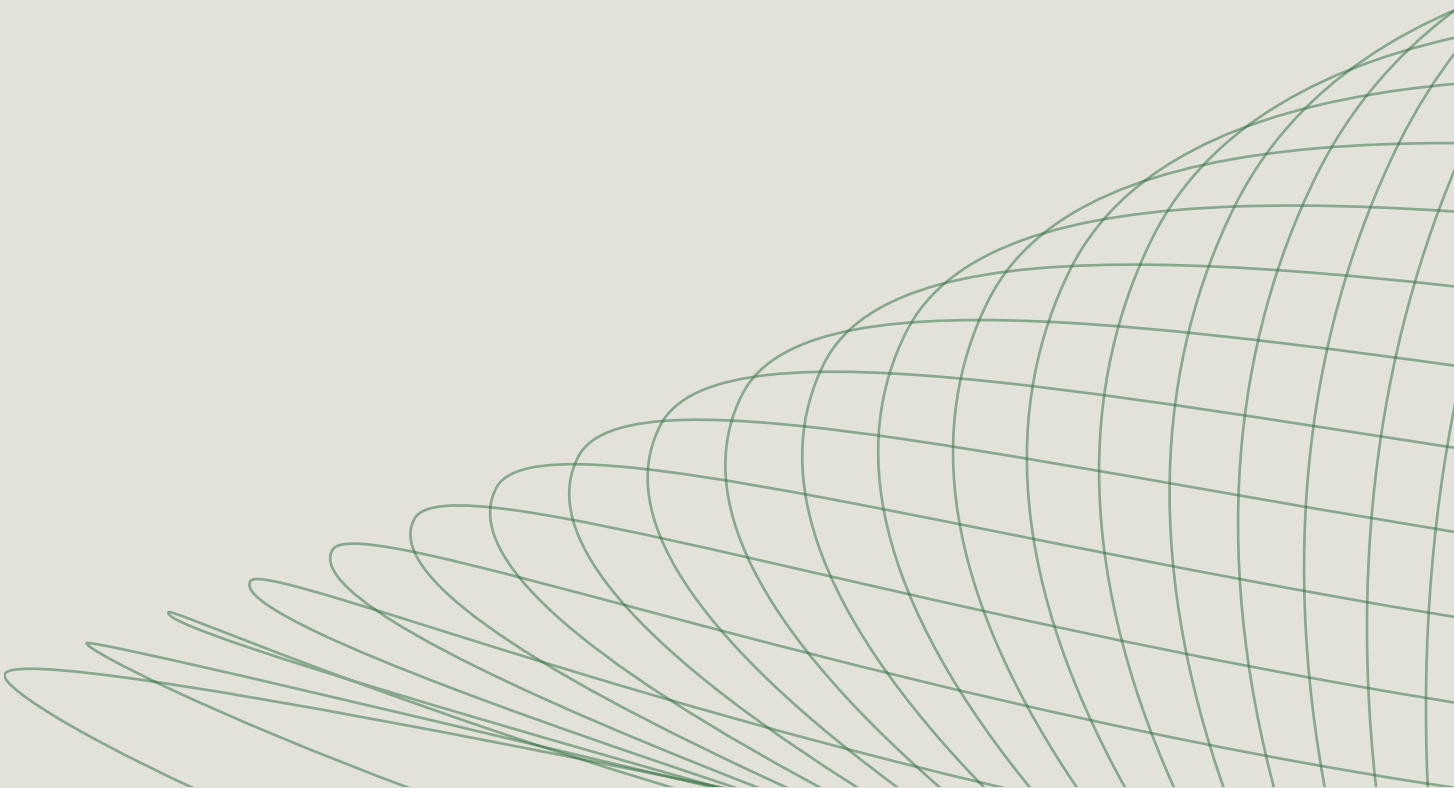
STACKLOK

The MCP Platform Buyer's Guide for Platform Teams



Contents

Over the past few months, we've worked closely with more than forty platform teams at large enterprises. This buyer's guide is based on the critical capabilities they required from an MCP platform; we've summarized their priorities and questions to help you leave no stone unturned as you identify the right path forward for your enterprise.

Why Platform Teams Need to Own MCP Now

Model Context Protocol (MCP) is not a trend you can wait out. It is rapidly becoming the integration layer between AI agents and enterprise systems. More than 40% of enterprises are already using some MCP servers in production, multiplying the value of their AI agents with the ability to access and act on GitHub, Jira, Confluence, Databricks, Salesforce, SAP, internal APIs, and more.

However, there are risks when any technology is rapidly adopted. A misconfigured or compromised MCP server can mean a language model has read access to your data warehouse, write access to your ticketing system, or an unaudited connection into a system that handles PII or regulated financial data. The challenge for platform engineers is that you cannot solve this by saying no to MCP. Developers will route around controls they find too restrictive and shadow MCP usage will emerge. The goal is to build governance infrastructure that's rigorous enough to satisfy the security team and lightweight enough that developers actually adopt it.

That window between organic MCP sprawl and the unmanageable security and operational liability is exactly where your leverage is. Platform teams that establish the right patterns now will define how their organizations use agentic AI for the next several years. Those who wait will spend that time in remediation mode.

This guide gives you the framework to evaluate MCP platforms seriously, ask the right questions of vendors, and build internal consensus around a set of requirements that will serve you both today and as your MCP footprint scales from a handful of servers to dozens or hundreds.

The Reality of Where Most Organizations Are Today

If your organization looks anything like the enterprises actively evaluating MCP platforms right now, the picture is probably recognizable:

Sprawl has already started

Developers are running MCP servers locally from their laptops, pulling community-sourced servers via npx, and connecting AI agents to internal systems without going through any formal approval process. Some of this happened before any governance rules existed. Some of it happened because the official path was too slow.

Your MCP client landscape is inconsistent

Codex may have MCP turned on with no central controls. VS Code may have it turned off entirely. Cursor users are doing whatever they want. There is no unified enforcement point, which means you have no coherent security posture, just a collection of individual decisions made at the developer level.

Authentication is improvised

Most early MCP implementations rely on API keys baked into configurations, shared service principals, or personal access tokens that have no rotation policy and leave no meaningful audit trail. When a downstream system like Databricks logs a tool call, it records the service account, not the engineer who triggered it.

Security teams are uncomfortable, and that discomfort is blocking adoption

The right response to security team concern is to give them the audit logs, the access controls, the identity passthrough, and the policy framework they need to say yes. Without that, every new MCP server requires a bespoke security review, and adoption stays slow.

Governance patterns are being invented ad hoc

Your team is defining reference architectures as you go, without much prior art to draw from. The MCP spec itself is still evolving. Patterns you establish today will need to hold up as the ecosystem matures and as your server count scales from five to fifty to five hundred.

The Core Challenges Platform Teams Must Solve

Before evaluating any platform, it is worth being explicit about the problem surface. There are five categories of challenge that consistently emerge as organizations try to scale MCP from experimentation to production.

Authentication and identity passthrough is the hardest technical problem, and it comes up in virtually every serious MCP conversation. The core issue: when an AI agent calls a tool through an MCP server, which identity does the downstream system see? Today, most implementations log a service principal.

What regulated environments and security teams actually require is end-to-end identity passthrough, meaning the actual user's identity flows through the MCP layer to the downstream system, with scoped, short-lived tokens and no stored credentials. Getting this right requires proper OAuth token exchange flows, on-behalf-of (OBO) token support, and integration with enterprise IdPs like Okta and Entra ID.

Governance and policy enforcement at scale is the second major challenge. It is relatively straightforward to govern five MCP servers. It is a fundamentally different problem to govern fifty, spanning multiple teams, business units, deployment environments, and use cases. You need a policy model that can express fine-grained controls, not just "who can access this server" but "what tools can this user invoke, under what conditions, and with what data constraints." You need those policies to be declarative, version-controlled, and auditable. And you need enforcement that happens at runtime, not just at provisioning time.

Discoverability without shadow IT is a tension that every platform team feels. The harder you make it to consume MCP servers through official channels, the more developers will route around those channels. But without a curated registry and an approval workflow, you have no visibility into what's running, and no way to enforce supply chain security. The right answer is a registry that is easy enough to use that it becomes the default, not a compliance checkbox.

Operational maturity and observability are non-negotiable for production workloads. You need to know what MCP servers are running, who is calling them, what tools are being invoked, and whether those invocations are succeeding or failing. You need that telemetry to flow into your existing observability stack (Datadog, Splunk, Grafana, etc.) without custom instrumentation work. And you need audit logs that are retained, queryable, and exportable for compliance purposes.

Deployment flexibility across heterogeneous environments matters more than most vendors acknowledge. Your organization probably has Kubernetes clusters in multiple clouds, on-premises infrastructure, air-gapped environments, and a mix of local developer tooling. A platform that only works in one deployment model is a constraint you will eventually need to work around.

The Capability Checklist: Requirements for Your MCP Platform

Use the following framework when evaluating vendors. These requirements reflect the real challenges enterprises are working through as they scale MCP to production. Where capabilities are table stakes that any credible platform should offer, they are noted as such. Where they represent a higher bar that separates serious enterprise platforms from lighter-weight alternatives, that distinction is called out.

Section 1: Runtime and Deployment

Capability	Why it Matters	What to Require
Kubernetes-native operator	MCP servers need to be managed like any other workload with CRDs, Helm charts, operator-based lifecycle management, and native scheduling. Anything less creates an operational island.	Require a native Kubernetes operator with CRD-based server lifecycle management. Verify support for your specific distribution (EKS, AKS, GKE, OpenShift).
Container isolation per server	Each MCP server should run in its own isolated container with minimal default permissions. Shared runtimes create blast radius risk.	Require per-server container isolation with configurable network egress and filesystem access controls. Ask specifically about support for hardware-isolated microVM runtimes (Kata Containers, Firecracker) for high-sensitivity environments.
Local development support with policy enforcement	Developers will want to run servers locally during development. That's fine, but local execution should still enforce registry policies and emit telemetry, not bypass governance.	Require that local execution modes remain under central policy control and still report usage telemetry to your observability stack.
Air-gapped and on-premises deployment	Many regulated environments cannot send data to external SaaS platforms. This is a hard requirement for financial services, defense, and certain healthcare environments.	Confirm the platform is fully self-hostable with no required SaaS components or external callbacks. Verify this is not a roadmap item; it must be available today.
Multi-tenant, multi-namespace support	Platform teams support multiple development teams, each with different access requirements. The MCP platform needs to reflect that organizational structure.	Require namespace-level isolation with RBAC controls that allow different teams to have different server catalogs and access policies without admin intervention for each change.
High availability and failover	Production MCP infrastructure cannot be a single point of failure.	Require multi-replica deployment support and documented failover behavior. Ask about circuit breakers at the gateway layer.
Cross-platform binary support	Your engineering population uses Linux, macOS, and Windows. If your organization uses ARM-based hardware (including Apple Silicon or ARM server architectures), binary support matters.	Require pre-built binaries for all major OS and architecture combinations, including ARM-based Linux.

Section 2: Authentication and Identity

This is the category where the gap between credible enterprise platforms and lightweight alternatives is widest. Treat any vendor that cannot give detailed, concrete answers here with significant skepticism.

Capability	Why it Matters	What to Require
Enterprise IdP integration (Okta, Entra ID, Google, Ping)	Developers should authenticate through your existing identity provider, not create new credentials for MCP access.	Require native OIDC/OAuth SSO with your IdP, with no stored API keys or personal access tokens in client configurations. If you're using EntraID, make sure you understand Entra's dynamic client registration limitations.
Per-user identity passthrough to downstream systems	This is the audit trail problem. When a developer's AI agent calls a Databricks tool, Databricks should log that developer's identity, not a shared service principal.	Require token exchange flows (OAuth on-behalf-of / token exchange RFC 8693) that preserve and pass the end user's identity to downstream systems. Ask for a specific architecture walkthrough of how this works end-to-end.
Short-lived, scoped tokens with automatic rotation	Static credentials are a security liability. Each tool call should operate under a scoped, short-lived token appropriate to the action being performed.	Require that the platform issues short-lived tokens per session or per request, with no requirement for developers to manage credential rotation manually.
Non-human identity support for agents	Agentic workflows operate without a human in the loop. The platform needs a story for workload identity, not just human SSO.	Require support for workload identity patterns including SPIFFE/SPIRE-based JWT authentication for non-human agents running in automated pipelines.
Read-only and granular operation-level scoping	Legal, compliance, and security teams will frequently need to allow tool access while restricting destructive operations. "Read Jira tickets but do not update them" is a real requirement.	Require the ability to scope tool permissions by operation type (read, write, delete) at the policy level (not just by server access). Verify this is enforced at the gateway, not just configured per-client.
Support for both interactive and non-interactive auth flows	Human developers using VS Code need a browser-based OAuth flow. Automated agents in CI/CD pipelines need a client credentials flow. The platform needs to support both.	Require support for both interactive OAuth (browser redirect) and client credentials flows within the same policy framework.

Section 3: Governance and Policy

Capability	Why it Matters	What to Require
Curated server registry with approval workflows	Without a registry, you have no visibility into what MCP servers are running across the organization. With a registry that is painful to use, developers route around it.	Require a registry that supports metadata (owner, security review date, data classification, approved client list) and a vetting workflow that integrates with your existing approval processes.
Supply chain security for MCP servers	MCP servers sourced from the internet (npm packages, GitHub repos) represent a real supply chain risk. Several enterprises have already experienced supply chain incidents through AI developer tooling.	Require CVE scanning and SLSA attestation for MCP servers in the registry. Ask whether the vendor maintains hardened versions of common community MCP servers (GitHub, Atlassian, etc.) with ongoing security patch commitments.
Granular, declarative policy engine	Access control at the server level is insufficient for production. You need tool-level, claim-based, and role-based policies that are readable, version-controlled, and testable.	Require a policy engine that supports RBAC, ABAC, and claim-based authorization at both the server and tool invocation level. Policies should be declarative (code), not UI-only configurations. Verify compatibility with your existing policy framework (Cedar, OPA/Rego).
Client-side enforcement hooks	Governance that only exists at the server layer can be bypassed by a developer who knows what they're doing. Enforcement at the client layer (inside the IDE) closes that gap.	Require hooks-based enforcement for your approved IDE clients (VS Code, Cursor, Windsurf, Claude Code). Ask specifically which clients are supported today, and which have gaps..
Policy changes without redeployment	Governance needs to respond to incidents in real time. If changing a policy requires a full redeployment cycle, your incident response capability is compromised.	Require that policy changes take effect at runtime without server or gateway redeployment.
Tool filtering and virtual MCP server composition	As your MCP footprint grows, individual MCP servers expose too many tools for effective agent use. A server exposing 1,200 AWS tools to every developer is both a governance risk and a context window problem.	Require the ability to define virtual MCP servers that can be tuned per team, role, or use case. This is a meaningful differentiator; most lightweight platforms do not support it.

Section 4: Observability and Audit

Capability	Why it Matters	What to Require
OpenTelemetry-native telemetry	You already have an observability stack. You should not have to run a parallel one for MCP.	Require OTLP-compatible traces, metrics, and logs following official OpenTelemetry MCP semantic conventions. Verify that telemetry exports to your existing backend (Datadog, Splunk, Grafana, Dynatrace, New Relic, Honeycomb, Prometheus, etc.) without custom instrumentation.
Tool call tracing	Knowing that a server was invoked is not enough. You need to know which tool was called, with what arguments, by which user, at what time, and whether it succeeded.	Require per-tool-invocation trace data including identity, arguments, latency, and outcome. Verify this applies to both local and remote server deployments.
SIEM integration	Security operations teams need MCP audit data in the same place they monitor everything else.	Require log export in formats compatible with your SIEM platform. Ask specifically about Splunk, Elastic, and Sentinel integrations.
Usage attribution for cost chargeback	As MCP usage scales, finance and engineering leadership will want to attribute usage and cost to individual developers, teams, or projects.	Require telemetry granular enough to support chargeback models. Tool call counts, token consumption estimates, and user-level attribution should all be available in the telemetry stream.
Audit log retention controls	Regulated industries have specific retention requirements. Financial services organizations commonly require 7+ year retention.	Require configurable retention periods and confirm that logs can be exported to long-term storage under your control. Do not accept a platform where retention is determined solely by the vendor's infrastructure.
Baseline and anomaly visibility	A SIEM integration is more useful when you have a sense of what normal looks like. The platform should help you establish usage baselines and surface deviations.	Require sufficient telemetry granularity to establish per-user, per-team, and per-server usage baselines over time. This is the foundation for detecting compromised credentials or unexpected agent behavior.

Section 5: Developer Experience and Adoption

Governance infrastructure that developers resent will be circumvented. The best enterprise MCP platforms make the governed path the easy path.

Capability	Why it Matters	What to Require
Self-service discovery portal	Developers need to find approved MCP servers without filing a ticket. A curated, searchable catalog is the difference between a registry that gets adopted and one that gets ignored.	Require a self-service portal that developers can access without admin assistance. The portal should support both a human-readable UI and a machine-readable API for programmatic integration.
Automatic client configuration	Requiring developers to manually configure each IDE client is a friction point that slows adoption and creates inconsistency.	Require automatic client configuration: when a developer gains access to a server, their IDE should be configured automatically, not through a manual YAML edit.
Context optimization	As MCP server counts grow, context window bloat becomes a real problem. Exposing 50 tools to a model that needs 3 produces worse results and burns tokens unnecessarily.	Require a gateway-level context optimizer that can dynamically surface relevant tools based on the task at hand, rather than flooding the model with every available tool on every request. This is a differentiating capability; most lightweight platforms do not offer it.
Composite tool and workflow support	Many real enterprise workflows require sequencing multiple tool calls in a defined order with transaction semantics. Leaving this to the LLM to figure out on its own produces inconsistent results.	Require the ability to define declarative composite tools that chain multiple underlying tool calls into a single, deterministic workflow. Ask for a specific demo of this capability with a realistic multi-system workflow.
Support for both local and remote server models	Some servers should run locally on the developer's machine. Others should run centrally in Kubernetes. Both models are valid. The platform should govern both the same way.	Require that local and remote server deployments are subject to the same registry policies, the same telemetry requirements, and the same identity controls. The deployment model should not determine the governance model.

Section 6: Commercial and Vendor Considerations

Governance infrastructure that developers resent will be circumvented. The best enterprise MCP platforms make the governed path the easy path.

Capability	Why it Matters	What to Require
Open source core with commercial enterprise layer	Open source lets you evaluate, prototype, and build internal confidence without a procurement conversation. It also signals a vendor philosophy that enterprise platform teams tend to trust.	Require an Apache 2.0 (or equivalent permissive) licensed core that includes the runtime, registry, and gateway. Enterprise tiers should add hardening, support SLAs, and convenience features.
Production support SLAs	When MCP infrastructure is business-critical, you need a vendor who will pick up the phone.	Require defined SLA tiers with escalation paths. Confirm support coverage matches your operational hours and that critical incident response times are contractually defined.
Outcomes-oriented engagement model	Delivering software and walking away is not adequate for infrastructure this new. The best engagements involve working alongside your team to implement, validate, and iterate.	Ask specifically about the vendor's field engineering model. Is there a dedicated deployment engineer for the initial rollout? What does the post-sale engagement look like? A vendor who can embed with your team is worth significantly more than one who hands you documentation.
Modular architecture (use what you need, bring what you have)	Your organization has already invested in identity systems, policy engines, registries, and portals. A platform that requires you to replace all of that is a much harder internal sell than one that augments your existing investments.	Require explicit confirmation of which components are swappable or optional. Can you use your existing OPA/Rego policies instead of Cedar? Can the registry coexist with your existing artifact management platform? Every time a vendor confirms modularity, it de-risks the procurement conversation.

Putting It Together: Scoring Your Evaluation

Not every capability on this list carries equal weight for every organization. Use the following framework to prioritize based on your current situation.

If your primary concern is security team buy-in

Lead with authentication/identity passthrough, client-side enforcement hooks, audit log retention, and supply chain security. These are the capabilities that allow you to walk into a security review with credible answers. The platform team's job is often to get security comfortable enough to unlock broader adoption.

If your primary concern is developer adoption

Lead with the self-service portal, automatic client configuration, and the open source evaluation path. Developers who can evaluate and start using a platform without a procurement conversation become internal advocates. Developers who hit friction become workarounds.

If your primary concern is scaling from your current footprint

Lead with the Kubernetes operator, multi-tenant support, virtual MCP server composition, and context optimization. The patterns you establish for five servers need to hold at fifty. The platforms that seem adequate today often cannot make that transition gracefully.

A Note on What the MCP Ecosystem Looks Like Today

The MCP platform landscape is early. Several vendors have entered the space, and it is worth being clear-eyed about where they focus and where they fall short when evaluated against enterprise requirements.

Lightweight gateway tools can proxy MCP traffic and provide basic routing, but they typically lack the runtime management, supply chain security, and identity depth that production enterprise deployments require. They are useful for prototyping. They are not adequate for governing a multi-team MCP estate in a regulated environment.

Developer-tooling platforms that have added MCP support often treat it as a feature rather than a first-class primitive. Their governance models are designed for developer convenience, not enterprise security posture.

Cloud provider offerings address deployment but typically lack vendor-neutral flexibility, deep identity passthrough, and the ability to govern MCP usage across a hybrid local-plus-remote server landscape.

The right platform will be purpose-built to solve the enterprise MCP problem with a Kubernetes-native operator, a full identity and policy stack, supply chain security, and an open source core that lets you evaluate without asking for permission. It treats governance not as a constraint on developer productivity but as the foundation that makes broader adoption possible.

The questions in this guide will help you tell the difference.