Model Context Protocol (MCP) Deployment Patterns on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Model Context Protocol (MCP) Deployment Patterns on AWS

Table of Contents

Model Context Protocol (MCP) Deployment Patterns on AWS

Deepika Kumar and Lingling Cui, Amazon Web Services

Executive Summary

The Model Context Protocol (MCP) is an open protocol that enables seamless integration between AI applications and external data sources and tools. As organizations increasingly adopt AI-powered development tools like Kiro, Amazon Q Developer, Claude Code, and custom AI assistants, deploying MCP servers on AWS provides scalable, secure, and cost-effective infrastructure for contextual AI interactions.

This prescriptive guidance provides architecture patterns and implementation approaches for deploying MCP servers on AWS using AWS Lambda with Amazon API Gateway, Amazon Elastic Container Service (Amazon ECS), and Amazon Elastic Kubernetes Service (Amazon EKS) and others. Each pattern addresses different operational requirements, scaling needs, and management preferences.

The implementation examples and deployment scripts referenced in this guidance are available in the open-source repository at: https://github.com/aws-samples/sample-mcp-deployment-patterns

Intended audience

This guide is intended for cloud architects, DevOps engineers, platform engineers, and developers who want to deploy MCP servers on AWS infrastructure. It assumes L200-300 familiarity with core AWS services, networking concepts (Amazon VPC, subnets, security groups), and container fundamentals.

Objectives

After reading this guide, you will be able to:

- Understand the differences between local and remote MCP server deployment models
- Evaluate five AWS deployment patterns (Amazon Bedrock AgentCore, Lambda + API Gateway, Amazon ECS, Amazon EKS, Amazon EC2) for hosting remote MCP servers

- Select the appropriate deployment pattern based on your scaling, security, cost, and operational requirements
- Apply best practices for production MCP deployments aligned with the AWS Well-Architected Framework

Attachments

To access additional content that is associated with this document, download and unzip the following file:

attachment.zip

What is the Model Context Protocol (MCP)?

Overview

The Model Context Protocol (MCP) is an open protocol introduced by Anthropic that enables seamless integration between LLM applications and external data sources and tools to access real-time information. Whether you're building an AI-powered IDE, enhancing a chat interface, or creating custom AI workflows, MCP provides a standardized way to connect LLMs with the context they need.

Why MCP Matters

Traditional AI applications operate in isolation, lacking access to organization-specific data, internal APIs, and business tools. Each integration requires custom development, creating fragmentation and maintenance overhead. MCP addresses this challenge by providing a standardized protocol for AI applications to discover and interact with contextual resources through three core primitives:

1. Resources, which expose data and content such as files, databases, and APIs to AI applications
2. Tools, which enable LLM's to run actions like creating tickets, running queries, or calling APIs; and
3. Prompts, which provide reusable prompt templates with dynamic context.

Key Benefits

MCP delivers several significant advantages for organizations building AI-powered applications.

- Standardization through a single protocol eliminates the need for custom integrations for each AI-to-system connection.
- Discoverability allows AI applications to automatically discover available capabilities without manual configuration.
- Security is enhanced through centralized authentication and authorization at the protocol level.
- Maintainability improves as you can update integration logic once and benefit all connected AI applications.
- The protocol's extensibility enables adding new data sources and tools without modifying AI applications.

Common use cases

Organizations deploy MCP servers to enhance AI applications across various domains:

- AI Development Tools:#Organizations deploy MCP servers to connect with AI-powered development tools like Kiro, Claude Code, Cursor, or custom AI coding assistants with version control repositories, enabling real-time code analysis and context-aware suggestions. These integrations provide access to internal code documentation, API references, and development environment tooling, allowing AI to understand project structure and dependencies. The MCP server acts as a bridge between the AI assistant and the development infrastructure, enabling features like automated refactoring based on repository patterns, intelligent code completion using internal libraries, and CI/CD pipeline integration for deployment automation. This approach significantly accelerates development workflows by providing AI with a comprehensive understanding of the codebase and organizational coding standards.
- Enterprise Knowledge Access:#Enterprise knowledge management becomes significantly more accessible when MCP servers connect AI assistants to document management systems, databases, wikis, and internal knowledge bases. Employees can query organizational information conversationally, asking natural language questions that retrieve relevant documentation, technical specifications, and historical decisions. The MCP server provides secure access to customer documentation, support articles, product catalogs, and technical specifications, enabling AI assistants to provide accurate, context-aware responses.
- Workflow Automation:#MCP servers enable AI to interact directly with business process tools, creating a powerful automation layer for routine tasks. AI assistants can create and update tickets in project management systems like Jira or Asana, interact with CRM platforms to access and update customer data, and trigger business process automation workflows based on conversational requests. This integration streamlines operations by reducing context switching between tools and enabling voice-driven or chat-driven workflow execution.
- Data Analysis:#MCP servers translate natural language requests into appropriate database queries, retrieves results, and formats them for AI interpretation. AI assistants can generate reports, create visualizations, and provide summaries of business metrics by providing access to data warehouses, business intelligence tools, and analytics platforms without writing SQL queries or navigating complex dashboard interfaces.
- Customer Support:#Customer support operations benefit significantly from MCP servers that provide AI assistants with a comprehensive customer context. Support agents and AI chatbots gain access to customer databases, order history, product catalogs, and inventory systems through standardized MCP interfaces. AI assistants can provide personalized recommendations

based on customer preferences and past behavior, resolve common issues by referencing similar cases, and if required, escalating complex problems with full context to human agents. The result is faster resolution times, improved customer satisfaction, and reduced operational overhead for support teams

MCP server deployment models

Understanding where and how to deploy MCP servers is fundamental to building scalable AI-powered applications. MCP servers can run:

1. Locally on developer workstations for personal productivity and development scenarios or
2. Deployed remotely on cloud infrastructure for team-wide access and production workloads.

The choice between local and remote deployment significantly impacts the development of workflows, security models, scalability, and operational complexity.

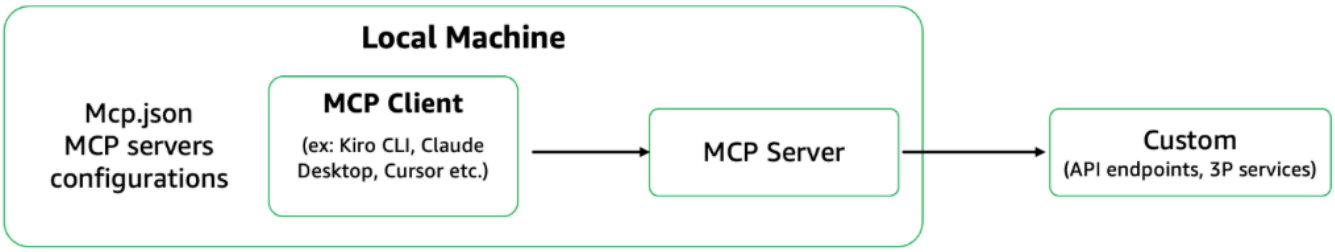
The following section explores both deployment models, their characteristics, and appropriate use cases. While local MCP servers excel for individual development and testing, remote deployments on AWS infrastructure enable enterprise-scale AI applications with centralized management, security controls, and reliable access for multiple users and applications.

Local MCP Servers

Overview

Local MCP servers run directly on a developer's workstation or laptop, operating as standard processes that communicate with MCP client applications through local protocols. This deployment model provides the fastest path to integrating AI tools with local resources and development environments, making it ideal for personal productivity, experimentation, and early-stage development.

The local architecture eliminates network latency entirely, as communication occurs through standard input/output (stdio) streams or inter-process communication (IPC) mechanisms. AI applications like Kiro, Claude Desktop/Code, Cursor, or custom development tools connect to local MCP servers through configuration files that specify the server's executable path and startup parameters.



AWS Documentation MCP server

Local MCP servers require minimal configuration, typically defined in the MCP client settings file. For example, provides tools to access AWS documentation, search for content, and get recommendations.

For test the MCP servers, developers can use the MCP Inspector which is an official tool from Anthropic that provides an interactive web-based interface for exploring MCP server capabilities, testing protocol compliance, and troubleshooting integration issues.

Common use cases for MCP servers deployed locally

- File system access for code analysis: AI-powered code editors use MCP servers to analyze project files, provide context-aware suggestions, and perform automated refactoring. The MCP server grants read and write access to specific directories, enabling intelligent code completion based on project structure, automated documentation generation from code comments, and refactoring operations that maintain consistency across multiple files
- Git repository integration: Developers build MCP servers that expose Git operations, allowing AI assistants to review commit history, analyze branch differences, and suggest merging conflict resolutions. This enables conversational interactions for version control tasks without leaving the AI interface.
- Local development tools: Integration with local build tools and test runners enables conversational commands like "run the unit tests" or "build the Docker image." Local database access for development instances provides AI with query capabilities for testing and data exploration during development.
- Personal Productivity Applications: Calendar integration enables AI assistants to check availability and schedule meetings. Note-taking applications like Obsidian or Notion expose their local databases through MCP, allowing AI to search notes, create entries, and maintain knowledge bases.

Limitations of local deployment

While local MCP servers are convenient for individual developers, they have inherent limitations that make them unsuitable for team collaboration and production workloads and face significant limitations:

- Security and Compliance risks: Sensitive credentials and API keys stored on individual workstations increase exposure risk through lost devices, malware, or inadequate access controls. Audit logging fragments across multiple machines, making compliance verification difficult. Organizations cannot enforce centralized security policies or access controls.
- Environment inconsistency: Each developer must install and configure MCP servers independently, leading to environment drift and version inconsistencies across team members. Sharing capabilities require duplicating deployment on each workstation, complicating maintenance, and updates.
- Limited scalability: Consumer laptops lack compute resources, memory, and storage for enterprise-scale data processing. Local servers cannot handle concurrent requests from multiple users or applications, restricting use to single-user scenarios.
- Reliability concerns: Workstation reboots, network disconnections, and hardware failures impact availability. No built-in redundancy or failover capabilities exist. Service availability depends entirely on the developer's workstation being powered on and connected.
- These limitations drive the need for remote deployment models on cloud infrastructure for team collaboration and production workloads.

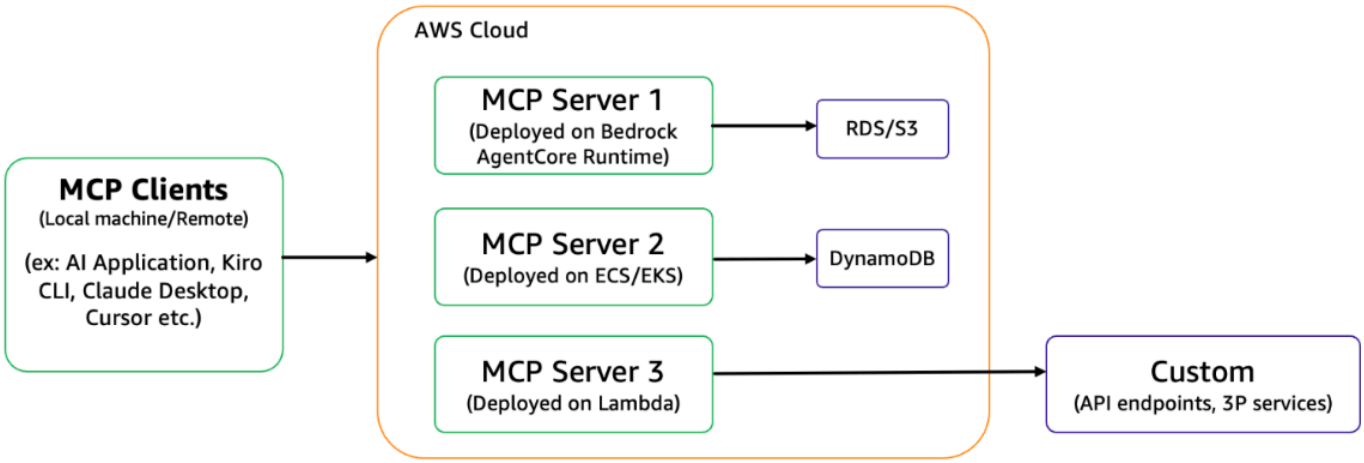
Remote MCP Servers

Overview

Remote MCP servers deploy on centralized cloud infrastructure, providing HTTP/HTTPS or WebSocket endpoints that multiple MCP clients access over the network. This deployment model transforms MCP servers from personal development tools into production-grade services that support team collaboration, enterprise security requirements, and scalable AI applications.

The remote architecture separates the MCP client from the MCP server through network communication. Clients connect to server endpoints using HTTPS for request-response patterns or WebSocket protocols for persistent bidirectional communication. Load balancers distribute traffic across multiple server instances, providing horizontal scalability and fault tolerance. Cloud

infrastructure enables automatic scaling based on demand, maintaining consistent performance during traffic spikes.



Key architectural considerations while migrating from local to remote MCP deployment:

- **Authentication and Security**: Remote MCP servers require robust authentication and authorization mechanisms. AWS provides multiple authentication options: IAM with SigV4 request signing for service-to-service authentication using temporary credentials, Amazon Cognito for user-based authentication with JWT tokens, and OAuth 2.0 flows for enterprise identity provider integration. API Gateway adds security layers including rate limiting, request throttling, API keys, and AWS WAF integration. Network security uses Amazon VPC for subnet isolation, security groups for traffic control, and TLS/SSL encryption. Organizations can deploy AWS PrivateLink or VPN connections for secure corporate network access without public internet exposure.
- **Network latency and reliability**: Communication crosses internet or corporate networks rather than remaining local. Mitigate through geographic proximity, caching strategies, efficient API design, retry logic with exponential backoff, and clear error messaging.
- **State management:** Stateless designs enable horizontal scaling but require external storage like Redis or DynamoDB for shared state. Stateful WebSocket patterns maintain context but complicate load balancing and failover.
- **Cost Management:** Balance performance against infrastructure costs through appropriate sizing, usage tracking, cost allocation, and reserved capacity for predictable workloads.

The following sections explore specific deployment patterns on AWS including Lambda with API Gateway for serverless deployments, ECS for containerized applications, and EKS for Kubernetes-based orchestration. Each pattern addresses different operational requirements, scaling needs, and team capabilities.

Why deploy MCP servers on AWS

Organizations choose AWS for remote MCP server deployments due to its comprehensive range of services, proven scalability, and deep integration capabilities.

- Flexible compute options: AWS provides multiple compute options ranging from serverless AWS Lambda functions to managed Kubernetes clusters on Amazon EKS, enabling organizations to select deployment patterns that match their operational maturity and scaling requirements with pay-per-use pricing that aligns costs with actual usage.
- Native service integration: AWS SDK enables seamless access to S3, DynamoDB, RDS, Bedrock, and other AWS services. Managed services eliminate infrastructure management, allowing teams to focus on business logic.
- Security and Compliance: Leverage existing AWS accounts, IAM roles, and security policies for consistent governance. Compliance certifications (SOC 2, ISO 27001, HIPAA, PCI DSS) support regulated industries.
- Global Infrastructure: This supports deploying MCP servers close to users worldwide, reducing latency and improving user experience, enabling data residency compliance and geographic redundancy.
- Operational Reliability: Operational maturity of AWS services provides production-grade reliability. Managed services like Application Load Balancer and ECS handle health checks, traffic routing, and rolling deployments. CloudWatch monitoring and alarms provide operational visibility and automated incident response.

Prerequisites

This guidance assumes L200-300 AWS experience with familiarity across core AWS services. You should understand networking concepts including Amazon VPC, subnets, security groups, and network ACLs. container fundamentals and Docker basics are necessary for ECS and EKS patterns. Before implementing any MCP deployment pattern, ensure you have the necessary AWS account permissions, tools, and MCP client configured.

Some AWS services aren't available in all AWS Regions. For Region availability, see AWS services by Region. For specific endpoints, see the Service endpoints and quotas page, and choose the link for the service.

- AWS Account and Permissions: You need an active AWS account with appropriate permissions to create AWS Lambda functions and execution roles for serverless deployments, Amazon EC2 instances and security groups for EC2 patterns, Amazon ECS clusters, task definitions, and services for containerized deployments, Amazon EKS clusters and node groups for Kubernetes patterns, Amazon API Gateway REST or HTTP APIs, Amazon CloudWatch Logs groups, and AWS Secrets Manager secrets for secure API key storage.
- Tools and Software: Install and configure AWS CLI version 2 or later, AWS CDK, CloudFormation or AWS SAM for optional IaaC, Docker for ECS and EKS patterns, kubectl for EKS pattern implementation, and Git for version control. Install the official MCP SDK for your programming language, such as @modelcontextprotocol/sdk for Node.js or other.
- MCP Client Application: You need an AI application that supports MCP, such as Amazon Q Developer, Claude Desktop, or a custom client. Understanding of MCP client configuration enables successful integration with deployed servers.

Remote MCP deployment patterns on AWS

AWS provides various services suitable for deploying remote MCP servers, each with distinct characteristics, operational models, and cost structures. Understanding these options allows you to select the deployment pattern that best aligns with your requirements, team capabilities, and organizational constraints.

Deployment Pattern 1 - Amazon Bedrock AgentCore

Amazon Bedrock AgentCore is a fully managed, purpose-built enterprise-grade platform for building, deploying, and operating highly capable AI agents securely at scale, removing the complexity of managing underlying infrastructure. The serverless runtime automatically scales to meet demand without requiring manual infrastructure management, ensuring high availability. Key advantages include enterprise-grade security with isolated sessions and comprehensive observability through integration with Amazon CloudWatch, making it a robust and hassle-free environment for hosting your agent's functionality.

How it works:

Amazon Bedrock AgentCore Runtime hosts MCP servers as containerized workloads. When you deploy an MCP server, AgentCore expects a container that exposes the MCP streamable-HTTP endpoint at `0.0.0.0:8000/mcp` — the default path supported by official MCP SDKs (e.g., FastMCP).

Deployment steps:

1. **Author the MCP server** — Use the MCP Python SDK (`FastMCP`) or TypeScript SDK to define your tools. Configure the server with `stateless_http=True` (default, recommended) or `stateless_http=False` for stateful multi-turn interactions.
2. **Test locally** — Run the server on `localhost:8000` and validate with an MCP client or the MCP Inspector.
3. **Scaffold the project** — Install the AgentCore CLI (`npm install -g @aws/agentcore`) and run:

```
agentcore create --protocol MCP
```

This generates the project structure including `agentcore/agentcore.json` configuration.

4. **Deploy** — Run `agentcore deploy`. The CLI packages your code and dependencies, uploads to S3, and creates an AgentCore Runtime. You receive an agent runtime ARN.
5. **Invoke remotely** — Clients call the deployed MCP server via the AgentCore invocation endpoint:

```
https://bedrock-agentcore.<region>.amazonaws.com/runtimes/<encoded-arn>/invocations?qualifier=DEFAULT
```

6. Authentication uses OAuth bearer tokens (via Cognito) or IAM SigV4.

Stateless vs Stateful modes:

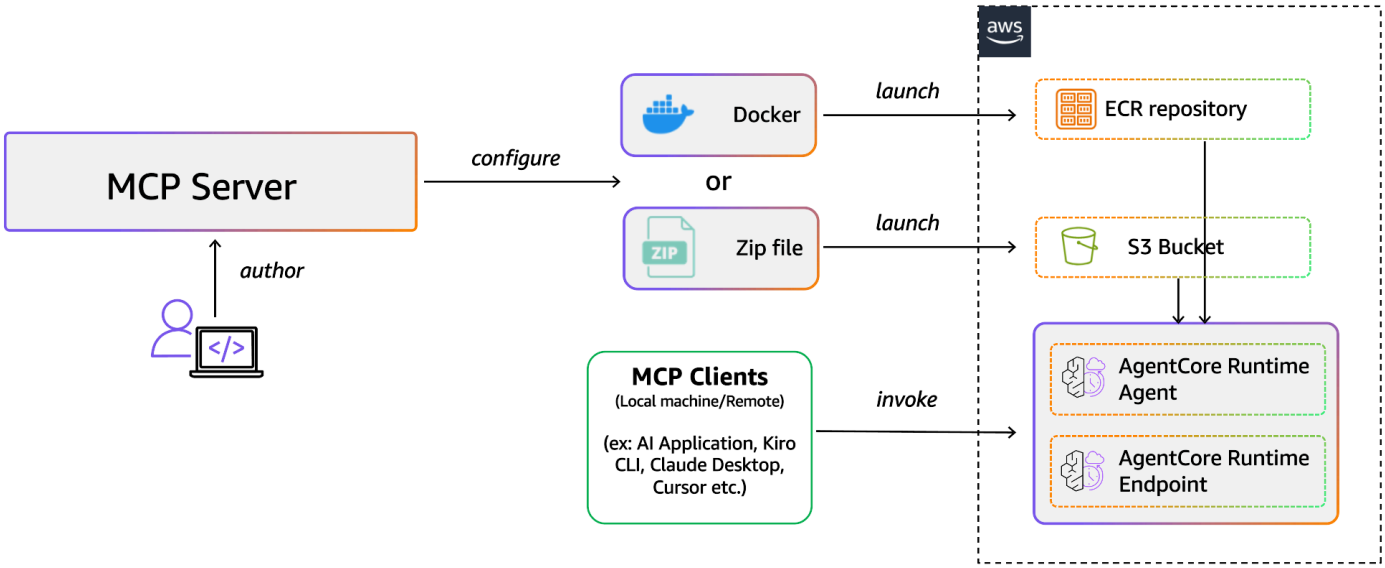
- **Stateless** (`stateless_http=True`): Recommended default. The platform auto-adds `Mcp-Session-Id` headers for connection continuity.
- **Stateful** (`stateless_http=False`): Required for elicitation (multi-turn prompts), sampling (LLM-generated content), or progress notifications.

For the full walkthrough, see Deploy MCP servers in AgentCore Runtime.

Sample Implementation:

- Amazon Bedrock AgentCore with IAM SigV4 sample implementation: https://github.com/aws-samples/sample-mcp-deployment-patterns/tree/main/deploy-agentcore-iam

- Amazon Bedrock AgentCore with OAuth sample implementation: https://github.com/aws-samples/sample-mcp-deployment-patterns/tree/main/deploy-agentcore-oauth



Architecture Characteristics

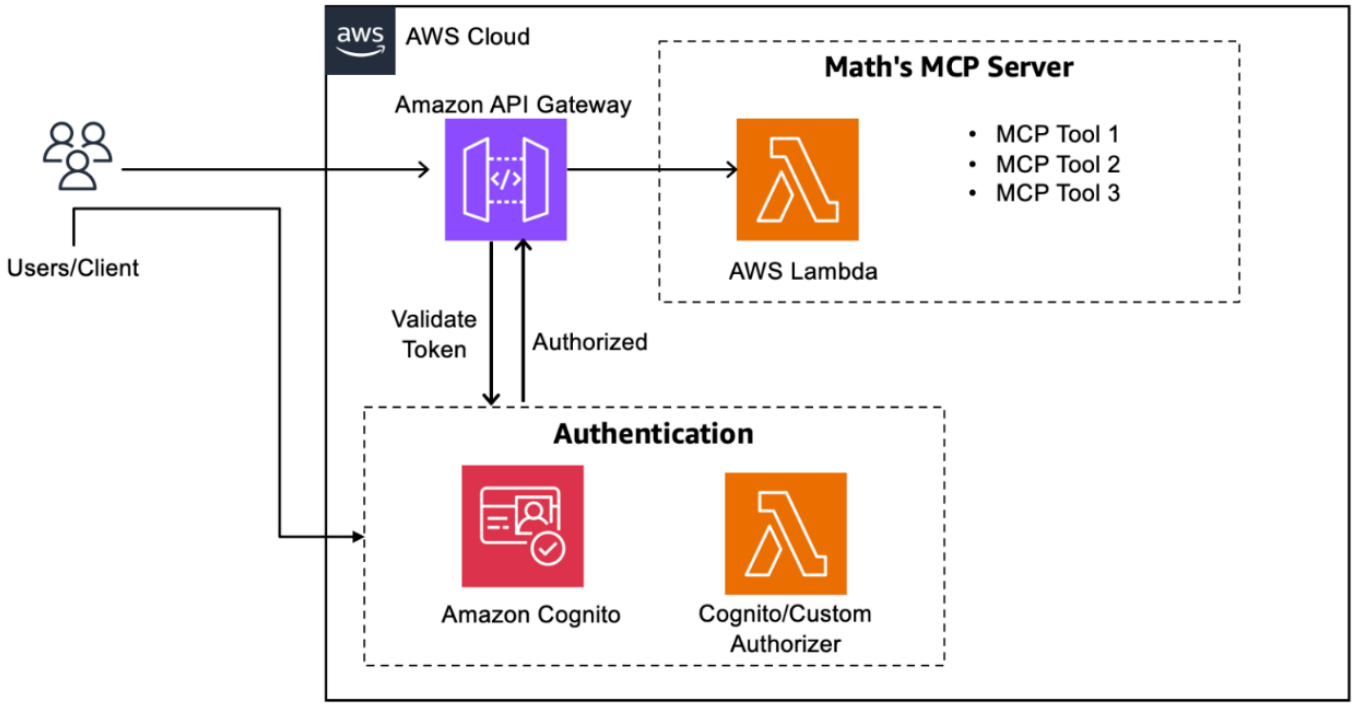
- Pros: infrastructure management (no servers, containers, or orchestration), built-in authentication and authorization, automatic scaling and high availability, integrated monitoring and logging, native AWS service access, rapid deployment (minutes vs hours/days).

- Limitations: AWS-specific (no portability), limited runtime customization compared to self-managed options, pricing based on request volume and compute time.

Deployment Pattern 2 - AWS Lambda + Amazon API Gateway

AWS Lambda with Amazon API Gateway represents the serverless approach where MCP server logic runs in ephemeral functions triggered by API requests. Lambda automatically manages the underlying compute resources, scaling from zero to thousands of concurrent executions without manual intervention. This pattern significantly reduces infrastructure management overhead, making it ideal for variable workloads and teams prioritizing rapid development over operational control.

Sample implementation: https://github.com/aws-samples/sample-mcp-deployment-patterns/tree/main/deploy-serverless



Architecture Characteristics

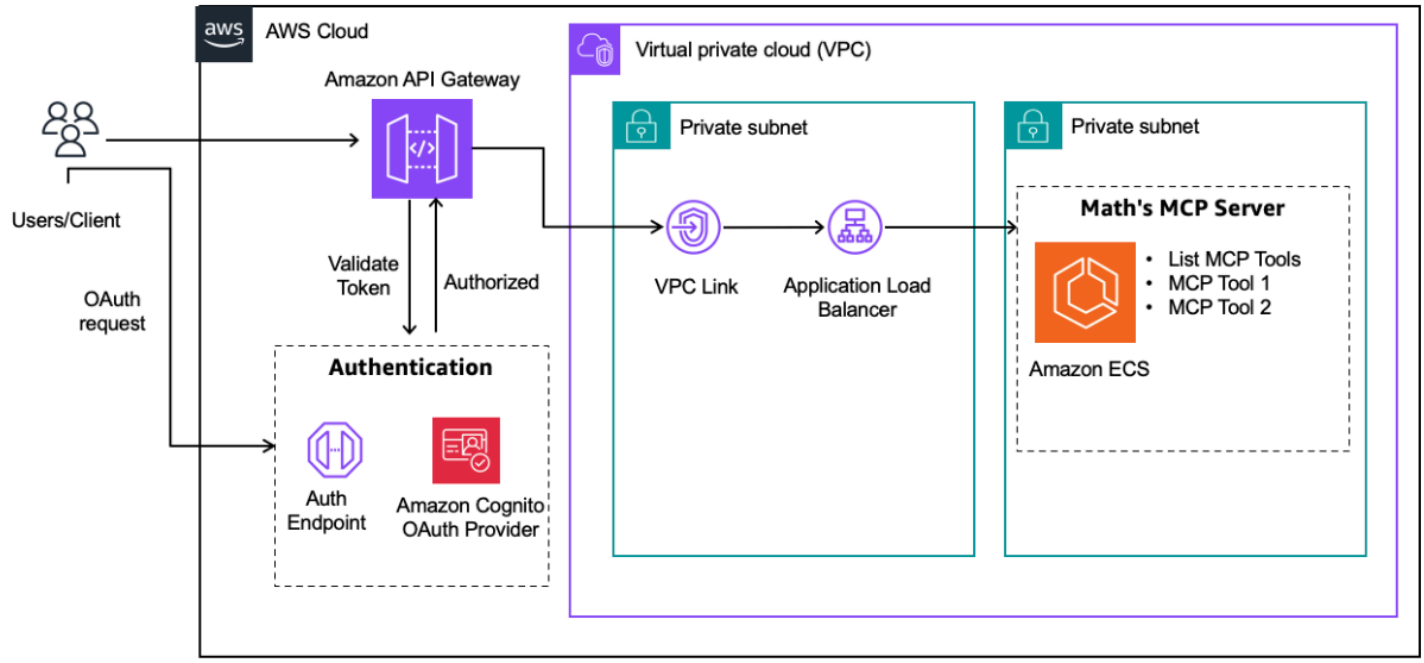
- Pros: Zero server management with automatic infrastructure scaling; pay-per-request pricing aligns costs with actual usage, instant scaling from zero to thousands of concurrent executions. Built-in multi-AZ high availability and fault tolerance, native integration with AWS services through IAM roles, fast deployment cycles with minimal operational overhead, no idle costs when not processing requests.
- Limitations: 15-minute maximum execution time per request (synchronous), cold start latency of 0.5-3 seconds impacts first request performance, 10 GB memory limit per function, stateless architecture requires external storage for session management, WebSocket support requires API Gateway WebSocket APIs with additional configuration, limited runtime customization compared to container or VM approaches.

Deployment Pattern 3 - Amazon ECS (Containerized)

Amazon Elastic Container Service (ECS) with API gateway orchestrates Docker containers running MCP server applications. ECS supports both AWS Fargate for serverless container execution and EC2 launch types for more control. The architecture diagram illustrates the "Public front door + private MCP runtime" pattern, where Amazon API Gateway is the only internet-facing endpoint, and enforces OAuth authorization. API Gateway then reaches into the VPC using VPC link v2 to

call an internal Application Load Balancer. The private ALB forwards an ECS service running on the remote MCP server.

Sample implementation: https://github.com/aws-samples/sample-mcp-deployment-patterns/tree/main/deploy-ecs



Architecture Characteristics

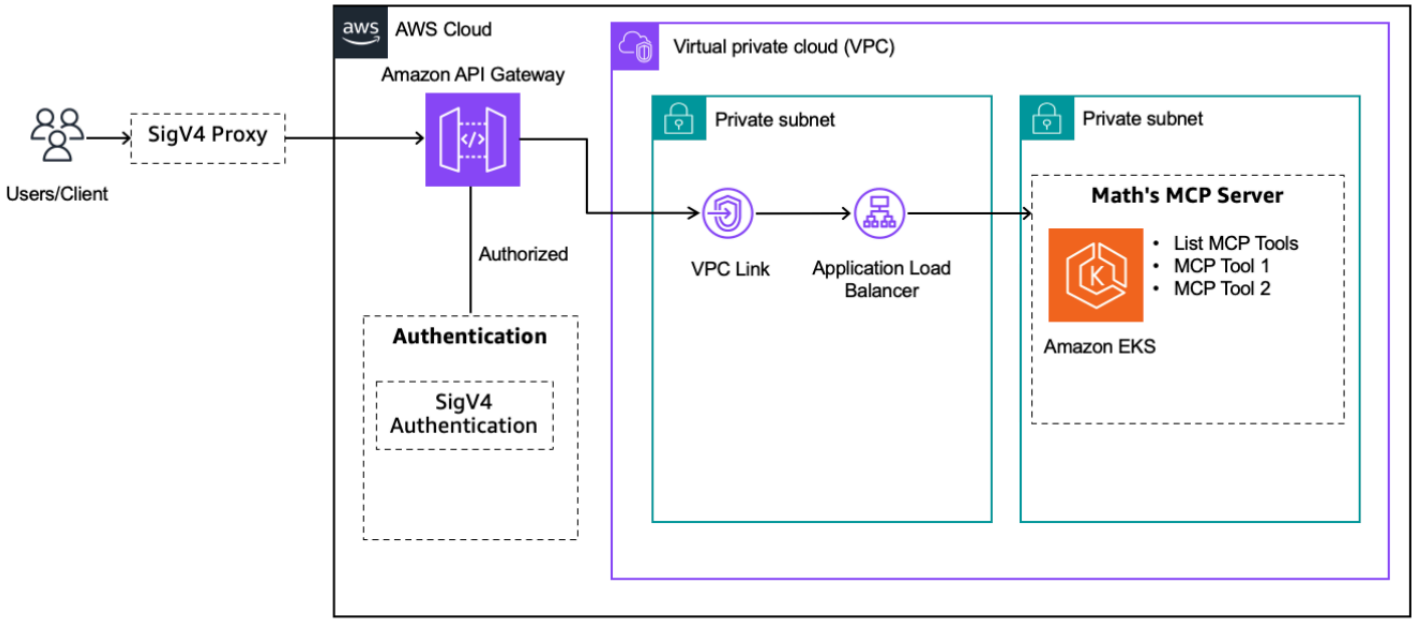
- Pros: Backend stays private: ALB and ECS are not internet-exposed; only API Gateway is public. Centralized security controls at API Gateway, Centralized throttling, quotas, and logging management.
- Limitations: API Gateway has default Timeouts and Payload limits. If any of the MCP tools are taking longer, then need to make the MCP tool async, or use Rest API and raise timeouts beyond 29s. If there are tools designed to run for a long time, then need to remove API Gateway and use public Application Load Balancer instead.

Deployment Pattern 4 - Amazon Elastic Kubernetes Service

Amazon Elastic Kubernetes Service (EKS) delivers managed Kubernetes control planes for running containerized MCP servers with advanced orchestration capabilities. EKS enables sophisticated traffic management, multi-tenant isolation through namespaces, and integration with the rich Kubernetes ecosystem. Organizations with existing Kubernetes expertise or complex microservices

architectures benefit most from this pattern despite its higher complexity and cost. The proposed architecture uses API Gateway with IAM (SigV4) authorization as the public entry point and integrate privately to an internal ALB via VPC Link v2. Run the MCP streamable HTTP endpoint on EKS behind the internal ALB. This pattern keeps the MCP server private, leverages AWS-native authorization and is best for AWS-controlled clients.

Sample implementation: https://github.com/aws-samples/sample-mcp-deployment-patterns/tree/main/deploy-eks



Architecture Characteristics

- Pros: The API Gateway + Private ALB + IAM SigV4 pattern provides great security isolation for EKS workloads by keeping all Kubernetes infrastructure completely private—no public load balancers or endpoints—while leveraging AWS-native IAM authentication that eliminates OAuth token services and credential proxies entirely. For EKS specifically, it delivers multi-tenancy isolation flexibility through fine-grained permissions scoped to paths/methods/accounts, enabling workload isolation across namespaces/teams without complex network policies or shared ALB rules.
- Limitations: SigV4 authentication creates friction for general internet clients lacking AWS credentials or SigV4 signing capability—most off-the-shelf MCP tools expect OAuth/bearer tokens, requiring a SigV4 proxy layer for third-party compatibility; API Gateway HTTP APIs impose hard limits (30s timeout, 10MB payload) that break long-running MCP tool calls.

Deployment Pattern 5 - Amazon EC2 (Persistent Server)

Amazon Elastic Compute Cloud (EC2) provides virtual server instances where MCP servers run traditional long-running processes. Organizations gain complete control over the operating system, installed software, and runtime environment. This pattern suits scenarios requiring persistent connections, custom runtime configurations, or migrations from on-premises infrastructure. The trade-off is higher operational overhead for managing instances, patching, and scaling.

Architecture Characteristics

- Pros: Full OS and runtime control with no execution time limits, native WebSocket and SSE support without gateway timeout constraints, zero cold starts with predictable performance, cost-efficient at sustained high throughput with reserved Instances, familiar operational model for teams migrating from on-premises environments.
- Limitations: Highest operational overhead — OS patching, AMI maintenance, and security hardening are fully self-managed. Scaling via Auto Scaling Groups is slower (2–5 min) compared to serverless patterns. Instances accrue costs 24/7 regardless of traffic. Multi-AZ high availability requires explicit configuration. Long-lived access keys on instances are a common security anti-pattern.

Best practices for MCP deployments

The AWS Well-Architected Framework provides a set of proven principles for building secure, reliable, and efficient cloud workloads. The following practices are most relevant when deploying MCP servers in production.

- **Apply least-privilege access** - Every component — whether a function, container, or instance — should have only the permissions it needs for its specific task. Avoid wildcard resource permissions. Use short-lived credentials wherever possible and never store long-lived credentials on compute resources.
- **Protect data at rest and in transit** -Encrypt all stored data using customer-managed or platform-managed keys depending on your compliance requirements. Enforce TLS 1.2 or higher for all network communication. Secrets should be stored in a managed secrets service with automatic rotation — never hardcoded or stored in environment variables unencrypted.
- **Keep compute private** - Place all MCP servers behind a load balancer in private subnets. Only the load balancer or API entry point should be internet-facing. Use private endpoints for internal service-to-service traffic to avoid traversing the public internet.
- **Design for failure** - Deploy across at least two availability zones. Implement circuit breakers and explicit timeouts on all downstream calls — never rely on platform defaults. Applications should degrade gracefully when a dependency is unavailable rather than failing entirely.
- **Enable threat detection from day one** - Turn on audit logging for all API activities across your accounts. Use automated threat detection to analyze logs for anomalous behavior. Centralize security findings into a single dashboard and define incident response runbooks before an incident occurs.
- **Right-size before you scale** - Start with a conservative allocation and tune based on observed metrics. Over-provisioned resources waste money and energy. Use automated recommendations after a period of baseline operation rather than guessing upfront.
- **Define infrastructure as code** - Never create or modify production resources manually. All compute, networking, IAM, and monitoring resources should be version-controlled, peer-reviewed, and deployed through automated pipelines. This approach is designed to provide repeatability, auditability, and safe rollbacks.
- **Design long-running operations asynchronously** - MCP tools that take more than a few seconds to complete should be decoupled from the synchronous request path. Accept the request, queue the work, return a job ID, and provide a polling or callback mechanism. This avoids gateway timeouts and improves user experience.

- **Tag every resource** - Apply consistent tags — at minimum environment, application, team, and cost centre — to all resources. Enforce tagging policy through automated rules. Without tagging, cost attribution and compliance reporting become impractical at scale.
- **Prefer managed services over self-managed infrastructure** - Managed and serverless compute eliminates idle resource consumption and reduce operational burden. When evaluating compute options, only choose self-managed infrastructure when there is a specific capability requirement that managed services cannot satisfy.

Deployment Considerations

Selecting the appropriate deployment pattern depends on your specific requirements, operational capabilities, and workload characteristics. Refer to the following decision matrix for different patterns:

Considera tion	Amazon Bedrock AgentCore	Lambda + API Gateway	ECS	EKS	EC2
Best For	Production MCP, AWS-native	Variable traffic, event-driven	Container ized apps, moderate scale	Complex microserv ices, multi-tenant	Persistent connections, full control
Management Overhead	Minimal	Minimal	Medium	Very High	High
Complexity	Low	Low	Medium	High	Medium
Scaling	Automatic	Automatic	Automatic	Automatic (advanced)	Manual/ASG
Cost (low traffic)	Low	Low	Medium	Highest	Medium
Cost (high traffic)	Low	Medium	Medium	Medium	Lowest
Cold Start	fast cold starts for real-time interactions	0.5-3s	<30s (Fargate)	None	None
Max Request Duration	15 mins for synchrono us request, 8 hours for	15 min	Unlimited	Unlimited	Unlimited

	asynchronous requests				
WebSocket Support	Limited (For browser interaction)	Limited	Native	Native	Native
Deployment Speed	Fast	Fast	Medium	Slow	Slow
Portability	AWS-only	AWS-specific	High	Highest	Low
Monitoring	Built-in	Built-in	Integrated	Rich ecosystem	Custom
Multi-tenancy	Service-level	Function-level	Service-level	Namespace-level	Custom
Customization	Limited	Limited	High	Full	Full
Auth Support	SigV4, OAuth and JWT validation	Rich (Through API Gateway)	Rich (Through API Gateway)	Rich (Through API Gateway)	Rich (Through API Gateway or ALB)

Next steps

Deploying MCP servers on AWS provides organizations with flexible, scalable, and secure infrastructure for AI-powered applications. This prescriptive guidance presented different deployment patterns, each optimized for different operational requirements, cost constraints, and scaling needs.

Next Steps

Organizations beginning MCP deployment on AWS should follow a structured approach. First, evaluate requirements by assessing traffic patterns, operational capabilities, and scaling needs against the pattern selection criteria. Consider current infrastructure and team expertise while selecting the MCP deployment pattern. Start small by beginning with Lambda for rapid prototyping or ECS for production-ready deployments that balance simplicity with flexibility.

Implement monitoring from day one by setting up CloudWatch dashboards, alarms, and log aggregation before production traffic begins. Follow AWS Well-Architected Framework security best practices including least-privilege IAM policies, VPC isolation, encryption at rest and in transit, and comprehensive audit logging. Gather metrics during initial operations and optimize based on actual usage patterns rather than assumptions. Consider hybrid approaches using multiple patterns for different MCP server types based on specific requirements such as Lambda for simple read-only servers and EKS for complex stateful servers.

Reference Implementation

The complete reference implementation with working code examples, deployment scripts, and detailed instructions for all four deployment patterns is available at: https://github.com/aws-samples/sample-mcp-deployment-patterns/

Resources

AWS service documentation

- Strands Agents documentation at: https://strandsagents.com/docs/user-guide/quickstart/overview/
- Get started with Amazon Bedrock AgentCore at https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/agentcore-get-started-toolkit.html
- AWS Lambda documentation at https://docs.aws.amazon.com/lambda/ covers function creation, triggers, and runtime environments.
- Amazon API Gateway documentation at https://docs.aws.amazon.com/apigateway/ explains REST and HTTP API creation.
- https://github.com/aws-solutions-library-samples/guidance-for-deploying-model-context-protocol-servers-on-aws
- Amazon EC2 documentation at https://docs.aws.amazon.com/ec2/ details instance types, networking, and management.
- Amazon ECS documentation at https://docs.aws.amazon.com/ecs/ covers containerized deployments.
- Amazon EKS documentation at https://docs.aws.amazon.com/eks/ explains Kubernetes on AWS.
- Amazon ECR documentation at https://docs.aws.amazon.com/ecr/ covers container image management. AWS Fargate documentation at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html explains serverless container compute.
- Amazon CloudWatch documentation at https://docs.aws.amazon.com/cloudwatch/ covers monitoring and logging.
- AWS Secrets Manager documentation at https://docs.aws.amazon.com/secretsmanager/ explains secure credential management.
- AWS Well-Architected Framework: https://aws.amazon.com/architecture/well-architected/ provides design principles and best practices.
- Amazon Bedrock AgentCore samples: https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/01-tutorials/01-AgentCore-runtime/02-hosting-MCP-server

MCP Resources

- The Model Context Protocol specification at https://modelcontextprotocol.io/specification/latest provides the authoritative protocol definition. Official SDK implementations:
- TypeScript SDK at https://github.com/modelcontextprotocol/typescript-sdk
- Python SDK at https://github.com/modelcontextprotocol/python-sdk
- Example MCP servers at https://github.com/modelcontextprotocol/servers demonstrate common integration patterns.
- Deploy MCP Servers on AgentCore Runtime: https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/runtime-mcp.html
- Infrastructure as Code
- AWS CloudFormation at https://aws.amazon.com/cloudformation/ enables template-based infrastructure provisioning.
- AWS CDK at https://aws.amazon.com/cdk/ provides programmatic infrastructure definition.
- eksctl at https://eksctl.io/ streamlines EKS cluster creation. Container and Kubernetes Resources

Monitoring and observability

- Amazon CloudWatch Container Insights: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/ContainerInsights.html

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 link.

Date	Description
28 July, 2026	Initial publication