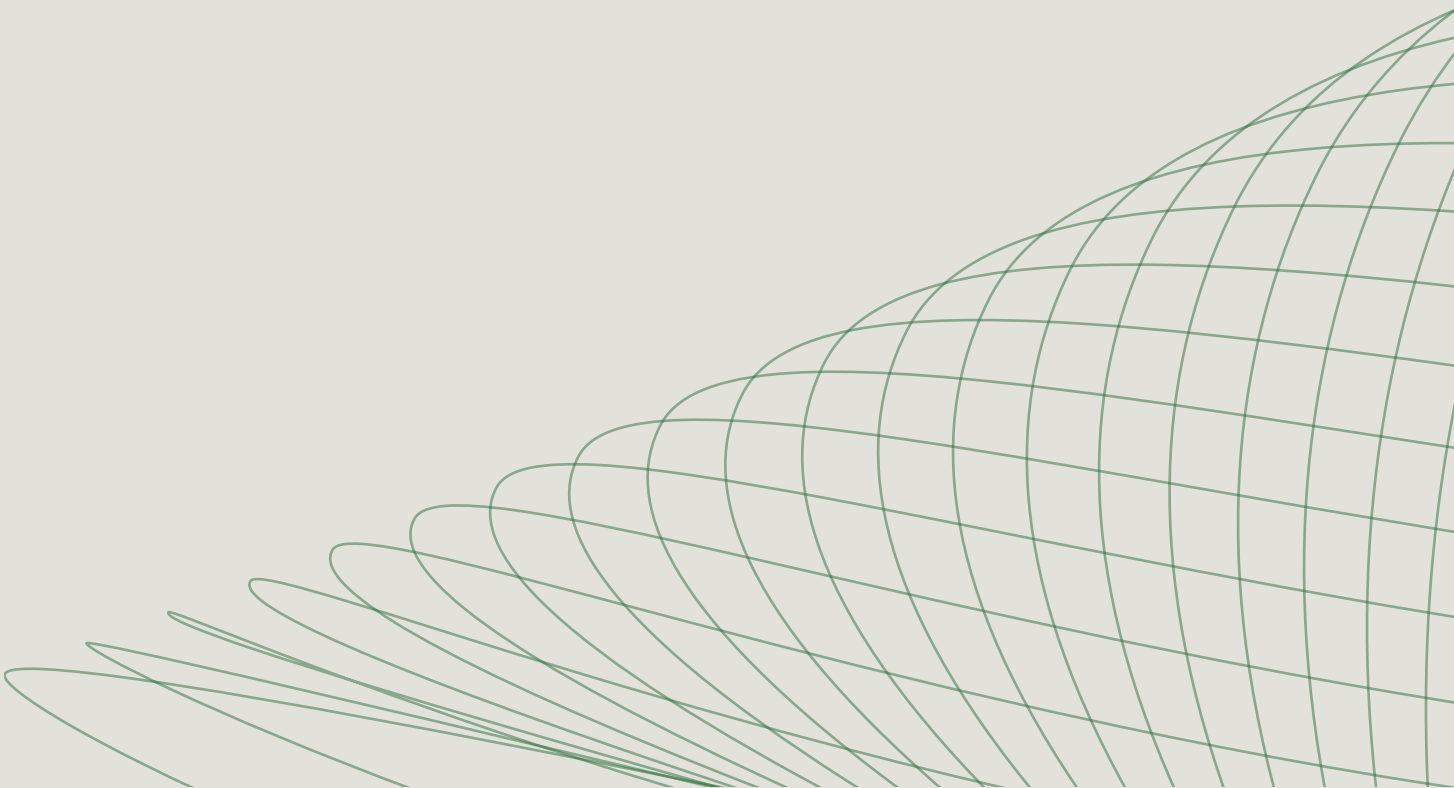
STACKLOK

How to Run AI Agents on Kubernetes

A practical guide for platform and systems engineers



Contents

The infrastructure problem nobody warned you about

Your organization has committed to AI agents. Your CIO and CTO have a mandate. Developers are already experimenting (sometimes with your blessing, and sometimes without). And somewhere in your infrastructure, MCP servers are starting to appear … built by different teams, deployed in different ways, governed by nobody in particular.

You've seen this movie before. Not with agents, but with containers. In the early days of containerization, teams ran workloads wherever they could: bare metal, VMs, whatever was available. There was no standard way to manage them in production. Then Kubernetes emerged and gave platform teams one place to solve the five hard problems: isolation, identity, connectivity, ingress, and observability.

AI agents are at the exact same inflection point. Everyone is building them. Almost nobody is running them in production with real governance. The infrastructure patterns haven't settled.

This guide is for the platform and systems engineers who will be responsible for making AI agents production-grade. It makes the case that Kubernetes is the right control plane from which to build and govern agent infrastructure, explains why MCP servers are the connective tissue that makes agents genuinely useful, and walks through the architectural steps to get your organization there.

Why Kubernetes is the right foundation for AI agents

The instinct to reach for a new, purpose-built platform for AI agents is understandable. Agents feel new. But the problems they create in production are not new at all.

Consider what an enterprise AI agent actually needs to run safely:

Isolation

An agent that can call tools across your internal systems (e.g. Salesforce, Databricks, ServiceNow, GitHub) needs hard boundaries. One misconfigured tool call, one prompt injection, one compromised server shouldn't be able to cascade across your infrastructure. You need sandboxing at the workload level, not just at the application layer.

Identity

When an agent acts on behalf of a user, downstream systems need to know which user, not just that the AI agent made a request. Audit logs that show a service account instead of a human identity are useless for compliance. You need end-to-end identity from the developer's IDE or knowledge worker's client all the way to the data source.

Connectivity

Agents need a consistent, governed way to discover and invoke tools across your systems. Without it, every team builds their own integration pattern, and you end up with a different security posture for every data source.

Ingress

Not every user should have access to every tool. Not every tool should be available in every context. You need a single enforcement point for access control, rate limiting, and audit logging, not distributed policy logic scattered across individual servers.

Observability

When an agent makes a decision you didn't expect, or a tool call fails in production, you need a complete trace: which user, which agent, which tool, which inputs and outputs, at what time. Without it, you can't debug, audit, or improve.

These are exactly the five problems Kubernetes was designed to solve for container workloads; they map directly to what agent infrastructure requires.

If your organization already runs Kubernetes in production, you are not starting from zero. You have the orchestration layer, the networking primitives, the secrets management, the RBAC framework, the observability stack. Instead of building from scratch, you can extend what you already have.

The role of MCP in making agents actually useful

An AI agent without access to your internal systems is a capable but isolated assistant. It can reason, summarize, and generate, but it can't act on real data, update records, or integrate with the tools your teams depend on.

The Model Context Protocol (MCP) is the emerging standard that changes this. It provides a consistent interface through which an agent can discover and invoke tools. With MCP, your AI agent can query a database, retrieve a Jira ticket, look up a customer record and/or trigger a deployment.

Each MCP server exposes a set of tools that an agent can invoke. A Databricks MCP server exposes data access tools. A GitHub MCP server exposes repository tools. A ServiceNow MCP server exposes incident management tools. When an agent is connected to the right MCP servers, it can do real work across your stack.

This is why MCP matters for enterprises, but it also introduces a new governance problem. Every MCP server is a potential access point into your internal systems. Without a platform to govern them, you end up with the same sprawl problem that plagued containers before Kubernetes: ungoverned workloads proliferating across your environment, each with its own security posture, and without central visibility or control.

The answer is to treat MCP servers as first-class infrastructure that are deployed, versioned, governed, and observed using the same patterns you already apply to your production workloads.

The architecture: MCP on Kubernetes

Running MCP servers on Kubernetes gives you the right foundation. But governance requires more than deployment. It requires a control plane that spans the full lifecycle, from how servers are discovered and vetted, to how they are deployed, isolated and audited.

The architecture that follows is organized into four layers. Each layer addresses a distinct set of problems. Implemented together, they give you the production-grade foundation that security teams, platform teams, and developers all need.

Layer 1: The Registry — governed discovery

The first problem is discoverability. In most organizations today, developers find MCP servers the same way they find NPM packages. They search the internet, find something that looks right, and run it. Ungoverned servers proliferate through convenience.

A governed registry changes this. It is the authoritative catalog of approved MCP servers in your organization that are vetted, attested, and ready to deploy. When a developer needs a tool, the registry is where they look first. When a security team needs to know what's running in the environment, the registry is the source of truth.

The registry operates at two levels. For external and open-source servers, it provides a vetting layer: servers are pulled from upstream sources, assessed for security and compliance, and promoted into the internal registry only after passing your organization's acceptance criteria. For internally-built servers, it provides a publishing mechanism; teams can register their own tools and make them discoverable to the rest of the organization.

The practical effect is significant. Instead of seven different teams building seven slightly different Jira integrations, the registry surfaces the one that already exists. Instead of a security review that starts after a server is already in production, the registry makes compliance a prerequisite for discoverability.

Layer 2: The Runtime — isolated execution

The second problem is execution. An MCP server is a piece of software that has access to your internal systems. It should be treated with the same rigor as any other privileged workload. That means that an MCP server needs to be deployed with defined resource limits, isolated from adjacent workloads, observable, and replaceable.

Kubernetes provides the orchestration substrate. A Kubernetes operator for MCP extends this with MCP-specific lifecycle management. Deploy MCP servers as first-class Kubernetes resources, with the same deployment, scaling, and upgrade patterns you already use for your other workloads.

The isolation story matters particularly to security teams. Each MCP server runs in its own process boundary, with network access scoped to only what it needs. Filesystem access is constrained. Secrets are injected at runtime through your existing secrets management, not hardcoded in plain text config files. If a server is compromised, the blast radius is contained; it cannot reach other servers or infrastructure components without explicit policy allowance.

For organizations with stringent isolation requirements, stronger-than-container boundaries are possible. MicroVM-based isolation provides hardware-level separation between MCP server processes, addressing concerns that standard container isolation is insufficient for workloads with access to sensitive data systems.

This layer also makes the upgrade story tractable. When an MCP server needs a patch it can be rolled out using the same deployment patterns as any other Kubernetes workload, with version tracking and the ability to roll back.

Layer 3: The Gateway — unified enforcement

The third problem is access control. In most early MCP implementations, authorization is handled at the server level; each server makes its own decisions about who can call which tools. This means policy is fragmented across dozens of servers, there is no single audit log, and adding a new policy requires touching every server individually.

A gateway layer solves this by providing a single enforcement point for all MCP traffic. Every tool call from every agent, in every client, passes through the gateway before it reaches a server. This gives you:

Unified authentication
Developers authenticate once through your existing identity provider, whether that's Okta, Entra ID, or another and that identity flows through to every downstream system. The agent acts on behalf of the user, and the downstream system logs the user, not a service account.

Granular authorization
Policy can be defined at multiple levels: which users can access which servers, which tools within a server each user or role can invoke, and what claims are passed downstream. A read-only analyst should not be able to trigger write operations. A contractor should not have access to production data sources. The gateway enforces this without requiring changes to individual servers.

Tool filtering
Not every tool on every server needs to be available to every agent in every context. The gateway can present different views of the same server to different users or roles, exposing only the tools relevant to a given workflow, which reduces both context overhead and attack surface.

Audit logging
Every tool call produces a structured log entry that includes the identity of the caller, the tool invoked, the parameters passed, the response received, and a timestamp. These logs flow into your existing SIEM or observability platform, providing the baseline of identity and tool usage that security and compliance teams require.

Layer 4: The Management Plane — operational visibility

The fourth problem is visibility. Platform teams need to know what's running, who's using it, and whether it's healthy. Security teams need to know whether policy is being enforced and where the gaps are. Developers need to know what's available and how to get access.

The management plane surfaces this across the organization. It provides a self-service interface through which developers can discover approved servers, request access, and understand what each tool does. It provides an operational view for platform teams that covers deployment status, health metrics, usage patterns, and version currency across the server estate. And it provides a governance view for security teams: policy status, audit log access, and visibility into any servers operating outside the governed perimeter.

This layer also addresses the scale problem. Organizations that start with five MCP servers typically have fifty within eighteen months, as more teams adopt the pattern and more enterprise SaaS vendors ship their own MCP implementations. The management plane is what makes the difference between a system that scales and one that collapses under its own complexity.

Getting from here to there: a phased approach

The architecture above is the destination. The path there is incremental. Most enterprises are somewhere in the middle, and the goal is to establish the right patterns early enough that they survive contact with scale.

The following phases are not rigid; they reflect the sequence that tends to work in practice, based on where enterprises are starting and what security teams need to see before they will approve a broader rollout.

Phase 1: Establish the registry and stop the sprawl

The first priority is visibility. Before you can govern what's running, you need to know what exists. This phase focuses on standing up a governed registry and establishing the expectation that approved MCP servers come from there (and only there).

The outputs of Phase 1 are a:

- Curated catalog of approved servers your organization has vetted
- Clear process for teams to submit servers for review and promotion
- Baseline policy that developer tooling points to the registry for MCP configuration rather than sourcing servers ad hoc

This phase often has an immediate security dividend. In most organizations, the act of standing up the registry surfaces servers that the security team didn't know existed. The goal is not to make ungoverned usage impossible, but to make the governed path preferable.

Phase 2: Deploy governed MCP servers on Kubernetes

With the registry established, Phase 2 focuses on the runtime: deploying approved MCP servers into your existing Kubernetes infrastructure with proper isolation, secrets management, and observability.

The outputs of Phase 2 are:

- MCP servers running as first-class Kubernetes workloads, with the same deployment rigor as your other production services
- Telemetry flowing into your existing observability stack
- Reference architecture your teams can follow when building new servers

This is the phase where the Kubernetes operator becomes central. Rather than managing MCP server deployments as bespoke Kubernetes configurations, the operator provides an abstraction layer that is purpose-built for MCP lifecycle management, handling deployment patterns, health checks, upgrades, and the MCP-specific configuration that generic Kubernetes tooling doesn't know about.

The practical milestone that marks the end of this phase: a developer can run an approved MCP server in a shared, governed environment without needing to manage their own Docker or local runtime, and the platform team has full visibility into what's running and how it's performing.

Phase 3: Connect identity and enforce access policy

Phase 3 is where the governance story becomes complete and where the security team moves from cautious observer to active supporter of broader rollout.

This phase introduces the gateway layer, connecting your existing identity provider to MCP server access and establishing per-user, per-tool authorization policy. The key technical challenge here is identity passthrough, ensuring that when an agent acts on behalf of a user, downstream systems log the actual human user, not a service principal.

The outputs of Phase 3 are:

- Single endpoint through which all MCP traffic flows
- Authorization policy expressed in your preferred policy language (Cedar, OPA, or equivalent)
- Audit logs that show exactly who called what, when
- Ability to walk into a security review with credible, documented answers to the questions that typically block AI rollout.

This is also the phase where token consumption becomes manageable. A naive implementation exposes every tool on every server to every agent, which both bloats context windows and degrades agent accuracy. The gateway enables tool filtering, so that agents see only the tools relevant to their task, reducing both cost and error rate.

Phase 4: Scale and extend

With the foundation in place, Phase 4 is about extending the platform to additional teams, use cases, and systems. This typically means onboarding servers for additional enterprise systems (ERP, ITSM, data platforms), expanding developer access beyond the initial engineering cohort, and beginning to support non-developer use cases (e.g. analysts, product managers, operations teams) who need agent capabilities but are not working in an IDE.

The decisions made in Phases 1–3 compound here. An organization that established governed patterns early can extend to new teams and systems quickly, because the framework already exists.

Phase 4 is also where more sophisticated capabilities become relevant: composite tools that chain multiple API calls into a single agent-invocable action, context optimization that keeps tool metadata from overwhelming model context windows, and evaluation frameworks that measure tool call accuracy at scale. These are the capabilities that distinguish production-grade MCP infrastructure from proof-of-concept implementations.

What you get: the case for your security team

The architecture described in this guide is, in part, a document you can hand to your security team. It answers the questions that typically block AI rollout at the enterprise level.

"How do we know what's running?" The registry is the authoritative catalog. Nothing runs that isn't in it, and everything in it has been vetted.

"Who has access to what?" The gateway enforces authorization policy centrally. Access is tied to identity, not to knowing a URL or having a configuration file.

"How do we audit this?" Every tool call produces a structured log entry. Those logs flow into your existing SIEM. You can reconstruct exactly what happened in any agent interaction.

"What's the blast radius if something goes wrong?" Each server runs in an isolated runtime with scoped network and filesystem access. Compromise of one server does not imply compromise of adjacent systems.

"What happens when we want to change models or frameworks?"
The platform is model-agnostic and framework-agnostic. Changing your underlying LLM or agent framework does not require rebuilding the governance infrastructure. The control plane is neutral.

That last point deserves emphasis. One of the structural risks in AI infrastructure is lock-in: building governance, identity, and observability into a model provider's platform means that switching models (which you will want to do as the landscape evolves) requires rearchitecturing everything underneath. A Kubernetes-native, vendor-neutral platform avoids this. The orchestration layer does not belong to your model provider. You can swap models, swap frameworks, and swap providers without dismantling the infrastructure your teams depend on.

Starting the conversation

If you have Kubernetes in production, you have most of what you need to run AI agents with enterprise-grade governance. What Stacklok's Kubernetes operator adds is the MCP-specific layer, including the registry, the runtime, the gateway, and the management plane, built on the cloud-native infrastructure you already trust.

Stacklok is built on the open source project, ToolHive, which is Apache 2.0 licensed. You can evaluate it, run it in a pilot, and establish your first governed MCP deployment without a procurement conversation. Stacklok's enterprise offering adds hardened builds, SLAs, and the self-service management capabilities that scale beyond the initial engineering team.

The organizations getting ahead of this problem are not waiting for the governance story to be perfect before they start. They are establishing patterns now while the footprint is still small, and while the security team can be involved from the beginning.

Regardless of where you are in the journey, the Stacklok team is available as a resource. You can learn more and get in touch with us at stacklok.com, or get started with ToolHive directly at github.com/stacklok/toolhive.