Hosting MCP Servers & Agents on Amazon Bedrock AgentCore

Runtime, Gateway, and Local → Cloud Workflows

Agenda

1. Bedrock overview
2. AgentCore concepts: Runtime, Identity, Gateway
3. Why AgentCore (vs basic Bedrock calls)
4. Local testing → production deployment

- Hosting MCP servers & tools

What is Amazon Bedrock?

- Managed enterprise platform for building with foundation models
- Provides model hosting, versioning, evals, and safety guardrails
- Integrated features: Knowledge Bases (RAG), workflows, prompt mgmt
- Secure, private deployment with no data leaving your AWS boundary

What is MCP (Model Context Protocol)?

- Open protocol for structured, tool-based interaction with LLMs
- Standardizes how tools are described, discovered, and invoked
- Defines transport (stdio / WebSocket / streamable HTTP) and message schema
- Allows agents to call tools safely, consistently, and across ecosystems

MCP & Tools — How They Fit Together

- Tools are *functions*; MCP defines *how* those functions are exposed
- MCP tools include: name, schema, description, parameters, return types
- Agent frameworks (Strands, LangGraph, CrewAI) consume MCP tools consistently
- AgentCore Runtime & Gateway rely on MCP contracts for compatibility

The Challenge: From POC → Production

- POCs are relatively easy: “vibe code” + local demos
- Production is hard: security, auth, scaling, monitoring
- Tools can multiply fast → M×N integrations
- Multiple teams want to reuse tools safely

Introducing AgentCore

- Fully managed agent hosting platform
- Components: Runtime, Identity, Memory, Gateway
- Bring any framework, any model
- Handles the undifferentiated heavy lifting

AgentCore Runtime

- Serverless hosting for agents & MCP servers (and more)
- True session isolation (micro-VM per session, built on firecracker)
- Supports streaming + async tasks out of the box, something Bedrock didn't (long-running tasks up to 8 hrs)
- Works with any Model and any agent framework

Key Runtime Benefits

- Zero infrastructure management
- Deploy via CLI or SDK (configure → launch → invoke)
- Only pay for active compute time
- Automatic compatibility with observability stack

The Gateway Problem (Pre-AgentCore)

- MCP servers require hand-built JSON-RPC plumbing
- Managing OAuth inbound + outbound auth
- Versioning, patching, scaling dozens of MCP servers
- Hard to share tools across teams safely

Introducing AgentCore Gateway

- Converts APIs, Lambdas, and services → MCP tools
- Centralized, secure tool layer
- Semantic tool search (avoid context blow)
- Built-in OAuth for inbound + outbound access

Why Gateway Matters

- Scale tools from 3 → 300+ safely
- **Predictable Performance**: Reduce model performance degradation caused by tool
  overload and context saturation
- **Governance**: control which clients use which tools
- **Serverless**: unlimited gateways, pay per request

API vs MCP — When to Use Which

Use API directly when:

- Deterministic, simple request/response
- No tool orchestration needed

Use MCP (via Gateway) when:

- You want tool discovery & semantic matching
- You want agents, not developers, choosing tools
- You want standardized contracts across teams

Local Development Workflow

- Run Strands / LangGraph agents locally
- Test tools with MCP Inspector
- Deploy to Runtime when stable
- Same code, zero changes required

Deploying to Runtime (High-Level Steps)

1. `agentcore configure`
2. Autogenerate Dockerfile
3. Build → ECR → Runtime endpoint
4. Invoke with CLI / SDK / app

Hosting MCP Servers on Runtime

- Python FastMCP or TypeScript MCP supported
- Stateless HTTP mode for cloud scaling
- Deploy like any Runtime agent
- Test via MCP Inspector or remote client

Authentication (Inbound & Outbound)

- Inbound: validate JWT/OAuth tokens
- Outbound: securely call external APIs
- Supports Cognito, Auth0, Okta, etc.
- Credentials stored & rotated automatically

Semantic Tool Search & Lazy Loading

- LLM queries tools by meaning, not raw names
- Lazy loading: tools are fetched only when relevant
- Prevents context saturation from large tool libraries
- Improves accuracy & reduces hallucination

Common Pitfalls

- Too many tools loaded eagerly → context saturation
- Missing or vague tool descriptions → poor matching
- Agents may forget to call tools without scaffolding
- No lazy loading = degraded performance & higher cost

Testing Strategies

- Local MCP Inspector
- Local Runtime via `--local` flag
- Cloud: low-cost invocations
- Use CloudWatch GenAI Observability for traces

Costs & Practical Considerations

- Runtime: pay only for active compute time
- Gateway: no charge for gateways; requests only
- Model inference: standard Bedrock pricing
- Easy to scale down (serverless everything)

Putting It All Together

- Build locally (Strands / LangGraph / MCP)
- Test MCP tools locally
- Deploy agents & MCP servers to Runtime
- Expose enterprise APIs via Gateway
- Use semantic search + OAuth for production safety

Workshop Flow

- Part 1: Presentation (this deck)
- Part 2: Notebook demo
    - Configure Runtime
    - Deploy agent
    - Deploy MCP server
    - Test via MCP Inspector & remote client
    - Explore Gateway concepts

Q&A

Ask anything about Runtime, Gateway, MCP, deployment, auth, or real-world use cases.