



Model Context Protocol strategies on AWS

AWS Prescriptive Guidance



AWS Prescriptive Guidance: Model Context Protocol strategies on AWS

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction	1
Intended audience	2
Objectives	2
What is MCP?	4
Understanding tools	4
When to use MCP	7
MCP tool design strategy	10
Tool scope	10
Granular	11
Coarse-grained	12
Best practices for MCP tool scoping	13
Tool definitions	14
Tool specification approach	14
Docstring approach	15
Best practices for MCP tool definitions	16
Tool discovery	16
Static definition	16
Dynamic discovery	17
Search function	18
Best practices for MCP tool discovery	18
Tool organization	18
Best practices for MPC tool organization	20
MCP hosting strategy	21
Hosting approaches	21
Local hosting	21
Remote hosting	22
MCP gateway	23
Best practices for hosting MCP servers	24
MCP governance strategy	25
Authentication and authorization	25
Best practices for MCP authentication and authorization	26
Controlling load	27
Best practices for controlling load	28
Operational metrics	28

Contributors **29**

Authoring 29

Reviewing 29

Technical writing 29

Document history **30**

Model Context Protocol strategies on AWS

Amazon Web Services ([contributors](#))

March 2026 ([document history](#))

This guide can help you develop and implement Model Context Protocol (MCP) strategies across your organization to support your agentic AI journey. As agents and language models become increasingly central to business operations, establishing an MCP strategy is critical for successful agentic solutions.

This guide explores three foundational pillars for building an MCP strategy: MCP tool design, MCP server hosting, and MCP governance. By addressing these interconnected components, organizations can create scalable, secure, and effective systems for managing model context across their AI implementations. This guidance provides actionable insights and strategic guidance for organizations at any stage of an organization's AI journey, from initial experimentation to full-scale production deployments. This helps them develop tailored MCP solutions that align with their specific needs and objectives.

These best practices are derived from real-world implementations of organizations deploying MCP at enterprise scale, an analysis of current MCP specification standards, and lessons learned from custom Large Language Model (LLM) applications in production.

AI systems use increasingly sophisticated and robust LLMs in a wide variety of use cases. LLMs excel at understanding natural language, generating human-like responses, and reasoning over complex information. However, to transform LLMs from conversational interfaces into systems that can autonomously accomplish complex tasks, organizations are adopting *agentic AI architectures*, AI systems that can perceive their environment, reason about goals, make autonomous decisions, orchestrate across multiple steps, and take actions to achieve objectives on behalf of users. This agentic approach helps organizations build AI systems that can understand user intent through natural language, autonomously coordinate across multiple data sources and tools, and deliver personalized experiences at a scale that was not possible with traditional request-response patterns. To make these agents more capable, organizations need to provide access to their existing tools and data to enrich the agent's contextual understanding and allow it to act on a user's behalf.

[MCP](#) provides a standardized protocol for AI-tool integration, enabling consistent communication between agents and external resources. While MCP itself defines the communication standard,

implementing it effectively requires careful consideration of architectural patterns, security models, operational practices, and performance optimization strategies to achieve scalable, secure, and maintainable solutions.

This guide synthesizes lessons learned from enterprise MCP deployments, providing actionable recommendations that align with the [AWS Well-Architected Framework](#). It covers strategies for MCP tool design, MCP server hosting, and MCP governance, which are essential in building your own MCP solutions. The recommendations in this guide map to the following five pillars of the AWS Well-Architected Framework:

- **Security** – Token isolation, scoped-down credentials, separate read/write authorization
- **Operational Excellence** – Tool selection accuracy metrics, golden datasets for regression testing
- **Reliability** – Per-user and per-tool rate limiting, load shedding
- **Performance efficiency** – Workflow-scoped tools, tool filtering, semantic search to reduce context window usage
- **Cost optimization** – Reusable MCP servers across teams, reduced per-request token costs through tool filtering

Intended audience

This guide is intended for architects, developers, and technology leaders who are implementing agentic AI solutions in their organizations. To understand the concepts in this guide, you should understand how LLMs work and have foundational knowledge about MCP, tools, and prompt engineering.

Objectives

Building Agentic AI systems that are production-ready means solving for governance, optimization, and security together to support your organization's policies. The below explains how this guide addresses these objectives:

- **Governance** – Without centralized governance, you cannot answer audit questions about your AI workloads, including which agents accessed which data, with what permissions, and when. You also cannot enforce versioning. The [MCP hosting strategy](#) section of this guide explains how users could be running outdated local MCP servers with known vulnerabilities due to lack of systematic enforcements.

For regulated industries, governance is critical. Auditors want to see policy enforcement and tool usage tracking across all agents from a single pane. MCP governance provides that.

By following the recommendations in this guide, you can improve task accuracy by 28-32% in peer-reviewed benchmarks. For more information, see [MARCO: Multi-Agent Real-time Chat Orchestration](#) (ACL Anthology website). Governance is not just about compliance; it also improves your agentic AI system performance.

- **Optimization** – Your teams might build the same integrations more than once. For example, when five different teams write their own database query script for their AI application to communicate with their databases, that's five times the development cost and five sets of bugs list to maintain. MCP lets you build it once and share it across the entire engineering community. The savings compound as your agent count grows.

There is also a per-request cost problem that most teams don't notice at first. Every tool definition consumes context window tokens. At 20 tools, you're spending 5,000-10,000 tokens per invocation on descriptions alone, alongside the user inquiries. This increases latency and LLM inference costs and degrades accuracy as the model struggles to pick the right tool from the list of available tools.

Agents that use structured tool wrappers are approximately three times more accurate on database tasks than agents that access APIs directly (for more information, see [Middleware for LLMs: Tools Are Instrumental for Language Agents in Complex Environments](#)). How you design and present tools to an AI model is important. This guide recommends giving tools clear schemas, scoping them to actual workflows instead of raw endpoints, and limiting information in the context window. The [MCP tool design strategy](#) section of this guide dives deep into these aspects.

- **Security and compliance** – Imagine an agentic AI system that hallucinates a cleanup step and tries to delete a production database. If the agent inherited the user's full admin credentials, the deletion might go through. With token isolation and scoped-down credentials that only grant read and create access, it fails safely.

Regulated workflows sharpen this further. The guide provides examples (healthcare pipelines that require HIPAA validation and personally identifiable information anonymization before processing patient data). Embedding such logic in MCP tools means compliance happens deterministically every time.

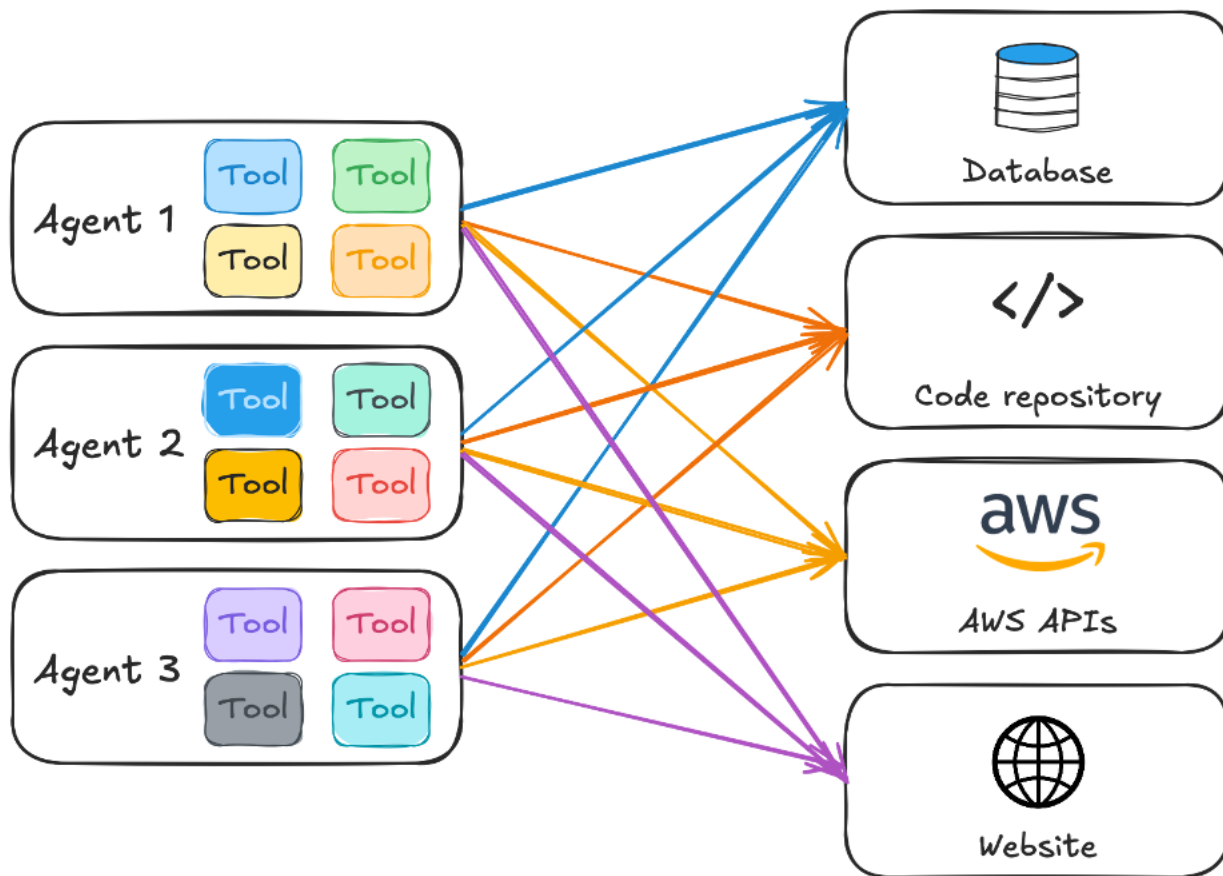
What is MCP?

LLMs work by predicting an answer to a prompt based on their training data. This means that the LLM can only provide answers about data and events it has already seen. Methods like Retrieval Augmented Generation (RAG) and knowledge bases allow you to include contextual data. However, if you were to ask an LLM what tomorrow's weather forecast is going to be or how many customers are in your database, it would likely hallucinate or not be able to provide an answer because these fall outside of the LLM's pre-trained knowledge. To be able to answer these kinds of questions, an agent needs access to external capabilities, data, and APIs outside of the LLM's native context.

Understanding tools

We can give the LLM access to additional systems and context through tools. *Tools* are functions given to the LLM to achieve a clear objective. A tool could call an API, query a database, perform calculator operations, operate a code sandbox, perform a web search, and even invoke another AI system or agent-as-a-tool. Each tool should include a description that tells the LLM what the tool does, when to use it, and what parameters it accepts. This enables the LLM to make nuanced decisions about which tool or combination of tools to invoke based on the user's input. The LLM is told about what tools are available to the agent, allowing it to generate responses that instruct the agent to invoke the tool. For example, when you ask the LLM how many customers are in your database, the LLM will send a response back to the agent requesting to run the `query_database` tool with specific input parameters. The LLM determines which tool to invoke and the inputs for the tool call. The agent then executes the tool, which converts the natural language input into a syntactically correct function call and runs the query. The agent invokes the tool or tools based on the instruction from the LLM, and those results are passed back to the LLM. This takes advantage of the LLM's ability to reason over text-based input and select the appropriate tools for the job.

The following image shows how each agent manages its own tool set for each target.



Scaling tool access can present challenges for agentic AI solutions:

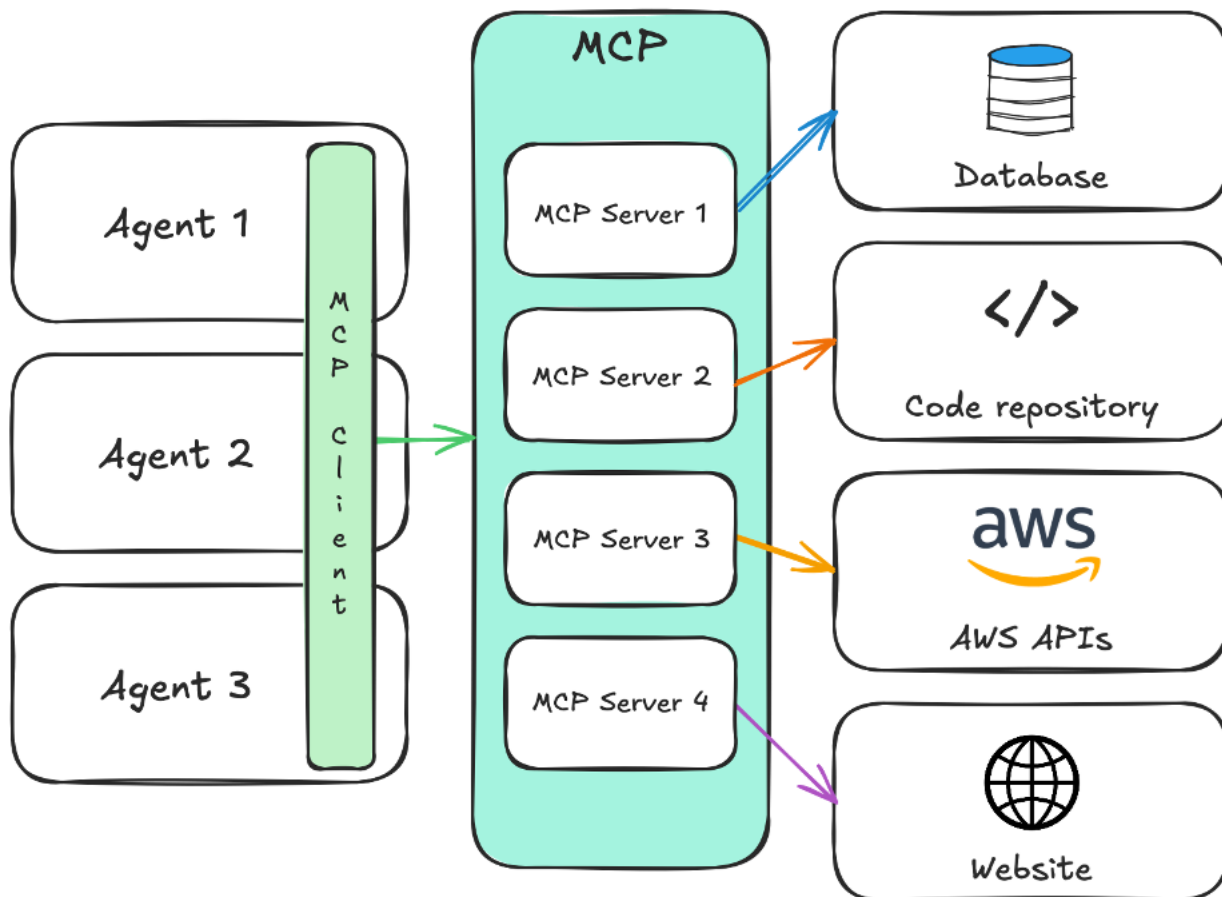
- If every developer is creating their own tool for the same external capabilities, there is a lot of duplicated effort and non-standardized ways of interacting with these external capabilities. This produces inconsistent implementations across your agents. While you could solve that problem by developing standard tools in libraries and distributing them, this lacks centralized governance. This makes it difficult to enforce security policies, track tool usage, manage versioning across teams, or ensure compliance with organizational standards. Additionally, when you embed tools directly with the agent, you must redeploy your agent every time a new tool is created or an existing one is updated.
- Providing tools to an LLM consumes its context window. *The context window* is the number of *tokens* (units of text that LLMs process—typically representing words, parts of words, or punctuation) that a model can consider at any one time. LLMs have a context window limits. Tools and their documentation consume that finite context window along with system prompts and user prompts. As the context window fills, LLMs can experience degraded performance due to multiple factors: difficulty in identifying relevant information, increased processing complexity, and reduced capacity for reasoning. The challenge is compounded when tool

definitions, system prompts, and conversation history compete for limited context window space because they are provided on each LLM invocation.

Thus, the number of tools and how they are documented have direct impact on the LLM's performance, such as response time, and accuracy.

MCP establishes a universal standard for connecting agents to external capabilities. It is commonly referred to as the "USB-C for AI applications." Instead of registering tools directly with agents, MCP servers act as an intermediary for hosting tools that are discovered and invoked over [JSON-RPC 2.0](#). Instead of adding tens or hundreds of different tools to your agent and maintaining them over time, MCP allows you to register MCP servers that encapsulate the tools that your agent can access. This approach standardizes how tools are packaged, presented, and invoked. This can help address the scale and governance challenges of tool usage inside your agents. It also decouples agent development and operations from the tools that it uses for external capabilities.

The following figure shows agents using MCP to access external resources.



However, the MCP standard does not solve all scaling and governance challenges. Implementation of MCP servers must be combined with effective tool design, hosting, and enterprise governance strategies. This guide provides best practices for each strategy to help you build and use MCP as part of your agentic AI solutions.

When to use MCP

MCP provides strategic infrastructure for scaling your agentic AI initiatives. By centralizing tool management and governance, MCP servers reduce the cumulative cost of building and maintaining custom integrations across multiple agents. This delivers increasing returns as your agent ecosystem expands.

MCP likely becomes part of your strategy when:

- You need centralized governance for how agents access enterprise systems and services, such as databases, APIs, internal tools, and third-party integrations.
- Developers spend too much time writing custom integrations that are not consistent across implementations.
- You have duplicated tools that could serve common capabilities.
- You want to offer your proprietary tools or data to external consumers or third-party agentic systems through standardized, governed MCP interfaces, unlocking new revenue streams while maintaining security and control.

After you decide that MCP servers are going to be part of your strategy, evaluate whether existing open-source MCP server implementations meet your needs, whether they require enhancement, or whether you need to build custom servers. Many pre-built MCP server implementations are available in public repositories, and they cover common capabilities such as file system access, web browsing, code sandboxes, database access, and API integrations.

In many cases, pre-existing MCP servers are sufficient. For example, AWS provides the [AWS MCP Server](#), a managed remote MCP server that provides AI assistants and agents with secure, authenticated access to AWS services through natural language interactions. You can use the AWS MCP Server to perform complex, multi-step AWS tasks by combining real-time access to AWS documentation, syntactically correct API calls, and pre-built workflows called [Agent SOPs](#) that follow AWS best practices. AWS continuously tests the AWS MCP Server to make sure that customer agents can use them successfully.

You should test these existing MCP servers with your agents to determine if they fulfill your use cases. If an agent fails to complete workflows, generates incorrect or suboptimal responses, fails to navigate complex multi-step processes, or misses important domain-specific best practices or security considerations, you should consider enhancements in several dimensions.

When existing MCP servers do not fully meet your needs and they struggle to use the existing tools correctly or produce accurate responses, consider these enhancement approaches before building custom servers:

- **Enrich agent context** – If your agent struggles to correctly or efficiently use the tools in an existing MCP server, consider supplementing those tool definitions with additional documentation or examples. This helps provide additional context to the LLM.
- **Add complementary tools** – Extend existing MCP servers with tools that access additional organizational data or context that agents need to complete workflows successfully.
- **Improve underlying APIs** – Simplify your service APIs to make them more LLM-friendly by reducing parameter complexity, providing clearer error messages, and offering sensible defaults that agents can use.

While using existing MCP server implementations accelerates development for common capabilities, building custom MCP servers is a necessity when your use case requires specialized functionality. Custom MCP servers help you encapsulate domain expertise, enforce organizational standards, improve agent reliability for complex workflows, and support compliance with security requirements. Consider building a custom MCP server in the following situations:

- **Domain-specific workflows** – Multi-step workflows requiring domain expertise should be encapsulated in custom MCP tools when the necessary knowledge is not captured in API documentation. For example, instead of letting agents orchestrate complex healthcare data pipelines that must validate Health Insurance Portability and Accountability Act (HIPAA) compliance, anonymize PII, and transform to [HL7 FHIR](#) format, provide a `process_patient_data` tool that embeds the domain expertise directly. This removes the dependency on the LLM to correctly orchestrate and run the workflow steps, which improves consistency and compliance.
- **Golden path abstractions** – Agents might struggle to implement optimal approaches because they lack organizational context and default to basic patterns rather than organizational best practices. In these scenarios, you can enforce prescriptive standards for cost, performance, or security by encapsulating these golden paths in custom MCP tools. For example, instead of letting agents deploy infrastructure with default settings that might be unsecure or inefficient,

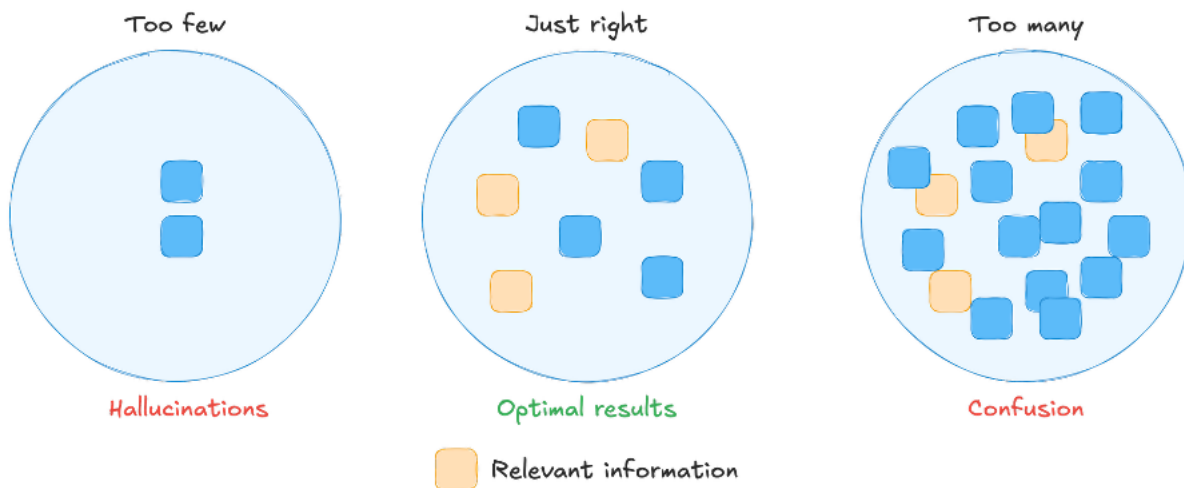
provide a `deploy_secure_infrastructure` tool that directly embeds your organization's standards.

- **Complex multi-service orchestration** – Instead of making the agent orchestrate complex workflows by trying to infer the correct sequence and set of services to use in each step, you can deterministically build that logic inside an MCP tool. You may also want to provide expertise about optimal service integration patterns that the agent might be unaware of. This can also improve the accuracy and efficiency of your agents.
- **Service-specific best practices** – This is common for security-focused tools that help agents implement encryption policies, access controls, and compliance patterns specific to the service being accessed through the agent tool. Additionally, if there are service-specific operational best practices that are not obvious, using an MCP server can help you make sure that they are implemented and not left up to an agent to reason about.

MCP tool design strategy

The main job of the MCP client and server is to discover and present tools to the LLM so that it can use them to improve its responses. This makes MCP tool design one of the most important strategies for building effective MCP solutions. From the model's perspective, tools are a function that they can invoke as needed to provide more accurate and complete responses. The function interface abstracts the underlying implementation of a tool, which can range from a wrapper around a single API call to complex workflow logic.

However, you must strike a balance with the quantity of tools provided to the LLM. If there are too few tools, the LLM might not be able to collect the right context and information, so it will take the best guess with the information available within the model. If there are too many tools, the LLM may get confused about the right tool selection and sequence, leading to hallucinations. Your goal is to get the number of tools just right. The following image shows the challenges of too few and too many tools.



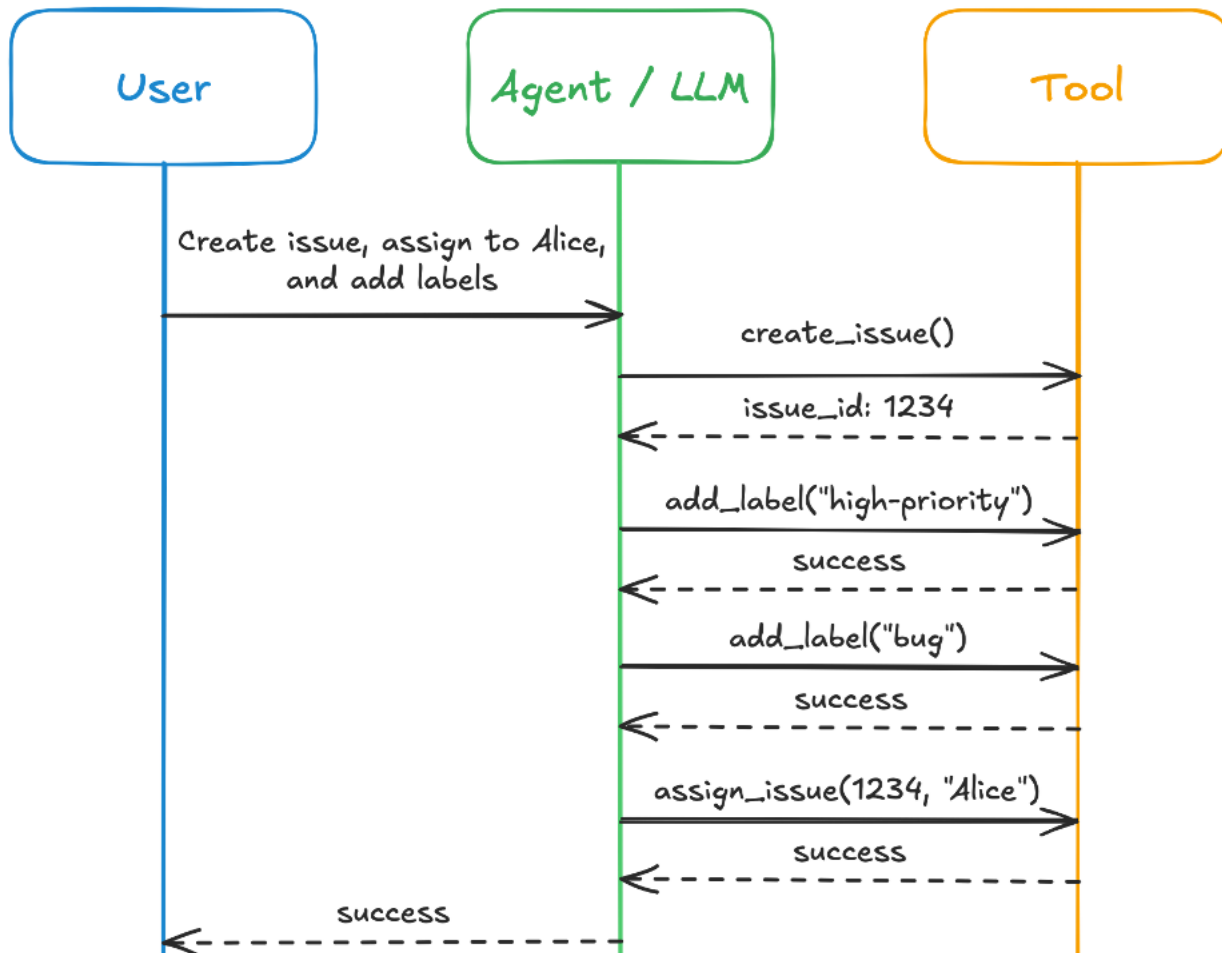
The solution requires understanding how many tools to provide and how to scope each tool. The granularity of your tools, whether they map to individual API calls or complete workflows, directly impacts the total number of tools that agents need and how effectively they can use them. This section provides best practices for scoping MCP tools, creating tool definitions, discovering tools, and organizing them.

Tool scope

There are two approaches for developing tools: granular and coarse-grained.

Granular

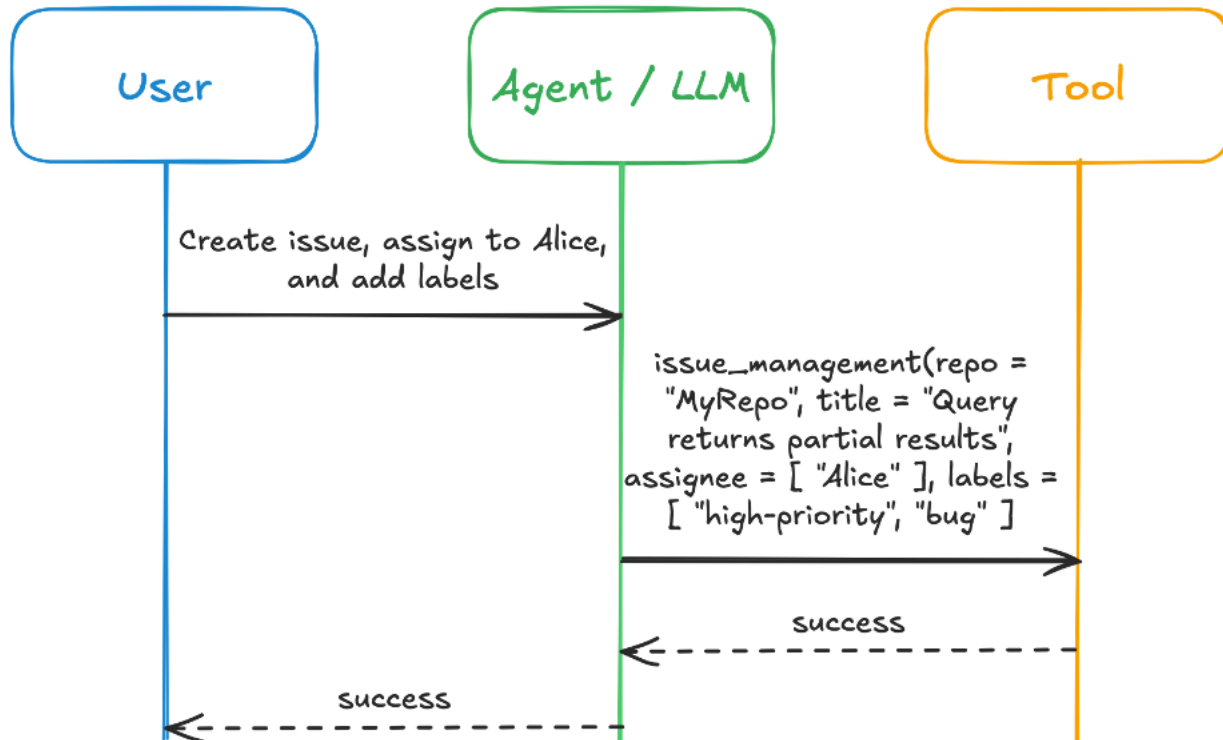
In a granular approach, you would create a tool per API, action, or query. For example, you could create `create_issue`, `get_issue`, `add_label`, `assign_issue`, and `close_issue` tools for your Git repository. This would allow the LLM to make granular calls to each API and orchestrate each one as necessary. Consider the following prompt: "Create an issue for the product service called 'Query only returns partial results', label it as a bug and high-priority, and assign it to Alice." The following image shows how a tool-per-API approach would respond to this prompt.



In this approach the system prompt and every registered tool definition are provided to the LLM on each call. This consumes additional context and incurs a latency penalty because each tool call represents an individual call to the LLM. It also increases the complexity of handling errors within the workflow.

Coarse-grained

A coarse-grained, or *workflow-driven*, approach would be tools that are workflow oriented. The tool focuses on end-to-end user intent over API structure. Instead of a tool-per-API, you have one tool that deterministically calls many APIs. Using the previous Git repository example, you could create a `create_and_setup_issue` tool that is called once by the agent. The tool implementation creates the issue, adds labels, and assigns it to a user, based on the parameters provided to the tool. The following image shows how a coarse-grained approach would process the same prompt.



This approach shows how all complexity remains hidden from the LLM layer. When the orchestration logic is embedded within the tool implementation, all of the sequential steps, logging, retry logic, circuit breakers, and rate limiting are performed deterministically in the tool. The workflow-driven approach makes it simpler for the LLM to invoke the correct tool with the right parameters. It is important to note that some APIs might already provide workflow intent, such as the Amazon EC2 RunInstances API. In these cases, a tool-per-API might provide the workflow-oriented design you desire.

However, tools can become too coarse-grained as well. If your single workflow tool attempts to do too many things and has many possible parameters, the LLM can struggle to reason about how to correctly use the tool. It can also create challenges with parameter selection and error handling. Thus, tool development must strike a balance that aligns with user intent and avoids too little or

too much functionality in a single tool. We recommend that you design tools around complete user workflows, bundling operations that commonly occur together (such as three or more API calls). We also recommend that you decompose tools that exceed eight or more parameters or handle multiple distinct user intents. Test with real prompts to verify that agents can correctly use each tool.

If you have complex and dynamic workflows that cannot easily be encapsulated as a deterministic tool, you might consider using the *agent-as-tool pattern*. Instead of your primary agent trying to orchestrate complex tasks in a workflow, a specialized agent can act as a tool. These types of tools can implement advanced decision-making and branching, and they can handle errors and retry logic that cannot be easily managed in deterministic code. This is similar to, but distinct from, the [Agent2Agent \(A2A\) protocol](#). The A2A protocol is complementary, providing interoperability and collaboration between agents in any agentic framework.

We recommend that you start with your workflow analysis by mapping your most common user workflows to identify the core capabilities each agent needs. This establishes your minimum viable toolset. Based on our experience developing MCP servers at scale, we recommend the following practices. When these practices conflict, prioritize user intent and workflow.

Best practices for MCP tool scoping

- **Think in user stories and bundle common operations** – Tools should map directly to complete user interactions rather than requiring orchestration of multiple operations. If workflows commonly require three or more separate calls, combine them into a single tool. This reduces the cognitive load on the LLM, minimizes the number of tool calls, reduces context consumption and latency required to complete tasks, and improves accuracy and latency.
- **Limit parameters to eight or fewer** – If a tool exceeds eight parameters, decompose it into multiple tools. LLMs struggle with parameter selection as complexity increases.

Note

If bundling operations requires more than eight parameters, prioritize bundling over parameter count because simplifying the workflow is more valuable than strict parameter limits.

- **Separate read and write operations** – Provide different tools for reading data and modifying it. This separation makes it explicit when agents are performing potentially destructive operations,

enables different authorization policies, and reduces the risk of unintended modifications during information gathering.

- **Provide sensible defaults** – Design tools so that the LLM needs to specify only the parameters that are specific to the individual request. Defaults reduce parameter complexity and improve tool selection accuracy by minimizing the information that the LLM must reason about.
- **Prefer deterministic execution** – Make tool execution and output deterministic when possible. Deterministic tools are more reliable and easier to test. For complex workflows that require intelligent orchestration, branching logic, or advanced error handling that cannot be easily managed in deterministic code, consider using specialized agents as tools. However, use this pattern selectively because it adds complexity.

Tool definitions

When an LLM receives a request that it cannot handle directly, it will review available tools to help it complete the request. The LLM selects tools based on its semantic understanding of the provided tools' names and descriptions and any instructions provided in the prompt. It will then create input based on the defined input schema and expect output based on the output schema. Therefore, creating descriptive tool definitions and validated input and output schemas is critical in helping the LLM to select tools effectively. There are generally two approaches for creating this documentation: the tool specification approach and the docstring approach.

Tool specification approach

The recommended approach is to directly follow the MCP [tool specification](#) when defining the tool. The following example is shown using the [Strands Agent](#) tool decorator:

```
@tool(  
    name = "search_website",  
    description = "This tool searches the provided website for semantic matches to the  
query provided",  
    inputSchema = {  
        "json": {  
            "type": "object",  
            "properties": {  
                "url": {  
                    "type": "string",  
                    "description": "The url of the website to load and search."  
                },  
                "query": {
```

```
        "type": "string",
        "description": "The content you want to try and match in the website."
    }
},
"required": ["url", "query"]
},
outputSchema = {
    "json": {
        "type": "object",
        "properties": {
            "results": {
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        }
    }
}
)
def search_website:
    ...
```

Using standard fields, such as name, description, inputSchema, and outputSchema makes sure that every tool has consistent documentation that both the LLM and humans can understand. Every tool should define these fields at a minimum and optionally provide a title and annotations, which are optional hints about tool behavior. When possible, use enums for parameter values to make it easy for the LLM to select the correct options. Enums work best for finite sets, such as status or priority values, but are not suitable for free-form text, dynamic values, arbitrary numbers, or resource identifiers. In those cases, provide clear descriptions and examples instead. Also include a default value when possible so the LLM does not have to guess what the correct option is. Keep in mind that tool definitions are included in the LLM prompt on each invocation, consuming context window space alongside system instructions and conversation history.

Docstring approach

Another approach, if you are writing your tools in Python, is using docstrings to provide the tool's description, usage, and output. The following is an example of this approach:

```
def search_website(url: str, query: str) -> list:
```

```
"""
```

```
    This tool loads the specified website and then attempts to find content that
    matches the provided query through semantic search. It provides back a list of strings
    that are the sentences that match the query.
```

```
    Args:
```

```
        url: the website url to load
```

```
        query: the content you want to semantically match in the website
```

```
"""
```

Docstrings do not enforce a schema or standardized format. Using this approach might yield inconsistent results based on how tool developers choose to document each tool. Defining and enforcing an organization-wide standard is essential if you follow this approach.

Best practices for MCP tool definitions

- **Follow MCP tool specification** – Provide name, description, inputSchema, and outputSchema fields for every tool. For Python implementations, use [Pydantic models](#) to provide inline documentation through field descriptions, automatic type validation, and constrained values through enums. This makes schemas self-documenting and improves LLM understanding of valid parameter options.
- **Write descriptions as prompts** – Tool descriptions are instructions that guide LLM decision-making. Include the essential components of the tool's purpose (what the tool does), when to use it (user intent patterns or scenarios), the context of the output (what the output is used for), parameters, and error conditions.
- **Provide concrete examples** – Including workflow examples with actual values is the most effective way to guide LLMs about correct tool usage.
- **Document dependencies explicitly** – Include prerequisites, numbered sequences, state changes, and follow-up actions.

Tool discovery

There are three approaches for discovering and registering tools in your agent with MCP servers: static definition, dynamic discovery, and search function.

Static definition

First, you can statically define the available tools directly in the agent code. In this approach you define a remote tool (a client-side reference object in a framework like Strands Agent SDK) for each

tool provided by the MCP server that is accessed by an MCP client. The following example uses streamable HTTP transport:

```
from mcp.client.streamable_http import streamablehttp_client
from strands import Agent
from strands.tools.mcp import MCPClient

streamable_http_mcp_client = MCPClient(
    lambda: streamablehttp_client("https://mcp1:8000/mcp")
)

reverse_text = RemoteTool(
    name="reverseText",
    client=streamable_http_mcp_client
)

agent = Agent(tools=[reverse_text])
```

Individual tool registration helps you to be very selective about the tools you make available to the LLM, which minimizes the amount of context window used. The tradeoff is that it requires knowing the names of the available tools and can be brittle if the available tools change in the MCP server.

Dynamic discovery

The next approach is using dynamic discovery and registering all the available tools with the agent. This approach consumes context linearly as more tools are added to the MCP server. The following is an example of this approach:

```
from mcp.client.streamable_http import streamablehttp_client
from strands import Agent
from strands.tools.mcp import MCPClient

streamable_http_mcp_client = MCPClient(
    lambda: streamablehttp_client("https://mcp1:8000/mcp")
)

with streamable_http_mcp_client:
    tools = streamable_http_mcp_client.list_tools_sync()
    agent = Agent(tools=tools)
```

Consider a scenario where a typical tool definition consumes approximately 250–500 tokens (including name, description, and schema). Registering 20 tools would consume 5,000–10,000 tokens of your context window. When you have a small number of MCP servers and you have control over the number of tools, this option is the simplest to implement. However, if the list of tools is expected to grow, it can create silent context management issues in your agents. An alternative variation of this approach is to use a tool filter parameter when calling `list_tools`, such as what the [Strands Agents SDK provides](#), to reduce the number of tools that are registered with the agent.

Search function

The third option is to use a search function to find relevant tools during runtime. You list all available tools from your MCP server and then perform a semantic search over those tools based on the user prompt. Then, the resulting tools are registered with your agent. [Amazon Bedrock AgentCore Gateway](#) provides a [native semantic search capability](#) that can make this kind of solution easier to implement.

Best practices for MCP tool discovery

- **Context window preservation** – Pick a tool discovery and registration approach that conserves as much of your context window as possible.
- **Use tool filtering or semantic search capabilities** – Dynamically provide the LLM with a scoped-down set of tools to choose from, which improves its accuracy and effectiveness at choosing the right tool. Tool filtering can operate on tool names (exact matching or patterns), tool descriptions (semantic matching), or domain or category tags. Semantic search is particularly effective for matching user intent against tool descriptions. Both approaches reduce context window use.

Tool organization

Discovering the right tools and ensuring the LLM can use them effectively is one of the most critical parts of effective tool development. As you start to develop MCP servers, you need a strategy that determines:

- How many tools go into one MCP server
- What tools should not be put into the same MCP server

- How to name tools to make them searchable and prevent name collisions (different tools with the same name)
- How to document the tools and MCP server to make them easy to use by the LLM

Namespace organization is a design pattern that prevents tool name collisions, groups related functionality, and facilitates efficient tool identification by LLMs. The pattern establishes structured categorization that is analogous to organized storage systems rather than unstructured accumulation. We recommend the *domain-noun-verb* pattern for tool naming. For example, `github_issue_create`, `github_issue_list`, `github_issue_update`, `github_pullrequest_create`, `github_pullrequest_list`, or `github_pullrequest_merge`. The advantage of this pattern is evident when examining alphabetical sorting behavior. When tools are listed alphabetically, all issue-related operations cluster together (`create`, `list`, `update`), followed by pull request operations (`create`, `list`, `merge`). The noun (resource type) functions as an organizational boundary. This structure facilitates both LLM tool scanning and human documentation navigation because related functionality naturally groups together.

The MCP server should be bounded at the domain level but may be sub-divided based on the separation of duties for the capabilities that it provides. For example, you might have separate MCP servers for write operations and read operations to a database. To enforce this separation, it is recommended that you implement guardrails at the agent level that restrict which MCP servers can be accessed based on user intent and permissions. This can be achieved through a combination of the following:

- **Conditional server loading** – Load the read-only MCP server only when the agent detects read operations in the user input.
- **Permission-based filtering** – Use user authorization to grant access to only appropriate MCP servers.

Finally, you will want to create an upper bound on the number of tools provided by an MCP server. Do not make assumptions about how agents will use your MCP server. They may naively list all the available tools and provide them all to the LLM. If you have more than 50 tools in a single server, you should consider splitting it into multiple servers.

Best practices for MPC tool organization

- **Use the domain-noun-verb naming standard for tools** – Implement strategies to prevent name collisions in both MCP servers and in agents.
- **Set an upper bound** – Restrict the number of tools in a single MCP server.
- **Divide MCP servers** – Use separation of duties to divide MCP servers into logical groups.

MCP hosting strategy

Abstracting the available tools into MCP servers decouples your agent development from the available tools. This introduces the challenges of where you host your MCP server and how tools are organized inside those servers.

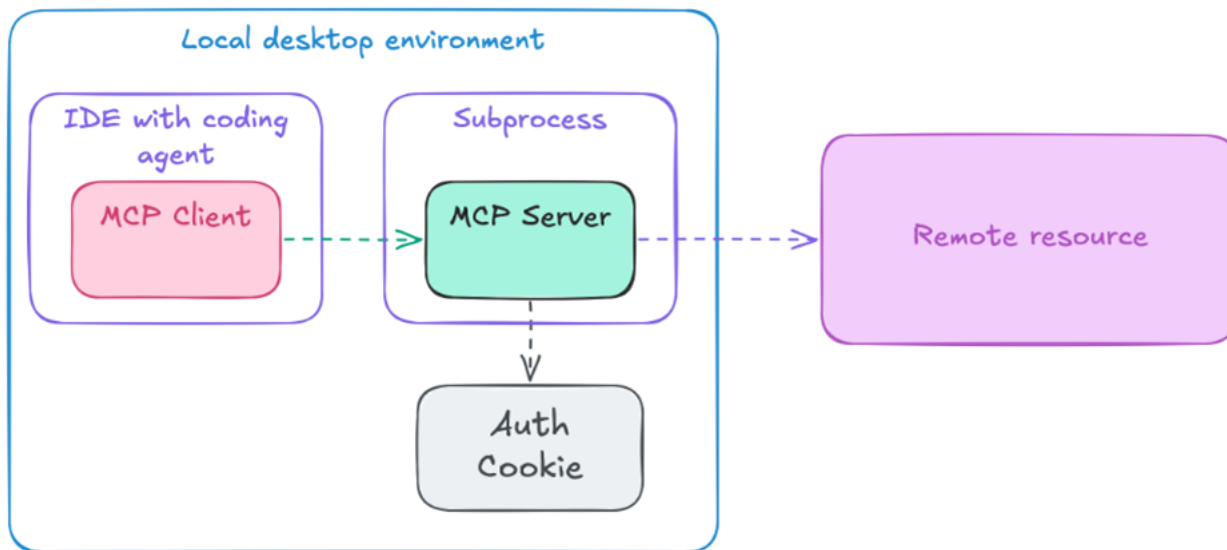
Hosting approaches

There are three options for hosting your MCP servers: running them locally on an end-user machine, hosting them remotely, or hosting them through an MCP gateway. Each option has advantages and tradeoffs.

Local hosting

Local hosting runs the MCP server as a subprocess on your local machine along with the agent that communicates with the server by using JSON-RPC over standard input and output streams. This approach does not require authentication between the client and the server. Tools can interact with local applications and files, use locally stored credentials, and they inherit the network access of the user's local machine. This is the simplest hosting pattern and has several benefits.

Many customers get started with MCP using local servers. They allow engineers to rapidly iterate and solve a variety of problems from their local environment. Consider an MCP server that connects to a Git repository that an engineer's coding assistant is using. Keeping the MCP server local makes a lot of sense because it can use the engineer's unique credentials to access the repository, and it does not add an extra network call to a remote MCP server. The following image shows a locally hosted MCP server being used with a coding agent in an IDE.



For these types of deployments, you must consider how the MCP servers are developed and distributed. Most customers develop an MCP registry where servers can be registered and downloaded by end users. It's very similar to a container registry where a user can search for specific capabilities and find the MCP servers suited to their needs.

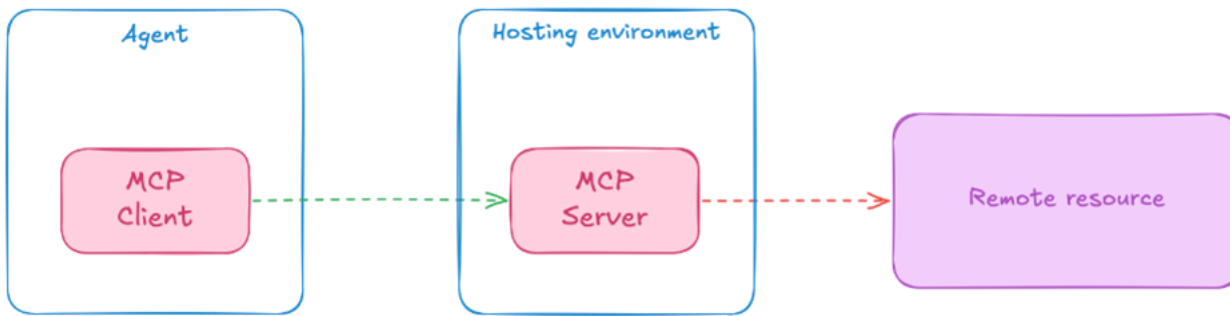
There are public MCP registries, such as [Official MCP Registry](#), and there are privately hosted registries. Organizations typically align their MCP registry strategy with existing policies around open source software distribution, container registries, and internal package management. You should consider factors like security scanning, approval workflows, and compliance requirements.

However, local hosting introduces operational challenges that organizations should consider. First, end-users must discover, download, and configure MCP servers independently. This can add complexity to get started with each individual MCP server they use locally. Second, you cannot control the MCP server lifecycle, meaning that users may continue running outdated versions locally with security vulnerabilities or missing features. This can complicate meeting compliance requirements. Some IDEs and CLI tools, such as [Kiro](#), allow organizations to [manage and control which MCP tools are available](#), ensuring consistency and security across teams.

Remote hosting

The second option is to host remote MCP servers that are accessed over HTTP or HTTPS. This provides access to any network-connected client. Using remote hosting allows you to centrally control access to MCP resources and capabilities, implement authentication and authorization, and control the versioning and updates of the MCP server logic. Remote hosting still requires the use

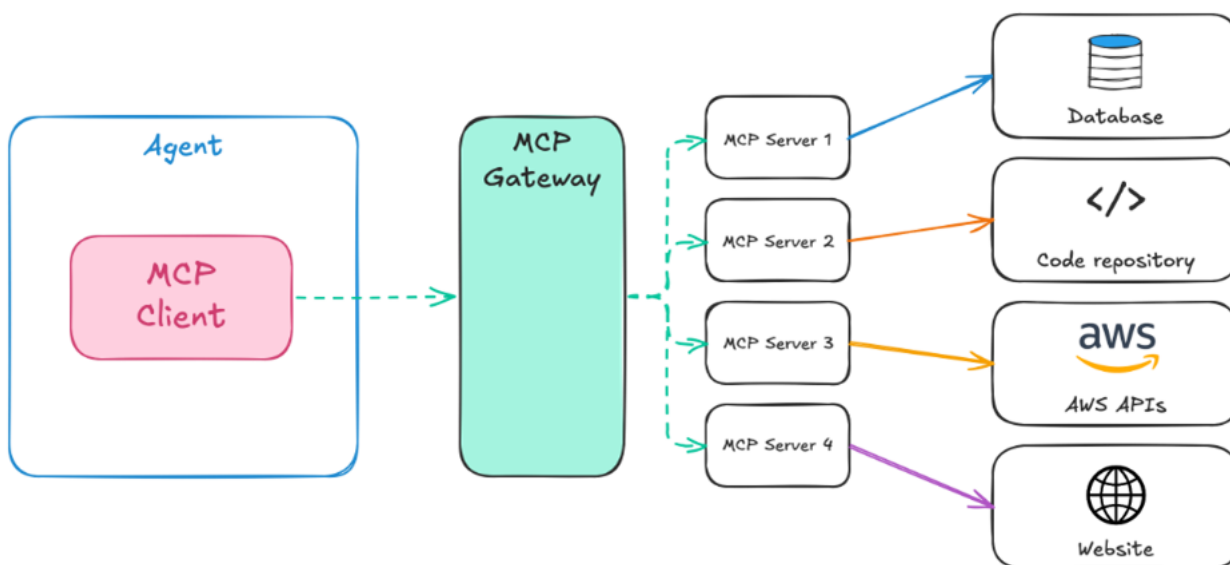
of an MCP registry so that end-users can discover the MCP servers that they want to use with their agent. The following image shows the remote hosting approach.



From an agent development perspective, the experience is similar whether the MCP server is local or remote. The most significant change is implementing authentication and authorization, including both the agent's access to the MCP server and the server's access to external resources. Remote MCP server implementations must be carefully planned to consider multi-tenant access and privilege management. The [MCP governance strategy](#) chapter contains more information about authentication and authorization considerations.

MCP gateway

The final option is using an MCP gateway. MCP gateways act as a centralized proxy between MCP clients and servers, and they orchestrate access to the registered MCP servers. Without a gateway, each agent needs to register every remote MCP server that it might want to use. A gateway allows the agent to connect to a single endpoint that manages the authentication, authorization, routing, and protocol translation. New MCP servers and tools can be added dynamically and made immediately available to the agent. The following image shows the MCP gateway approach.



Some gateway solutions, such as [Docker MCP Gateway](#), also manage the lifecycle of the MCP servers, launching servers on-demand as needed. MCP gateways, such as [Amazon Bedrock AgentCore Gateway](#), can also help manage tool discovery by providing [native semantic search capabilities](#). This provides agents with a single endpoint to connect with an MCP client and helps optimize their context window usage. The result is simple agents that can choose and use MCP tools effectively. However, it has similar identity-related challenges as the remote MCP server approach.

Best practices for hosting MCP servers

- The spectrum of hosting options are not a one size fits all. Much of the usage of MCP servers today is local.
- As you start to use remote MCP servers, your main consideration is consistent authentication and authorization to the MCP server and how the MCP server performs authentication and authorization to downstream resources.
- MCP gateways simplify connectivity and authentication and authorization for hosting multiple remote MCP servers. They also provide capabilities to improve context window management by searching for applicable tools.

MCP governance strategy

The other critical capability that MCP offers organizations is support for centralized governance. Your MCP governance strategy should address authentication and authorization to both the MCP servers as well as the resources they access. It should also address rate limiting to protect downstream resources, operational metrics for monitoring tool usage and performance, and managing deployments and distribution of MCP servers.

Authentication and authorization

One of the most important parts of your authentication and authorization strategy is managing downstream resource access from MCP servers. When a user calls an agent, authentication and authorization is performed to ensure the user has permissions to call the agent. Then, the agent orchestrates calling specific tools in MCP servers. You need to decide how to authorize access on a per-tool basis.

One option is *machine-to-machine authorization*, where user consent or interaction is not required. For example, a time-based agent invocation uses an MCP server to collect logs from an application and analyze them. In this scenario, the agent is pre-authorized to access the specified data. The second option is *user-delegated access*, where a user provides their consent to access user-specific data and resources.

The following table shows authentication and authorization patterns.

Factor	User-delegated access	Machine-to-machine
Data ownership	User-specific authorization to data	System or organization-wide data
User interaction	User is present and can consent	No user interaction
Operation timing	Interactive or real-time	Background, scheduled, or batch
Permission scope	Permissions vary by user	Consistent permissions at the agent level

User-delegated access requires careful implementation and should be developed with your security team. Agents must be able to evaluate which tools an LLM has selected and whether they require additional authorization. MCP tools must include descriptions to indicate their authentication and authorization requirements and where to retrieve access tokens. Client applications must support intermediate authentication requests, and the MCP client must provide the retrieved credentials back to the agent for each tool call.

You should ensure that MCP tools always have their own tokens to access external capabilities and that the access is logged and audited. User credentials should not be propagated through your agentic system. For example, your MCP servers should not use the same token to access data that was used to invoke the agent. Downstream calls should use explicitly scoped, purpose-generated tokens. This helps provide additional guardrails to prevent unintended data access on-behalf of actions. It can also help prevent hallucinations from producing unintended results. Imagine that a user with full admin permissions asks an agent to clone a production database for use in pre-production. To do so, the user only needs READ and CREATE permissions. Let's say the LLM hallucinates and believes it needs to clean up the old database as part of this request. If it reuses the user's credentials, it would likely succeed because the user's original credentials have DELETE permissions. Instead, if the MCP server uses an intentionally scoped-down token for the request with just READ and CREATE permissions, the attempt to delete the production database would fail.

You can use [Amazon Bedrock AgentCore Identity](#) to help implement these patterns. Make sure that you make an intentional choice about whether the permissions to list and invoke tools hosted by an MCP server implies permission to the external capabilities that the MCP server exposes. This identity flow from the MCP server to the resource and back to the user is dependent on the type of authentication and authorization service being used. You must decide how this is handled at scale for your MCP servers.

When designing your authentication and authorization patterns, implement token isolation mechanisms that retrieve different access tokens for each tool being accessed. Do not reuse tokens between tools and servers. AgentCore Identity provides this token isolation capability. It automatically manages both workload tokens (for machine-to-machine authentication) and user tokens (for user-delegated access) to ensure proper separation and prevent permission escalation. This is especially critical when incorporating remote MCP servers or MCP gateways.

Best practices for MCP authentication and authorization

- **Token separation** – Do not pass bearer tokens from callers to downstream services. Validate the aud (audience) field matches the server receiving the token. The audience claim specifies

which service the token is intended for, preventing unauthorized token reuse across different MCP servers.

- **Select an access approach** – Choose between machine-to-machine and user-delegated access for each tool your MCP servers provide. Consider grouping tools together in the same MCP server that use the same authentication pattern.

Controlling load

As with any distributed system, you must consider how to control load in your MCP server fleet. First, you consider whether to implement rate limiting in your MCP servers and where to implement the limits. If you choose not to implement rate limiting, you pass on any rate limiting performed by downstream resources. Many systems choose to rate limit based on request attributes, such as a user or account ID. Validate that the requests sent to downstream services carry on those same attributes so that multiple users are not affected by load being driven by another user.

If you do choose to implement rate limiting, the recommended approach is to implement primary rate limiting at the MCP server level, with backend services providing secondary protection and agents adapting their behavior based on rate limit feedback. Consider whether the rate limits are per-MCP server or per-tool. Per-MCP server rate limits help protect your MCP server fleet and services in a multi-tenant environment. However, that can be very coarse-grained. Per-tool rate limits are designed to prevent overwhelming downstream resources that might not sufficiently rate limit themselves. If a tool calls multiple APIs, you should set the rate limit to align to the lowest rate allowed by those APIs.

Passing rate limit information in HTTP headers can also be a useful metric for users and automated systems to help manage their own request rate and retry strategy. For example, you might send these headers back to the agent from your MCP server, as shown in the following example:

```
X-RateLimit-Limit: 100  
X-RateLimit-Remaining: 45  
X-RateLimit-Reset: 1640995200
```

Additionally, consider load shedding to protect the overall service when no single customer is exceeding a rate limit but the load is impacting the system's performance.

Best practices for controlling load

- **Choose a rate-limiting approach** – Plan to rate limit individual users based on either their use of downstream resources or through their use of your MCP server and tools.
- **Consider load shedding** – Protect your MCP server fleet from general overload that is not driven by a single or handful of customers.

Operational metrics

Key metrics to capture for MCP implementations should focus on the customer experience they deliver. These metrics commonly include token usage, tool selection accuracy, number of tools registered with the agent, and tool latency. For example, monitoring output tokens returned by each tool enables you to set alarms when tools exceed a threshold for context window usage. When a tool exceeds that threshold, you might want to review the tool's behavior. This ties into the MCP tool design strategy as well. Tool selection accuracy metrics indicate how well agents choose appropriate tools for given tasks, while execution speed and success rates highlight performance bottlenecks and reliability issues.

For example, to evaluate the tool-selection and tool-use accuracy metrics, AWS teams created golden datasets for regression testing. The datasets were generated synthetically by using LLMs from historical API invocation logs upon user queries. Using the pre-defined tool-selection and tool-use metrics (such as tool selection accuracy, tool parameter accuracy, and multi-turn function call accuracy), the AWS teams could objectively evaluate the AI agent's ability to correctly identify the appropriate tools, populate their parameters with accurate values, and maintain coherent tool invocation sequences across conversational turns.

Measuring metrics about the number of tools registered with an agent can help you identify potential context window management challenges as well as changes in the available tools presented by MCP servers. You should regularly review operational metrics that indicate the user experience with your MCP server and tools.

Contributors

Authoring

- Alex Torres, Sr. Solutions Architect, AWS
- Saikat Gomes, Sr. Customer Solutions Manager, AWS
- Mike Haken, Sr. Principal Solutions Architect, AWS
- Sreeja Das, Principal Engineer, AWS

Reviewing

- Ted Swinyar, Solutions Architect Manager, AWS
- Raju Patil, Senior Data Scientist, AWS

Technical writing

- Lilly AbouHarb, Senior Technical Writer, AWS

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an [RSS feed](#).

Change	Description	Date
Initial publication	—	March 16, 2026