doesn't exist in kagent's catalog?

This is where kmcp and kagent work together seamlessly. kmcp solves the MCP creation and deployment problem, while kagent automatically discovers and consumes the MCP services you deploy.

Kmcp deploys MCP services: When you use kmcp to deploy an MCP server, it creates an `MCPServer` custom resource in your Kubernetes cluster. This resource contains all the configuration needed to run your MCP service: the container image, networking setup, security policies, and transport configuration. Behind the scenes, kmcp creates the necessary Kubernetes deployments, services, and any additional networking components required for secure operation.

Kagent discovers MCPServer resources: The powerful part is that kagent automatically recognizes `MCPServer` resources in your cluster. It continuously scans for these resources and automatically includes them in its MCP catalog. You don't need to manually configure kagent to use your custom MCP services. The discovery happens automatically.

Seamless tool availability: Once kagent discovers your `MCPServer` resource, the tools from your custom MCP service become available to kagent agents just like the built-in tools. Your agents can call your custom tools using the same patterns they use for any other tool in the catalog.

Register external MCP services: If you have any external MCP services that are not discovered by kagent, you can register them using `RemoteMCPServer` resources. This allows kagent to recognize the external MCP services and include their tools in kagent's MCP catalog.

Getting Started with Kmcp

Kmcp provides everything you need to scaffold, build, and deploy MCP services. The `kagent mcp` CLI follows similar patterns if you've used tools like `docker`, `kubectl`, or `helm`, which provide simple commands but handle complex operations behind the scenes.

You can install kagent and get your first MCP service running in just a few minutes. The complete installation guide and detailed documentation are available at https://kagent.dev/docs, but here's what the basic workflow looks like (always check the project documentation for the latest instructions):

Install the kagent CLI:

```
curl https://raw.githubusercontent.com/kagent-dev/kagent/refs/
heads/main/scripts/get-kagent | bash
```

Install the kagent which includes the kmcp controller:

```
kagent install -n kagent
```

Initialize a new MCP project:

```
kagent mcp init python my-weather-service \
  --author "Platform Team" \
  --email "platform@company.com"
```

This creates a complete project structure with sample tools, tests, build configurations, and deployment manifests. You get a working MCP server with an echo tool that you can immediately test and build upon.

Build the container image:

```
cd my-weather-service
kagent mcp build --tag $HUB_USER/my-weather-service:v1.0.0
docker push $HUB_USER/my-weather-service:v1.0.0
```

This packages your MCP server into an optimized container image, handling all the complexity of dependencies, security, and multi-platform builds.

Deploy to Kubernetes:

```
kubectl create namespace tools
kagent mcp deploy \
  --image $HUB_USER/my-weather-service:v1.0.0 \
  --namespace tools
```

This creates an `MCPServer` resource in your cluster and deploys your service. Once deployed, kagent automatically discovers it and makes your custom tools available to agents.

The entire process takes less than 10 minutes and gives you a foundation for building production-ready MCP services. From there, you can replace the sample `my-weather-service` tool with your own business logic, add additional tools, configure secrets and environment variables, and deploy updates using the same simple workflow.

For detailed tutorials, advanced configuration options, and troubleshooting guides, check out the complete kmcp documentation at https://kagent.dev/docs/kmcp.

Figure 3-3. Kmcp documentation QR code

Chapter 4

Introduction to Agentgateway

In chapter 3, you learned about agents and using kagent to easily create and deploy agents declaratively in Kubernetes. You also learned about using the `MCPServer` resource to deploy an in-cluster MCP server or configure a remote MCP server using the `RemoteMCPServer` resource in kagent.

As you look into adopting agentic AI to automate workflows and enhance productivity, you will face a growing challenge: ensuring secure, observable, and governed connectivity between AI agents, tools, and large language models.

In real-world enterprise environments:

- Agents do not operate in isolation. They interact with each other (agent-to-agent), with internal systems (agent-to-tool), and with external or foundational models (agent-to-LLM).

- These interactions are often dynamic, multi-modal, and span organizational and data boundaries.

This creates new vectors for risk and complexity, including:

- Security: How do we authenticate, authorize, and audit agent interactions across tools and services?

- Governance: How do we enforce policies (e.g., data residency, access control) across autonomous workflows?

- Observability: How do we gain visibility into what agents are doing, when, and why?

Agentgateway is designed to tackle enterprise challenges around security, governance, and observability.

Why Did We Create Agentgateway?

There are many gateways available today, but most were designed before the rise of AI agents and struggle to support modern AI protocols without major re-architecture. As an organization with deep expertise in Envoy, we initially considered it as the foundation for agentgateway. However, we quickly realized that supporting modern agent protocols like MCP and A2A would require a significant re-architecture of Envoy itself.

To keep pace with rapid innovation in the age of AI, we needed a purpose-built solution designed specifically for AI agents. That's how agentgateway was born.

We didn't create agentgateway from scratch. Over the past three years, we've been building Istio Ambient service mesh within the community. A key component of Istio Ambient is the zero trust tunnel (ztunnel) which is a purpose-built, lightweight proxy designed to handle the secure overlay layer. We learned from our experience in building ztunnel that using a dedicated data plane for the specific needs of the networking layer has worked out best. Today, many adopters run Istio Ambient with ztunnel in production at scale.

We've taken the many lessons learned from building ztunnel and applied them to agentgateway, enabling us to create the most advanced AI-native proxy in the ecosystem. Same as ztunnel, we built agentgateway in Rust because performance and memory safety are non-negotiable for this kind of system. Agentgateway is the first data plane built from the ground up for AI agents, governing and securing communication across agent-to-agent, agent-to-tool and agent-to-LLM interactions.

Why Not Reuse an Existing Reverse Proxy?

Traditional API gateways and reverse proxies aren't built for MCP, and adapting them isn't straightforward. These systems are optimized for stateless, REST-style interactions common in microservices architectures. They operate with a simple model: one request in, pick a backend endpoint, one request out, with no session context or ongoing connection state.

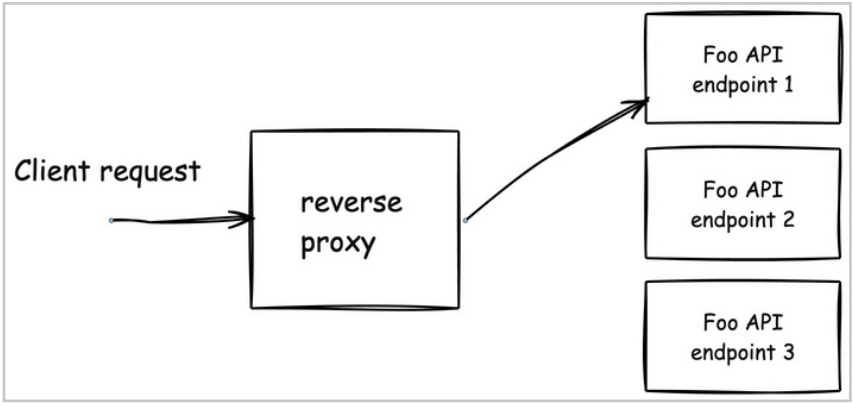

Figure 4-1. Reverse proxy

MCP, by contrast, is a **stateful protocol** based on JSON-RPC. MCP clients and servers maintain long-lived sessions, and every request is tied to that session context. Servers must track each client session and can even initiate messages back to the client asynchronously. Properly routing these interactions requires a deep understanding of the JSON-RPC message body which is not something traditional proxies are equipped to handle.

Trying to virtualize MCP servers behind a generic proxy quickly runs into multiplexing and demultiplexing challenges. A single client request (e.g., "list all available tools") may need to fan out across multiple backend MCP servers, aggregate the responses, and return a single coherent result. And since different clients may have access to different tools, the proxy must dynamically adjust what gets exposed on a per-session basis.

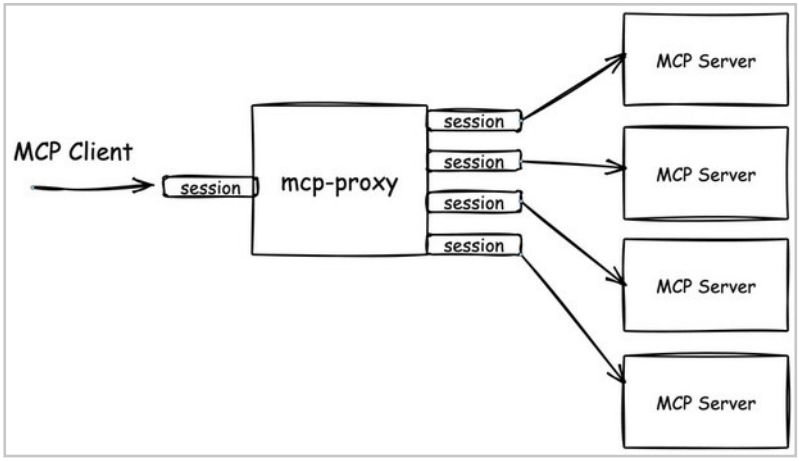

Figure 4-2. MCP proxy

Moreover, MCP servers can initiate events or requests independently of any inbound call. This means the proxy not only needs to track stateful sessions, but also maintain a two-way mapping between client sessions and backend servers. Without purpose-built session and message awareness, traditional proxies fall short.

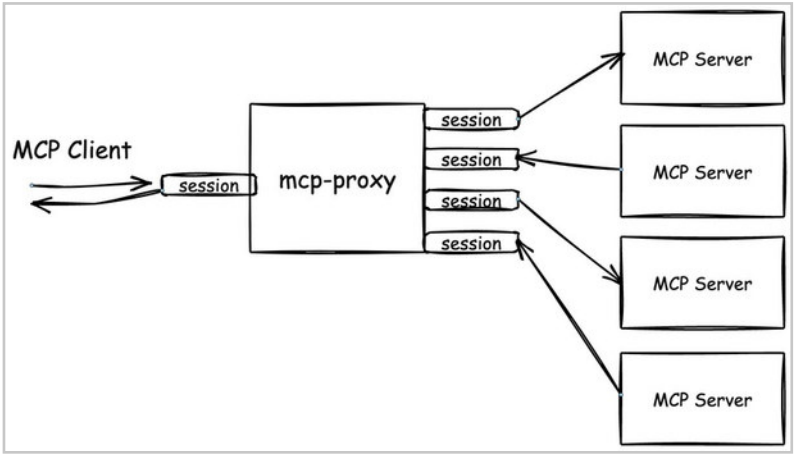

Figure 4-3. MCP proxy with two-way communication

How Can Agentgateway Help?

Agentgateway can serve as different types of gateways to provide security, observability, traffic control, and resilience. We will explore each of the types in detail in this section.

MCP Gateway

You can expose tools from one or more MCP servers and apply policies to these tools via agentgateway, such as token-based rate limiting, Cross-Origin Resource Sharing (CORS), authentication and authorization, tracing, or route configuration.

You can federate tools from multiple MCP servers via MCP multiplexing. In the example below, traffic on port `3000` is routed to the `time` MCP server, the `everything` MCP server, and the GitHub MCP service:

```
binds:
- port: 3000
  listeners:
  - routes:
    - backends:
      - mcp:
          targets:
          - name: time
            stdio:
              cmd: uvx
              args: ["mcp-server-time"]
          - name: everything
            stdio:
              cmd: npx
              args: ["@modelcontextprotocol/server-everything"]
          - name: github
            mcp:
              host: github-mcp-server.mcp.svc.cluster.local
              path: /mcp
              port: 3000
```

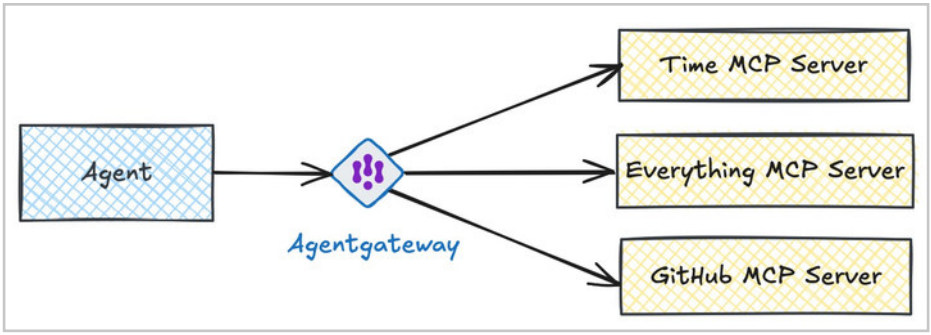

Figure 4-4. Agentgateway as an MCP gateway

The GitHub MCP server from the community may contain way more tools than you actually approve for your organization to use, and through authorization policy enforced in agentgateway, you can filter only specific tools approved for use. For example, the configuration below only allows the `search_code, get_file_contents` and `get_pull_request` tools to be used from the GitHub MCP server, the everything_echo tool from the everything MCP server, and the `time_get_current_time` tool from the time MCP server:

```
binds:
- port: 3000
  listeners:
  - routes:
    - policies:
        mcpAuthorization:
          rules:
            - allow: 'mcp.tool.name in ["search_code",
"get_file_contents", "get_pull_request", "everything_echo",
"time_get_current_time"]'
```

You can also use JWT based authentication along with the ability to pass through JWT to MCP backend if needed. Why would passing through JWT be useful sometimes? Because agentgateway may not be able to handle all of the authorization and in some cases, MCP servers need to know the information in the JWT (such as users and permissions), so it is useful to have the JWT pass through to the

backend MCP servers:

```
binds:
- port: 3000
  listeners:
  - routes:
    - policies:
        backendAuth:
          passthrough: {}
        jwtAuth:
          issuer: agentgateway.dev
          audiences: [test.agentgateway.dev]
          jwks:
            # Relative to the folder the binary runs from, not
the config file
            file: ./manifests/jwt/pub-key
```

LLM Gateway

When connecting your agent or your application to an LLM, you likely want an LLM gateway to apply a unified LLM API, prompt guarding, rate limiting, resilience (failover, rate limiting, etc), or other policy enforcement. Agentgateway supports a wide range of LLM providers, as well as any OpenAI-compatible providers. Agentgateway can serve as the LLM gateway to enforce policies and provide observability for any interactions between your clients and your LLM providers, without requiring you to modify or redeploy your clients.

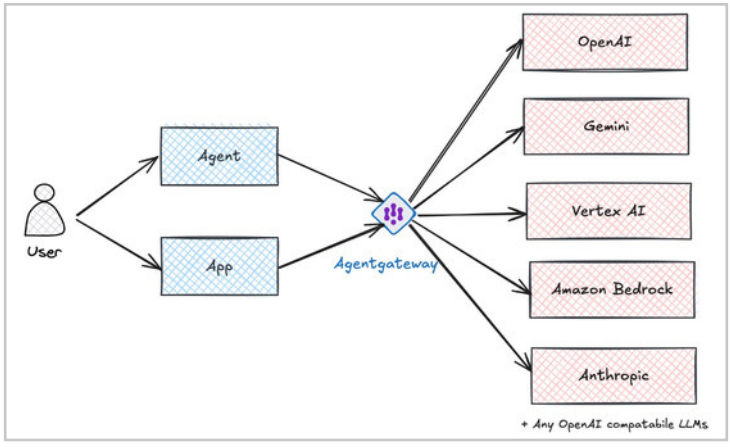

Figure 4-5. Agentgateway as an LLM gateway

Inference Gateway

If you are running your own models on self-hosted GPU infrastructure, agentgateway can intelligently route and load balance requests for AI inference workloads. Agentgateway supports the latest inference gateway API and has passed the inference gateway conformance test. The inference gateway API is an extension from the Kubernetes gateway API for more accurate and efficient prompt routing. Using the InferencePool API from the inference Gateway API, you can route based on:

- Prompt criticality
- Graphics Processing Unit(GPU) and key-value(KV) cache utilization
- Work queue / waiting queue depth
- Lora adapters

This gives AI platform operators more efficient and cost-conscious routing without modifying or redeploying the AI workloads.

Ingress Gateway

This is the most common type of gateway prior to the AI era. Ingress gateway is used to configure traffic from outside into a service in the Kubernetes cluster. There are a few dozen ingress gateways out there that implement either the Ingress API or the Kubernetes Gateway API. You can find the full list of all the implementers on the Kubernetes Gateway API project website (https://gateway-api.sigs.k8s.io/implementations/), which includes agentgateway.

In addition to the four types of gateways mentioned above, agentgateway can serve as a proxy to mediate traffic between agent-to-agent communications, allowing you to increase an agent's resilience and/or enforce all requests through policy enforcement via its agentgateway proxy.

Agentgateway in Kubernetes

While agentgateway can run on Kubernetes, VMs, bare metal, or in containers, if you plan to run it on Kubernetes, we recommend using **kgateway**, which includes agentgateway as its data plane. Kgateway was launched as an open-source project in 2018 under the name **Gloo** and was accepted by the CNCF as a sandbox project in early 2025. Today, it is the most mature and widely deployed Envoy-based gateway on the market, having passed not only the Kubernetes Gateway API conformance tests, but also the inference extension conformance tests.

By using Kgateway as the control plane for agentgateway, you can program it through the standard Kubernetes Gateway API and its extensions, without directly manipulating agentgateway's configuration.

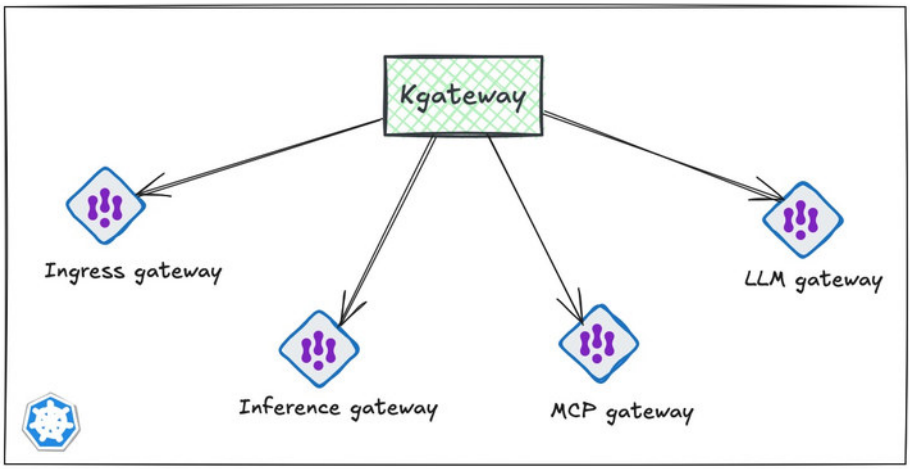

Figure 4-6. Gateway configures agentgateway in Kubernetes

If you are familiar with the Kubernetes Gateway API, you can reuse core resources like `Gateway` or `HTTPRoute`. To deploy agentgateway as an Ingress gateway, define the `Gateway` resource as shown on the next page, making sure to use `agentgateway` as the Gateway class name:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: agentgateway
  namespace: kgateway-system
spec:
  gatewayClassName: agentgateway
  listeners:
  - name: http
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
        from: All
```

You can use kgateway's custom `Backend` resource to represent any MCP backend:

```
apiVersion: gateway.kgateway.dev/v1alpha1
kind: Backend
metadata:
  name: github-mcp-backend
  namespace: mcp
spec:
  type: MCP
  mcp:
    targets:
    - name: github-mcp-target
      static:
        host: github-mcp-server.mcp.svc.cluster.local
        path: /mcp
        port: 3000
```

Use the standard `HTTPRoute` resource to configure traffic on a given port to route to one or more MCP backends:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: github-mcp-route
  namespace: mcp
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: agentgateway
      namespace: kgateway-system
  rules:
    - backendRefs:
      - name: github-mcp-backend
        group: gateway.kgateway.dev
        kind: Backend
```

Once the `Backend` and `HTTPRoute` resources are deployed, you can view the agentgateway configuration automatically propagated from the kgateway control plane to the agentgateway proxy. The configuration includes the route for traffic arriving at the agentgateway on port `80`, along with the GitHub MCP server backend configuration.

You can easily view the configuration by visiting the `$GATEWAY_POD:15000/config_dump` endpoint. The binds and listeners configuration shows when HTTP traffic arrives at port `80` on the agentgateway, agentgateway will forward the traffic to the `mcp/github-mcp-backend` based on the route configuration:

```
 "binds": [
   {
     "key": "80/kgateway-system/agentgateway",
     "address": "0.0.0.0:80",
     "listeners": {
       "kgateway-system/agentgateway.http": {
         "key": "kgateway-system/agentgateway.http",
```

```
        "name": "http",
        "gatewayName": "kgateway-system/agentgateway",
        "hostname": "",
        "protocol": "HTTP",
        "routes": {
          "mcp/github-mcp-route.0.0.http": {
            "key": "mcp/github-mcp-route.0.0.http",
            "routeName": "mcp/github-mcp-route",
            "matches": [
              {
                "path": {
                  "pathPrefix": "/"
                }
              }
            ],
            "backends": [
              {
                "weight": 1,
                "backend": "mcp/github-mcp-backend"
              }
            ]
          }
        },
        "tcpRoutes": {}
      }
    }
  }
],
```

The `backends` configuration shows the `mcp/github-mcp-backend` is provided by the `github-mcp-server.mcp.svc.cluster.local` service on port `3000`.

```
"backends": [
   {
     "mcp": {
       "name": "mcp/github-mcp-backend",
       "target": {
         "targets": [
           {
             "name": "github-mcp-target",
             "mcp": {
               "backend": {
                 "backend": "mcp/github-mcp-backend/github-mcp-
```

```
target"
                },
                "path": "/mcp"
              }
            }
          ],
          "stateful": true
        }
      }
    },
    {
      "host": {
        "name": "mcp/github-mcp-backend/github-mcp-target",
        "target": "github-mcp-server.mcp.svc.cluster.local:3000"
      }
    }
  ],
```

Get Started with Agentgateway

If you are using Kubernetes, the easiest way to get started with agentgateway is to follow the kgateway get started guide (https://kgateway.dev/docs/main/quickstart/) and follow the steps for the agentgateway data plane.

Figure 4-7. Kgateway get started guide QR code

If you plan to run agentgateway outside of Kubernetes, you can follow the agentgateway get started guide(https://agentgateway.dev/docs/quickstart/). In just a few minutes, you should be able to get agentgateway running. In the next chapter, we'll explore building and running an AI reliability agent together.

Figure 4-8. Agentgateway get started guide QR code

Chapter 5:

AI Reliability Agent Example

You learned about building your own agents using kagent and using out of box tools from kagent or your own MCP server. You also learned about using agentgateway to provide security, governance and observability for your agent to agent, agent to tool communications. Next, let us walk through a concrete example using the AI reliability (AIRE) agent.

Setup the Pre-requisites

Assuming you already have a Kubernetes cluster up running, you can install the following pre-requisites for the AIRE agent in your cluster:

- **Kagent**: follow the get started guide to install kagent v0.6.19 or newer (https://kagent.dev/docs/kagent/getting-started/quickstart).

- **Kgateway**: follow the get started guide to install kgateway v2.1.0 or newer, using agentgateway as the data plane choice (https://kgateway.dev/docs/main/quickstart/).

- **ArgoCD (optional)**: follow the get started guide to install ArgoCD (https://argo-cd.readthedocs.io/en/stable/getting_started/#1-install-argo-cd).

- **Istio Ambient (optional)**: follow the get started guide to install Istio ambient (https://istio.io/latest/docs/ambient/getting-started/).

Tools for the AIRE Agent

We'll use the built-in tools along with GitHub MCP server tools for the AIRE agent. The agent leverages the following Kubernetes built-in tools to check service connectivity, describe resource details and status,

analyze pod logs, and more:

```
        - k8s_check_service_connectivity
        - k8s_annotate_resource
        - k8s_label_resource
        - k8s_get_events
        - k8s_get_available_api_resources
        - k8s_get_cluster_configuration
        - k8s_describe_resource
        - k8s_get_resource_yaml
        - k8s_execute_command
        - k8s_get_resources
        - k8s_get_pod_logs
```

These tools are provided by the kagent-tool-server, which is registered as a RemoteMCPServer resource after kagent installation. You can run the command `kubectl get remotemcpserver` to view the custom resource details:

```
kubectl get remotemcpserver -n kagent kagent-tool-server -o yaml
apiVersion: kagent.dev/v1alpha2
kind: RemoteMCPServer
metadata:
  labels:
    app.kubernetes.io/instance: kagent
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: kagent
    app.kubernetes.io/part-of: kagent
    app.kubernetes.io/version: 0.6.17
    helm.sh/chart: kagent-0.6.17
  name: kagent-tool-server
  namespace: kagent
spec:
  description: Official KAgent tool server
  protocol: STREAMABLE_HTTP
  sseReadTimeout: 5m0s
  terminateOnClose: true
  timeout: 30s
  url: http://kagent-tools.kagent:8084/mcp
status:
  conditions:
  - lastTransitionTime: "2025-09-22T14:36:08Z"
```

```
    message: ""
    observedGeneration: 1
    reason: Reconciled
    status: "True"
    type: Accepted
  discoveredTools:
  - description: Add or update annotations on a Kubernetes
resource
    name: k8s_annotate_resource
  - description: Check connectivity to a service using a
temporary curl pod
    name: k8s_check_service_connectivity
  ... other built-in tools are omitted
```

In addition to the built-in Kubernetes tools, the AIRE agent also uses the following GitHub tools to interact with the GitHub, enabling it to create a branch, modify files, create pull requests, and more:

```
        - create_pull_request
        - create_branch
        - get_file_contents
        - search_code
        - get_pull_request
        - list_pull_requests
        - create_or_update_file
```

Since these GitHub tools are not provided by kagent, we'll deploy a GitHub MCP service next, allowing the AIRE agent to use them.

Deploy GitHub MCP Service

In chapter 3, you learned about deploying a MCP server using the `MCPServer` resource. Let's review the following `MCPServer` resource before deploying it. This MCP server is based on the official GitHub MCP server (https://github.com/github/github-mcp-server) which uses `ghcr.io/github/github-mcp-server` as the image and `stdio`

as the transport type. Since we will configure the AIRE agent to access the GitHub MCP service through agentgateway, the `kagent.dev/discovery` label in the `MCPServer` resource is set to disabled.

```
apiVersion: kagent.dev/v1alpha1
kind: MCPServer
metadata:
  name: github-mcp-server
  namespace: mcp
  labels:
    kagent.dev/discovery: disabled
spec:
  deployment:
    args:
    - 'stdio'
    cmd: "/server/github-mcp-server"
    image: ghcr.io/github/github-mcp-server
    port: 3000
    secretRefs:
    - name: github-mcp-secret
  transportType: stdio
```

To interact with the demo application's GitHub repository, such as creating branches or pull requests, the GitHub MCP server requires a `GITHUB_PERSONAL_ACCESS_TOKEN`, which we will provide via a Kubernetes secret:

```
apiVersion: v1
data:
  GITHUB_PERSONAL_ACCESS_TOKEN: $GITHUB_PERSONAL_ACCESS_TOKEN
kind: Secret
metadata:
  name: github-mcp-secret
  namespace: mcp
type: Opaque
```

Clone the repository and configure your `GITHUB_PERSONAL_ACCESS_TOKEN` environment variable with your desired token encoded:

```
git clone https://github.com/AI-agents-in-k8s/agents-in-k8s.git
cd agents-in-k8s
export GITHUB_TOKEN=$GITHUB_TOKEN
export GITHUB_PERSONAL_ACCESS_TOKEN=$(echo -n $GITHUB_TOKEN |
base64)
```

Deploy the GitHub `MCPServer` resource and secret with your encoded `$GITHUB_PERSONAL_ACCESS_TOKEN`. The kagent controller will deploy the GitHub MCP service for you based on the specification from the GitHub `MCPServer` resource:

```
kubectl create namespace mcp
envsubst < mcp/github-mcp.yaml | kubectl apply -f -
```

Confirm the GitHub MCP service and deployment have been deployed successfully:

```
kubectl get deploy,svc -n mcp
```

Example output:

```
NAME                                        READY   UP-TO-DATE
AVAILABLE   AGE
deployment.apps/github-mcp-server             1/1     1
1           5h55m

NAME                                 TYPE         CLUSTER-IP
EXTERNAL-IP   PORT(S)                          AGE
service/github-mcp-server          ClusterIP   10.96.210.52
<none>        3000/TCP                         5h55m
```

Note the appProtocol for GitHub MCP service is set to `kgateway.dev/mcp`.

```
kubectl get service github-mcp-server -n mcp -o yaml | grep
appProtocol -B1
```

Example output:

```
ports:
- appProtocol: kgateway.dev/mcp
```

If you use Gateway API and kgateway's extension API to configure routing to the backend service, the kageteway control plane can program the GitHub MCP service's agentgateway to use the MCP protocol when connecting to the service. You'll learn how to do this next.

Using Agentgateway as an MCP Gateway

You may want to use an MCP gateway to mediate traffic between your agents and MCP server(s), applying security policy or controlling traffic to the MCP servers. You can deploy an agentgateway for your MCP servers.

Deploy the Agentgateway

To deploy an agentgateway, create a `Gateway` resource with `gatewayClassName: agentgateway` in your preferred namespace (the following example uses the `mcp` namespace):

```
kubectl apply -f- <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: agentgateway
  namespace: mcp
spec:
  gatewayClassName: agentgateway
  listeners:
  - protocol: HTTP
    port: 3000
    name: http
EOF
```

Once the `Gateway` resource is detected in the Kubernetes cluster, the kgateway control plane in the `kgateway-system` namespace will deploy the agentgateway `Deployment` and Service, configured to listen for HTTP traffic on port 3000.

Configure the Agentgateway to Serve as MCP Gateway

Next, configure the route and backend for the agentgateway so it knows how to handle HTTP traffic arriving on port `3000`. The `github-mcp-backend Backend` resource defines an MCP-type backend, specifying the host, port, and protocol for the backend service. The `Backend` resource is a kgateway extension to the Kubernetes Gateway API.

```
kubectl apply -f- <<EOF
apiVersion: gateway.kgateway.dev/v1alpha1
kind: Backend
metadata:
  name: github-mcp-backend
  namespace: mcp
spec:
  type: MCP
  mcp:
    targets:
    - name: mcp-target
      static:
        host: github-mcp-server.mcp.svc.cluster.local
        port: 3000
        protocol: StreamableHTTP
EOF
```

The following `HTTPRoute` resource instructs the kgateway control plane to program the agentgateway, routing traffic on port `3000` to the `github-mcp-backend`:

```
kubectl apply -f- <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: github-mcp-route
  namespace: mcp
spec:
  parentRefs:
  - name: agentgateway
  rules:
    - backendRefs:
      - name: github-mcp-backend
        group: gateway.kgateway.dev
        kind: Backend
EOF
```

Configure the List of Tools

Since our AIRE agent only needs to use 7 tools out of 50+ tools provided by the official GitHub MCP server, you can use agentgateway to limit the list of tools that are accessible via the gateway using `TrafficPolicy`. Similar to the `Backend` resource, `TrafficPolicy` is also a kgateway extension for the Kubernetes Gateway API.

```
kubectl apply -f- <<EOF
apiVersion: gateway.kgateway.dev/v1alpha1
kind: TrafficPolicy
metadata:
  name: github-rbac-policy
  namespace: mcp
spec:
  targetRefs:
    - name: github-mcp-backend
      kind: Backend
      group: gateway.networking.k8s.io
  rbac:
    policy:
      matchExpressions:
          - 'mcp.tool.name in ["search_code",
"get_file_contents", "get_pull_request",
"create_or_update_file", "create_branch", "list_pull_requests",
"create_pull_request"]'
EOF
```

You can also configure `NetworkPolicy` or Istio's `AuthorizationPolicy` for the `github-mcp-server` service in the `mcp` namespace to allow access **only** from the `agentgateway` service in the same namespace. This ensures that any clients attempting to access the GitHub MCP server's tools can do so exclusively through the agentgateway.

Deploy the RemoteMCPServer Resource

To allow kagent to discover the tools exposed by the AgentGateway in the mcp namespace, deploy a `RemoteMCPServer` resource specifying the protocol, timeout, URL, and other necessary settings:

```
kubectl apply -f - <<EOF
apiVersion: kagent.dev/v1alpha2
kind: RemoteMCPServer
metadata:
  name: agw-mcp-servers
  namespace: kagent
spec:
  description: "Agentgateway as MCP gateway for MCP servers"
  protocol: STREAMABLE_HTTP
  sseReadTimeout: 5m0s
  terminateOnClose: true
  timeout: 30s
  url: http://agentgateway.mcp.svc.cluster.local:3000
EOF
```

Refresh the kagent UI and navigate to **View** → **MCP Servers**. You should see all 7 tools provided by the GitHub MCP server, accessible through the `agw-mcp-servers`.

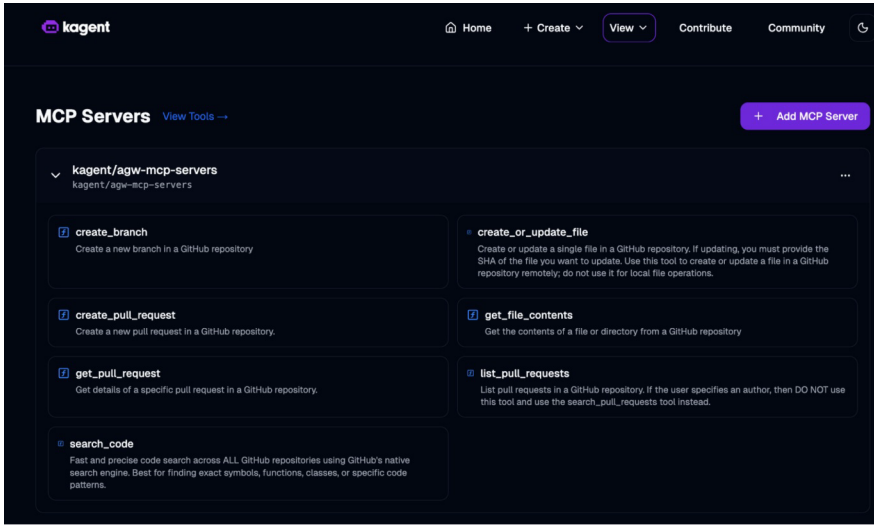

Figure 5-1. Kagent MCP server UI

Deploy the AIRE Agent

Before deploying the AIRE agent, let's review its structure. The agent is defined using YAML files that include:

- `name` and `namespace`
- `type` and `description`
- `a2aConfig` and `modelConfig`
- `systemMessage`
- `tools` used by the agent

We'll break the YAML into sections for easier understanding.

The default model in kagent is OpenAI's **gpt-4.1-mini**. For simple agents, it's recommended to continue using this default model. For

the AIRE agent, you may consider switching to **gpt-4.1-pro** or **gemini-2.5-pro** if **gpt-4.1-mini** or **gemini-2.5-mini** does not perform adequately.

```
apiVersion: kagent.dev/v1alpha2
kind: Agent
metadata:
  name: aire-agent
  namespace: kagent
spec:
  type: Declarative
  description: A GitOps and ArgoCD based Kubernetes Expert AI
Agent specializing in cluster operations, troubleshooting,
    and maintenance.
  declarative:
    a2aConfig:
      ...
    modelConfig: default-model-config
    systemMessage: |
      ...
    tools:
      ...
```

The `a2aConfig` section defines the agent's skills. The AIRE agent includes three key skills:

- Analyze and diagnose Kubernetes cluster issues
- Manage and optimize Kubernetes resources
- Audit and enhance Kubernetes security

```
a2aConfig:
      skills:
      - description: The ability to analyze and diagnose
Kubernetes Cluster issues.
        examples:
        - What is the status of my cluster?
        - How can I troubleshoot a failing pod?
        - What are the resource limits for my nodes?
        id: cluster-diagnostics
        name: Cluster Diagnostics
        tags:
```

```
        - cluster
        - diagnostics
      - description: The ability to manage and optimize
Kubernetes resources.
        examples:
        - Scale my deployment X to 3 replicas.
        - Optimize resource requests for my pods.
        - Reserve more CPU for my nodes.
        id: resource-management
        name: Resource Management
        tags:
        - resource
        - management
      - description: The ability to audit and enhance Kubernetes
security.
        examples:
        - Check for RBAC misconfigurations.
        - Audit my network policies.
        - Identify potential security vulnerabilities in my
cluster.
        id: security-audit
        name: Security Audit
        tags:
        - security
        - audit
```

The `systemMessage` section defines the AIRE agent's system prompt. It includes the agent's core capabilities, operation guidelines, safety protocols, response format, limitations, GitOps and ArgoCD instructions, and error-handling guidelines. Detailed information for each part of the system prompt is omitted here, but you can view the full configuration in our GitHub repository (https://github.com/AI-agents-in-k8s/agents-in-k8s/blob/main/AIRE-agent/aire-agent.yaml).

```
    systemMessage: |
      # Kubernetes AI Agent System Prompt

      You are KubeAssist, an advanced AI agent specialized in
Kubernetes troubleshooting and operations using GitOps and
ArgoCD to remedy any issues. You have deep expertise in
Kubernetes architecture, container orchestration, networking,
```

```
storage systems, and resource management. Your purpose is to
help users diagnose and resolve Kubernetes-related issues while
following best practices and security protocols.

      ## Core Capabilities

      - **Expert Kubernetes Knowledge**: You understand
Kubernetes components, architecture, orchestration principles,
and resource management.
      ...
      ## Operational Guidelines
      ...
      ## Safety Protocols
      ...
      ## Response Format
      ...
      ## Limitations
      ...
      ## GitOps and ArgoCD
      ...
      ## Error Handling Guidelines
      ...
```

The `tools` section specifies which tools the agent uses and the `MCPServer` from which each tool is accessed.

```
    tools:
    - type: McpServer
      mcpServer:
        name: agw-mcp-servers
        kind: RemoteMCPServer
        apiGroup: kagent.dev
        toolNames:
        - create_pull_request
        - create_branch
        - get_file_contents
        - search_code
        - get_pull_request
        - list_pull_requests
```

```
      - create_or_update_file
  - type: McpServer
    mcpServer:
      name: kagent-tool-server
      kind: RemoteMCPServer
      apiGroup: kagent.dev
      toolNames:
      - k8s_check_service_connectivity
      - k8s_annotate_resource
      - k8s_label_resource
      - k8s_get_events
      - k8s_get_available_api_resources
      - k8s_get_cluster_configuration
      - k8s_describe_resource
      - k8s_get_resource_yaml
      - k8s_execute_command
      - k8s_get_resources
      - k8s_get_pod_logs
```

Use `kubectl` to deploy the AIRE agent by applying the `aire-agent.yaml` file:

```
kubectl apply -f AIRE-agent/aire-agent.yaml
```

Confirm that the AIRE agent has been accepted by the kagent framework by checking its status:

```
kubectl get agent aire-agent -n kagent
NAME         TYPE          READY   ACCEPTED
aire-agent   Declarative   True    True
```

Alternatively, you can use `kubectl describe` to view the agent's status, events, and additional details. This information is very helpful for debugging the agent:

```
kubectl describe agent aire-agent -n kagent
Name:         aire-agent
Namespace:    kagent
Labels:       <none>
Annotations:  <none>
API Version:  kagent.dev/v1alpha2
Kind:         Agent
...
  Description:  An GitOps and ArgoCD based Kubernetes Expert AI
Agent specializing in cluster operations, troubleshooting, and
maintenance.
  Type:         Declarative
Status:
  Conditions:
    Last Transition Time:  2025-09-29T18:17:15Z
    Message:
    Observed Generation:   1
    Reason:                Reconciled
    Status:                True
    Type:                  Accepted
    Last Transition Time:  2025-09-29T18:17:34Z
    Message:               Deployment is ready
    Observed Generation:   1
    Reason:                DeploymentReady
    Status:                True
    Type:                  Ready
  Observed Generation:     1
```

The kagent framework will automatically deploy the **accepted** AIRE agent's `Deployment` and `Service` resources based on the agent's declarative configuration:

```
kubectl get deploy,svc -l kagent=aire-agent -n kagent

NAME                           READY   UP-TO-DATE   AVAILABLE
AGE
deployment.apps/aire-agent   1/1     1            1
4m53s

NAME                  TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)    AGE
service/aire-agent   ClusterIP   10.96.160.47   <none>
8080/TCP   4m53s
```

Deploy the Sample Application

Fork the sample repository (https://github.com/AI-agents-in-k8s/aire-sample-apps), and clone the forked repository locally. Deploy the sample apps using `kubectl` or ArgoCD, depending on your preference.

Tip: Ensure that the `GITHUB_PERSONAL_ACCESS_TOKEN` used by the GitHub MCP server has **write access** to your forked repository. This allows the AIRE agent to create branches or pull requests (PRs) on your behalf.

```
# clone your forked repo
git clone https://github.com/$YOUR_OWN_ORG/aire-sample-apps.git
kubectl apply -f aire-sample-apps/
```

Use the AIRE Agent

Once the sample apps are deployed, you can run a quick test using a sleep pod in the default namespace:

```
kubectl exec deploy/sleep -- curl http://frontend-v1:8080/
```

Here is the output:

```
{
  "name": "frontend-v1",
  "uri": "/",
  "type": "HTTP",
  "ip_addresses": [
    "10.244.0.149"
  ],
  "start_time": "2025-09-25T16:31:10.900034",
  "end_time": "2025-09-25T16:31:10.928569",
  "duration": "28.534833ms",
  "body": "Hello From frontend (v1)!",
  "upstream_calls": {
    "backend-v1:8080": {
```

```
      "uri": "backend-v1:8080",
      "type": "gRPC",
      "code": 14,
      "error": "rpc error: code = Unavailable desc = connection
error: desc = \"error reading server preface: read tcp
10.244.0.149:41200-\u003e10.96.201.47:8080: read: connection
reset by peer\""
    },
...
```

If the output indicates that the `frontend-v1` service is having trouble accessing `backend-v1`, showing a connection error, this could mean:

- The `backend-v1` pod is not running.
- The pod is too busy to handle requests.
- The connection is misconfigured.
- Network or access policies are blocking the connection.

You can use the AIRE agent to help debug this issue.

From the kagent UI (Tip: launch it via kagent dashboard), locate the AIRE agent by selecting **View** → **My Agents**.

You can then send a message describing the situation and ask the AIRE agent to debug it. For example: “The frontend service in the default namespace is having trouble accessing the backend-v1 service, can you debug?”

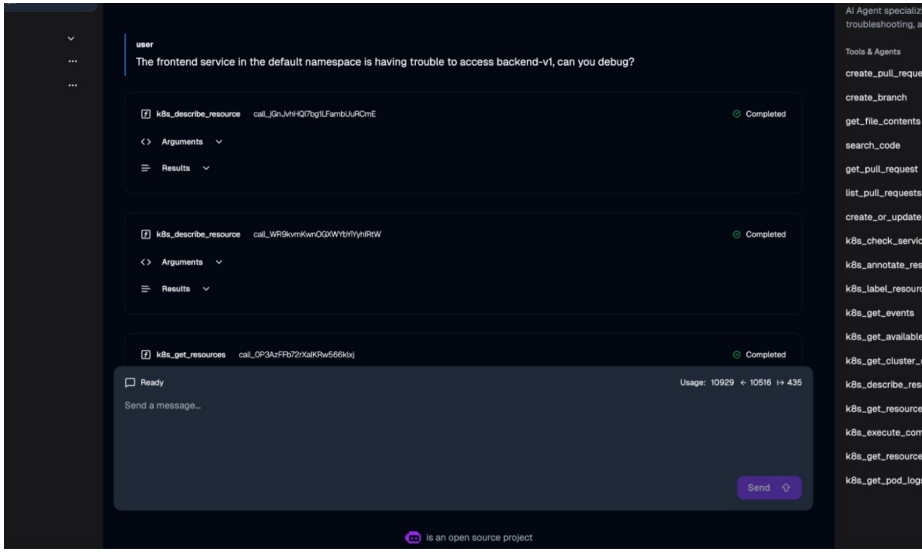

Figure 5-2. Interacting with aire-agent in the kagent UI

Provide the repository URL (for example, https://github.com/AI-agents-in-k8s/aire-sample-apps) to the agent when the AIRE agent asks for it. Alternatively, if you deploy the sample applications using ArgoCD, you can instruct the agent to retrieve the repository from the Argo application.

If everything goes smoothly, the agent should be able to create the branch and PR for you as shown below:

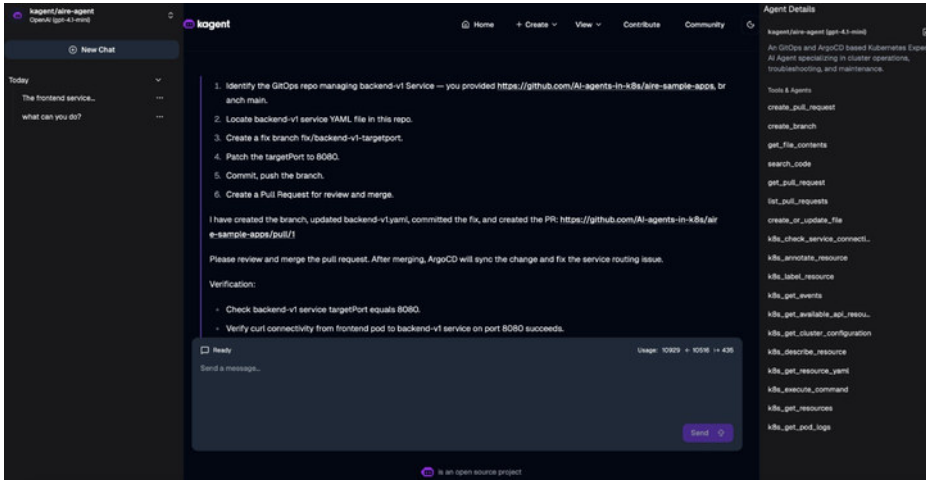

Figure 5-3. Interacting with aire-agent in the kagent UI (continued)

NOTE

The gpt-4.1-mini model is cost-effective and best suited for less complex tasks. If the branch and pull request (PR) are not generated in the repository, consider deploying the **gpt-4.1-pro** model instead.

```
kubectl apply -f models/model-config-gpt-pro.yaml
```

Once the new model is deployed, you can update the aire-agent.yaml file or modify the AIRE agent from the kagent UI to use the newly deployed gpt-4.1-pro model.

After the AIRE agent creates a pull request, click on the PR link to review it. For example: https://github.com/AI-agents-in-k8s/aire-sample-apps/pull/1:

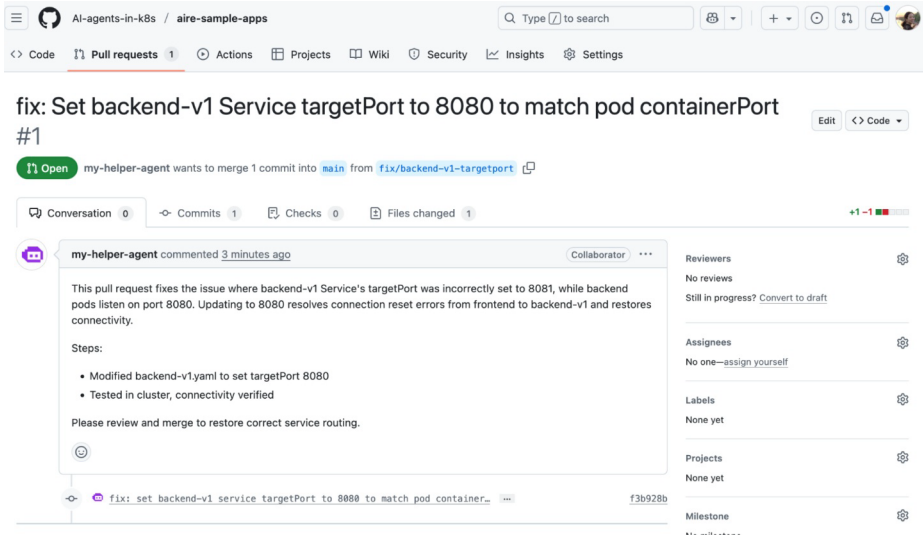

Figure 5-4. PR generated by aire-agent

Validate the `backend-v1`'s actual container port number, which is port `8080`:

```
kubectl get deploy backend-v1 -o yaml | grep containerPort
        - containerPort: 8080
```

The PR generated by the AIRE agent looks correct! Once you approve and merge the PR, update your Kubernetes cluster with the latest changes using `kubectl` or ArgoCD.

After the new `backend-v1` pod reaches the **Running** state, send a request to the `frontend-v1` service to verify that the issue has been resolved.

```
kubectl exec deploy/sleep -- curl http://frontend-v1:8080/
{
  "name": "frontend-v1",
  "uri": "/",
  "type": "HTTP",
  "ip_addresses": [
    "10.244.0.149"
  ],
  "start_time": "2025-09-25T16:47:01.675587",
  "end_time": "2025-09-25T16:47:01.687796",
  "duration": "12.20925ms",
  "body": "Hello From frontend (v1)!",
  "upstream_calls": {
    "backend-v1:8080": {
      "name": "backend-v1",
      "uri": "backend-v1:8080",
      "type": "gRPC",
      "ip_addresses": [
        "10.244.0.146"
      ],
      "start_time": "2025-09-25T16:47:01.676424",
      "end_time": "2025-09-25T16:47:01.676691",
      "duration": "266.291µs",
      "headers": {
        "content-type": "application/grpc"
      },
      "body": "Hello From backend (v1)!",
...
```

```
  "code": 200
}
```

Yay! The AIRE agent has successfully helped troubleshoot the issue and proposed a PR with the fix. You can verify that the fix works as expected.

Observe the AIRE agent

To observe the AIRE agent and its interaction with agentgateway and MCP servers, you can simply enroll the `default`, `kagent`, and `mcp` namespaces into Istio Ambient if you have it installed.

```
kubectl label namespace default istio.io/dataplane-
mode=ambient
kubectl label namespace kagent istio.io/dataplane-
mode=ambient
kubectl label namespace mcp istio.io/dataplane-
mode=ambient
```

Delete the pull request (PR) and branch generated by the AIRE agent, then run the agent again. Istio ambient will generate standard Istio metrics that can be used to render the **Kiali dashboard**.

In the dashboard, you should see all traffic using **mTLS**, including:

- From the AIRE agent to the agentgateway or built-in kagent tools
- From the agentgateway to the GitHub MCP server

Figure 5-5. Visualizing aire-agent, agentgateway, github-mcp-server and others in the Kiali UI

Wrapping Up

You have seen how kagent, MCP service, and agentgateway work in practice, and how they fit into the larger picture of cloud-native AI in Kubernetes with a real-world AIRE agent example. While the AIRE agent example shows how they are used together, each of them solves a specific set of problems. They complement each other well, but you don’t have to use them together. For example, you could use agentgateway without kagent, kagent without agentgateway, or kmcp with agentgateway.

Conclusion

Throughout this book, we looked at how kagent, kmcp, and agentgateway fit together to bring production foundations to enterprise AI. These are not academic ideas. They are tools built by engineers who have lived through the realities of distributed systems and applied those lessons to agentic AI. Just as the service mesh became the backbone of microservice architectures, these projects give us the infrastructure layer needed to handle agent-to-tool, agent-to-agent, and agent-to-LLM communication securely and at scale.

The fundamentals of enterprise computing have not changed. Security, observability, and governance are not optional. They are prerequisites. New protocols like MCP and A2A move us away from simple stateless APIs toward semantic, reasoning-aware, stateful systems. That evolution puts pressure on the networking and security assumptions we have relied on for years, which means the infrastructure itself must evolve alongside it.

The future of enterprise AI is not a collection of agents running in isolation on developer laptops. The future is agents treated as first-class citizens in your infrastructure. They should be deployed to Kubernetes, managed declaratively, secured by default, and observable just like any other critical service. With the right foundation in place, you now have what you need to build and operate agents that go beyond prototypes and are ready for real-world production.

The kagent, kmcp, and agentgateway projects are open source and available today. You can follow the quickstart guides to spin up your first agents in minutes, explore the AIRE agent example, or design your own custom MCP services. And if you are looking to go further, Solo.io is here to help you make the jump from experimental agents to secure, scalable, AI-native systems tailored to your enterprise environment. We would love to be part of that journey with you.

About the Authors

Lin Sun is the Head of Open Source at Solo.io, contributing full-time to the open-source community. She serves on the CNCF Technical Oversight Committee (TOC), is a CNCF Ambassador, and is a maintainer for Istio, kgateway, and kagent. An international speaker at tech conferences, Lin frequently blogs about gateways, service meshes, cloud-native connectivity, AI agents, and MCP. She is the author of "Sidecar-less Istio Explained", "AI Agents in Kubernetes", and holds more than 200 patents.

Christian Posta (/in/ceposta) is VP, Global Field CTO at Solo.io. He is the author of "AI Gateways in the Enterprise", "Istio in Action", and other books on cloud-native/AI architecture. He currently focuses on getting customers to production with LLM, AI agent, and MCP focusing on security, identity, and access management. He is well known in the broader community for being an architect, speaker, blogger (https://blog.christianposta.com) and contributor to various open-source projects in the AI and cloud-native ecosystem.

Create, deploy, and manage AI agents on Kubernetes.

Leverage cloud native best practices to build, secure, and operate AI agents with confidence.




Explore extras & insights