Developer Guide

Amazon Bedrock AgentCore

Amazon Bedrock AgentCore: Developer Guide

Table of Contents

What is Amazon Bedrock AgentCore?

Amazon Bedrock AgentCore enables you to deploy and operate highly effective agents securely, at scale using any framework and model. With Amazon Bedrock AgentCore, developers can accelerate AI agents into production with the scale, reliability, and security, critical to real-world deployment. AgentCore provides tools and capabilities to make agents more effective and capable, purpose-built infrastructure to securely scale agents, and controls to operate trustworthy agents. Amazon Bedrock AgentCore services are composable and work with popular open-source frameworks and any model, so you don’t have to choose between open-source flexibility and enterprise-grade security and reliability.

Services in Amazon Bedrock AgentCore

Amazon Bedrock AgentCore includes the following modular Services that you can use together or independently:

Amazon Bedrock AgentCore Runtime

AgentCore Runtime is a secure, serverless runtime purpose-built for deploying and scaling dynamic AI agents and tools using any open-source framework including LangGraph, CrewAI, and Strands Agents, any protocol, and any model. Runtime was built to work for agentic workloads with industry-leading extended runtime support, fast cold starts, true session isolation, built-in identity, and support for multi-modal payloads. Developers can focus on innovation while Amazon Bedrock AgentCore Runtime handles infrastructure and security—accelerating time-to-market

Amazon Bedrock AgentCore Identity

AgentCore Identity provides a secure, scalable agent identity and access management capability accelerating AI agent development. It is compatible with existing identity providers, eliminating needs for user migration or rebuilding authentication flows. AgentCore Identity's helps to minimize consent fatigue with a secure token vault and allows you to build streamlined AI agent experiences. Just-enough access and secure permission delegation allow agents to securely access AWS resources and third-party tools and services.

Amazon Bedrock AgentCore Memory

AgentCore Memory makes it easy for developers to build context aware agents by eliminating complex memory infrastructure management while providing full control over what the AI agent remembers. Memory provides industry-leading accuracy along with support for both short-term memory for multi-turn conversations and long-term memory that can be shared across agents and sessions.

Amazon Bedrock AgentCore Code Interpreter

AgentCore Code Interpreter tool enables agents to securely execute code in isolated sandbox environments. It offers advanced configuration support and seamless integration with popular frameworks. Developers can build powerful agents for complex workflows and data analysis while meeting enterprise security requirements.

Amazon Bedrock AgentCore Browser

AgentCore Browser tool provides a fast, secure, cloud-based browser runtime to enable AI agents to interact with websites at scale. It provides enterprise-grade security, comprehensive observability features, and automatically scales— all without infrastructure management overhead.

Amazon Bedrock AgentCore Gateway

Amazon Bedrock AgentCore Gateway provides a secure way for agents to discover and use tools along with easy transformation of APIs, Lambda functions, and existing services into agent-compatible tools. Gateway eliminates weeks of custom code development, infrastructure provisioning, and security implementation so developers can focus on building innovative agent applications.

Amazon Bedrock AgentCore Observability

AgentCore Observability helps developers trace, debug, and monitor agent performance in production through unified operational dashboards. With support for OpenTelemetry compatible telemetry and detailed visualizations of each step of the agent workflow, AgentCore enables developers to easily gain visibility into agent behavior and maintain quality standards at scale.

Common use cases for Amazon Bedrock AgentCore

- **Equip agents with built-in tools and capabilities**

Leverage built-in tools (browser automation and code interpretation) in your agent. Enable agents to seamlessly integrate with internal and external tools and resources. Create agents that can remember interactions with your agent users.

- **Deploy securely at scale**

  Securely deploy and scale dynamic AI agents and tools, regardless of framework, protocol, or model choice without managing any underlying resources with seamless agent identity and access management.
- **Test and monitor agents**

  Gain deep operational insights with real-time visibility into agents' usage and operational metrics such as token usage, latency, session duration, and error rates.

Are you a first-time Amazon Bedrock AgentCore user?

If you are a first-time user of Amazon Bedrock AgentCore, we recommend that you begin by reading the following sections:

- Host agent or tools with Amazon Bedrock AgentCore Runtime
- Add memory to your AI agent
- Use Amazon Bedrock AgentCore built-in tools to interact with your applications
- Amazon Bedrock AgentCore Gateway: Securely connect tools and other resources to your Gateway

For code examples, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/.

Pricing for Amazon Bedrock AgentCore

Amazon Bedrock AgentCore offers flexible, consumption-based pricing with no upfront commitments or minimum fees. For more information, see Amazon Bedrock AgentCore pricing.

AWS Regions

Amazon Bedrock AgentCore is supported in the following AWS Regions:

- US East (N. Virginia)

- US West (Oregon)
- Europe (Frankfurt)
- Asia Pacific (Sydney)

Host agent or tools with Amazon Bedrock AgentCore Runtime

Amazon Bedrock AgentCore Runtime provides a secure, serverless and purpose-built hosting environment for deploying and running AI agents or tools. It offers the following benefits:

Framework agnostic

> Runtime lets you transform any local agent code to cloud-native deployments with a few lines of code no matter the underlying framework. Works seamlessly with popular frameworks like LangGraph, Strands, and CrewAI. You can also leverage it with custom agents that don't use a specific framework.

Model flexibility

> Runtime works with any Large Language Model, such as models offered by Amazon Bedrock, Anthropic Claude, Google Gemini, and OpenAI.

Protocol support

> Runtime lets agents communicate with other agents and tools via Model Context Protocol (MCP).

Extended execution time

> Runtime supports both real-time interactions and long-running workloads up to 8 hours, enabling complex agent reasoning and asynchronous workloads that may involve multi-agent collaboration or extended problem-solving sessions.

Enhanced payload handling

> Runtime can process 100MB payloads enabling seamless processing of multiple modalities (text, images, audio, video), with rich media content or large datasets.

Session isolation

> In Runtime, each user session runs in a dedicated microVM with isolated CPU, memory, and filesystem resources. This helps create complete separation between user sessions, safeguarding stateful agent reasoning processes and helps prevent cross-session data contamination. After session completion, the entire microVM is terminated and memory is sanitized, delivering deterministic security even when working with non-deterministic AI processes.

Consumption-based pricing model

Runtime implements consumption-based pricing that charges only for resources actually consumed. Unlike allocation-based models that require pre-selecting resources, Runtime dynamically provisions what's needed without requiring right-sizing. The service aligns CPU billing with actual active processing - typically eliminating charges during I/O wait periods when agents are primarily waiting for LLM responses - while continuously maintaining your session state.

Built-in authentication

Runtime, powered by Amazon Bedrock AgentCore Identity, assigns distinct identities to AI agents and seamlessly integrates with your corporate identity provider such as Okta, Microsoft Entra ID, or Amazon Cognito, enabling your end users to authenticate into only the agents they have access to. In addition, Runtime lets outbound authentication flows to securely access third-party services like Slack, Zoom, and GitHub - whether operating on behalf of users or autonomously (using either OAuth or API keys).

Agent-specific observability

Runtime provides specialized built-in tracing that captures agent reasoning steps, tool invocations, and model interactions, providing clear visibility into agent decision-making processes, a critical capability for debugging and auditing AI agent behaviors.

Unified set of agent-specific capabilities

Runtime is delivered through a single, comprehensive SDK that provides streamlined access to the complete Amazon Bedrock AgentCore capabilities including Memory, Tools, and Gateway. This integrated approach eliminates the integration work typically required when building equivalent agent infrastructure from disparate components.

How it works

The AgentCore Runtime handles scaling, session management, security isolation, and infrastructure management, allowing you to focus on building intelligent agent experiences rather than operational complexity. By leveraging the features and capabilities described here, you can build, deploy, and manage sophisticated AI agents that deliver value to your users while helping to maintain enterprise-grade security and reliability.

Key components

Agent runtime

An AgentCore Runtime is the foundational component that hosts your AI agent or tool code. It represents a containerized application that processes user inputs, maintains context, and executes actions using AI capabilities. When you create an agent, you define its behavior, capabilities, and the tools it can access. For example, a customer support agent might answer product questions, process returns, and escalate complex issues to human representatives.

You can build and deploy agents to AgentCore Runtime using the AgentCore Python SDK or directly through AWS SDKs. With the Python SDK, you can define your agent using popular frameworks like LangGraph, CrewAI, or Strands Agents. The SDK handles infrastructure complexities, allowing you to focus on the agent's logic and capabilities.

Each Agent Runtime:

- Has a unique identity
- Is versioned to support controlled deployment and updates

Versions

Each AgentCore Runtime maintains immutable versions that capture a complete snapshot of the configuration at a specific point in time:

- When you create an Agent Runtime, Version 1 (V1) is automatically created
- Each update to configuration (container image, protocol settings, network settings) creates a new version
- Each version contains all necessary configuration needed for execution

This versioning system provides reliable deployment history and rollback capabilities.

Endpoints

Endpoints provide addressable access points (i.e., aliases) to specific versions of your AgentCore Runtime. Each endpoint:

- Has a unique ARN for invocation

- References a specific version of your Agent Runtime
- Provides stable access to your agent even as you update implementations

Key endpoint details:

- The "DEFAULT" endpoint is automatically created when you call `CreateAgentRuntime` and points to the latest version
- When you update your Agent Runtime, a new version is created but the DEFAULT endpoint automatically updates to reference it
- Custom endpoints can be created via `CreateAgentRuntimeEndpoint` for different environments (dev, test, prod)
- When a user makes a request to an endpoint, the request is resolved to the specific agent version referenced by that endpoint

Endpoints have distinct lifecycle states:

- CREATING - Initial state during endpoint creation
- CREATE_FAILED - Indicates creation failure due to permissions or other issues
- READY - Endpoint is operational and accepting requests
- UPDATING - Endpoint is being modified to reference a new version
- UPDATE_FAILED - Indicates update operation failure

You can update endpoints without downtime, allowing for seamless version transitions and rollbacks.

Sessions

Sessions represent individual interaction contexts between users and your AgentCore Runtime. Each session:

- Is identified by a unique `runtimeSessionId` provided by your application, or by the Runtime itself in the first invocation if the `runtimeSessionId` is left empty
- Runs in a dedicated microVM with completely isolated CPU, memory, and filesystem resources
- Preserves context across multiple interactions within the same conversation

- Can persist for up to 8 hours of total runtime

Session states include:

- **Active** - Currently processing a request or executing background tasks
- **Idle** - Not processing any requests but maintaining context while waiting for next interaction
- **Terminated** - Session ended due to inactivity (15 minutes), reaching maximum lifetime (8 hours), or being deemed unhealthy

Important session characteristics:

- After session termination, the entire microVM is terminated and memory is sanitized
- A subsequent request with the same `runtimeSessionId` after termination will create a new execution environment
- Session isolation prevents cross-session data contamination and ensures security
- Session state is ephemeral and should not be used for long-term durability (use Amazon Bedrock AgentCore Memory for context durability)

This complete isolation between sessions is crucial for enterprise security, particularly when dealing with non-deterministic AI processes.

Authentication and security

Inbound authentication

Inbound Auth, powered by Amazon Bedrock AgentCore Identity, controls who can access and invoke your agents or tools in AgentCore Runtime:

Authentication methods

- **AWS IAM** (SigV4): Uses AWS credentials for identity verification
- **OAuth 2.0**: Integrates with external identity providers

OAuth configuration options

- **Discovery URL**: Your identity provider's OpenID Connect discovery endpoint

- **Allowed Audiences**: List of valid audience values your tokens should contain
- **Allowed Clients**: List of client identifiers that can access this agent

Authentication flow

1. End users authenticate with your identity provider (Amazon Cognito, Okta, Microsoft Entra ID)
2. Your client application receives a bearer token after successful authentication
3. The client passes this token in the authorization header when invoking the agent
4. AgentCore Runtime validates the token with the authorization server
5. If valid, the request is processed; if invalid, it's rejected

This ensures only authenticated users with proper authorization can access your agents.

Outbound authentication

Outbound Auth, powered by Amazon Bedrock AgentCore Identity, lets your agents hosted on AgentCore Runtime securely access third-party services:

Authentication methods

- **OAuth**: For services supporting OAuth flows
- **API Keys**: For services using key-based authentication

Authentication modes

- **User-delegated**: Acting on behalf of the end user with their credentials
- **Autonomous**: Acting independently with service-level credentials

Supported services

- Enterprise systems (Slack, Zoom, GitHub, etc.)
- AWS services
- Custom APIs and data sources

Amazon Bedrock AgentCore Identity manages these credentials securely, preventing credential exposure in your agent code or logs.

Additional features

Asynchronous processing

AgentCore Runtime supports long-running workloads through:

- Background task handling for operations that exceed request/response cycles
- Automatic status tracking via the `/ping` endpoint
- Support for operations up to 8 hours in duration

Streaming responses

Agents can stream partial results as they become available rather than waiting for complete processing. This lets you provide a more responsive user experience, especially for operations that generate large amounts of content or take significant time to complete.

Protocol support

Runtime supports multiple communication protocols:

- HTTP for simple request/response patterns
- Model Context Protocol (MCP) for standardized agent-tool interactions

Implementation overview

Here's how to get started with the AgentCore Runtime:

Prepare your agent or tool code

- Define your agent logic using any AI framework or custom code
- Add the required HTTP endpoints using the AgentCore SDK or custom implementation
- Package dependencies in a requirements.txt file

Deploy your agent or tool

- Build and push a container image to Amazon ECR directly or via the AgentCore SDK

- Create an AgentCore Runtime using the container image
- The initial version (V1) and DEFAULT endpoint are created automatically

Invoke your agent or tool

- Generate a unique session ID for each user conversation
- Call the `InvokeAgentRuntime` operation with your agent's ARN and session ID
- Pass user input in the request payload

Manage and observe sessions, and make updates

- Use the same session ID for follow-up interactions to maintain context
- Review logs, traces, and observability metrics
- Deploy updates by modifying your AgentCore Runtime (creates new versions)
- Control rollout by updating endpoints to point to new versions

Understanding the AgentCore Runtime service contract

The AgentCore Runtime service contract defines the standardized communication protocol that your agent application must implement to integrate with the Amazon Bedrock agent hosting infrastructure. This contract ensures seamless communication between your custom agent code and AWS's managed hosting environment.

Topics

Supported protocols

The AgentCore Runtime service contract supports two communication protocols:

- **HTTP Protocol**: Direct REST API endpoints for traditional request/response patterns
- **MCP Protocol**: Model Context Protocol for tools and agent servers

HTTP protocol contract

Container requirements

Your agent must be deployed as a containerized application meeting these specifications:

- **Host**: `0.0.0.0`
- **Port**: `8080` - Standard port for HTTP-based agent communication
- **Platform**: ARM64 container - Required for compatibility with the AgentCore Runtime environment

Path requirements

/invocations - POST

This is the primary agent interaction endpoint with JSON input and JSON/SSE output.

Purpose

Receives incoming requests from users or applications and processes them through your agent's business logic

Use cases

The `/invocations` endpoint serves several key purposes:

- Direct user interactions and conversations
- API integrations with external systems
- Batch processing of multiple requests
- Real-time streaming responses for long-running operations

Example Request format

```
Content-Type: application/json

{
  "prompt": "What's the weather today?"
}
```

Response formats

Your agent can respond using either of the following formats depending on the use case:

JSON response (non-streaming)

Purpose

Provides complete responses for requests that can be processed quickly

Use cases

JSON responses are ideal for:

- Simple question-answering scenarios
- Deterministic computations
- Quick data lookups
- Status confirmations

Example JSON response format

```
Content-Type: application/json

{
  "response": "Your agent's response here",
  "status": "success"
}
```

SSE response (streaming)

Server-Sent Events (SSE) let you deliver real-time streaming responses. For full details, refer to the SSE specification.

Purpose

Enables incremental response delivery for long-running operations and improved user experience

Use cases

SSE responses are ideal for:

- Real-time conversational experiences
- Progressive content generation
- Long-running computations with intermediate results

- Live data feeds and updates

Example SSE response format

```
Content-Type: text/event-stream

data: {"event": "partial response 1"}
data: {"event": "partial response 2"}
data: {"event": "final response"}
```

/ping - GET

Purpose

Verifies that your agent is operational and ready to handle requests

Use cases

The `/ping` endpoint serves several key purposes:

- Service monitoring to detect and remediate issues
- Automated recovery through AWS's managed infrastructure

Response format

Returns a status code indicating your agent's health:

- **Content-Type**: `application/json`
- **HTTP Status Code**: `200` for healthy, appropriate error codes for unhealthy states

If your agent needs to process background tasks, you can indicate it with the `/ping` status. If the ping status is `HealthyBusy`, the runtime session is considered active.

Example Ping response format

```
{
  "status": "<status_value>",
  "time_of_last_update": <unix_timestamp>
}
```

status

`Healthy` - System is ready to accept new work

`HealthyBusy` - System is operational but currently busy with async tasks

time_of_last_update

Used to determine how long the system has been in its current state

MCP protocol contract

Protocol implementation requirements

Your MCP server must implement these specific protocol requirements:

- **Transport**: Stateless streamable-http only - Ensures compatibility with AWS's session management and load balancing
- **Session Management**: Platform automatically adds `Mcp-Session-Id` header for session isolation, servers must support stateless operation so as to not reject platform generated `Mcp-Session-Id` header

Container requirements

Your MCP agent must be deployed as a containerized application meeting these specifications:

- **Host**: `0.0.0.0`
- **Port**: `8000` - Standard port for MCP server communication (different from HTTP protocol)
- **Platform**: ARM64 container - Required for compatibility with AWS Amazon Bedrock AgentCore runtime environment

Path requirements

/mcp - POST

Purpose

Receives MCP RPC messages and processes them through your agent's tool capabilities, complete pass-through of `InvokeAgentRuntime` API payload with standard MCP RPC messages

Response format

JSON-RPC based request/response format, supporting both `application/json` and `text/event-stream` as response content-types

Use cases

The `/mcp` endpoint serves several key purposes:

- Tool invocation and management
- Agent capability discovery
- Resource access and manipulation
- Multi-step agent workflows

IAM Permissions for AgentCore Runtime

The following are IAM permissions you need to create an agent in an AgentCore Runtime and the execution role permissions that an agent needs to run in an AgentCore Runtime

Topics

Use Amazon Bedrock AgentCore

To use Amazon Bedrock AgentCore, you can attach the BedrockAgentCoreFullAccess AWS managed policy to your IAM user or IAM. role. This AWS managed policy grants broad permissions. We recommend creating a custom policy with only the permissions your application requires by copying the relevant statements and restricting the resources to your specific use case. To use the starter toolkit, you need additional permissions.

Use the starter toolkit

To use the Amazon Bedrock AgentCore starter toolkit, attach the following IAM policy to your IAM user or role. To change IAM permissions, see Change permissions for an IAM user.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "IAMRoleManagement",
            "Effect": "Allow",
            "Action": [
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:GetRole",
                "iam:PutRolePolicy",
                "iam:DeleteRolePolicy",
                "iam:AttachRolePolicy",
                "iam:DetachRolePolicy",
                "iam:TagRole",
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies"
            ],
            "Resource": [
                "arn:aws:iam::*:role/*BedrockAgentCore*",
                "arn:aws:iam::*:role/service-role/*BedrockAgentCore*"
            ]
        },
        {
            "Sid": "CodeBuildProjectAccess",
            "Effect": "Allow",
            "Action": [
                "codebuild:StartBuild",
                "codebuild:BatchGetBuilds",
                "codebuild:ListBuildsForProject",
                "codebuild:CreateProject",
                "codebuild:UpdateProject",
                "codebuild:BatchGetProjects"
            ],
            "Resource": [
                "arn:aws:codebuild:*:*:project/bedrock-agentcore-*",
                "arn:aws:codebuild:*:*:build/bedrock-agentcore-*"
            ]
        },
        {
            "Sid": "CodeBuildListAccess",
            "Effect": "Allow",
            "Action": [
                "codebuild:ListProjects"
            ],
            "Resource": "*"
        },
        {
            "Sid": "IAMPassRoleAccess",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/AmazonBedrockAgentCore*",
                "arn:aws:iam::*:role/service-role/AmazonBedrockAgentCore*"
            ]
        },
        {
            "Sid": "CloudWatchLogsAccess",
            "Effect": "Allow",
            "Action": [
                "logs:GetLogEvents",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws:logs:*:*:log-group:/aws/bedrock-agentcore/*",
                "arn:aws:logs:*:*:log-group:/aws/codebuild/*"
            ]
        },
        {
            "Sid": "S3Access",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket",
                "s3:CreateBucket",
                "s3:PutLifecycleConfiguration"
            ],
            "Resource": [
                "arn:aws:s3:::bedrock-agentcore-*",
                "arn:aws:s3:::bedrock-agentcore-*/*"
            ]
        },
        {
            "Sid": "ECRRepositoryAccess",
            "Effect": "Allow",
            "Action": [
                "ecr:CreateRepository",
                "ecr:DescribeRepositories",
                "ecr:GetRepositoryPolicy",
                "ecr:InitiateLayerUpload",
                "ecr:CompleteLayerUpload",
                "ecr:PutImage",
                "ecr:UploadLayerPart",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "ecr:ListImages",
                "ecr:TagResource"
            ],
            "Resource": [
                "arn:aws:ecr:*:*:repository/bedrock-agentcore-*"
            ]
        },
        {
            "Sid": "ECRAuthorizationAccess",
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken"
            ],
            "Resource": "*"
        }
    ]
}
```

Execution role for running an agent in AgentCore Runtime

To run agent or tool in AgentCore Runtime you need an AWS Identity and Access Management execution role. For information about creating an IAM role, see IAM role creation.

AgentCore Runtime execution role

The AgentCore Runtime execution role is an IAM role that AgentCore Runtime assumes to run an agent. Replace the following:

- *us-east-1* with the AWS Region that you are using
- *123456789012* with your AWS account ID
- *agentName* with the name of your agent. You'll need to decide the agent name before creating the role and AgentCore Runtime.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ECRImageAccess",
            "Effect": "Allow",
            "Action": [
                "ecr:BatchGetImage",
                "ecr:GetDownloadUrlForLayer"
            ],
            "Resource": [
                "arn:aws:ecr:us-east-1:123456789012:repository/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams",
                "logs:CreateLogGroup"
            ],
            "Resource": [
                "arn:aws:logs:us-east-1:123456789012:log-group:/aws/bedrock-agentcore/runtimes/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups"
            ],
            "Resource": [
                "arn:aws:logs:us-east-1:123456789012:log-group:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:us-east-1:123456789012:log-group:/aws/bedrock-
agentcore/runtimes/*:log-stream:*"
            ]
        },
        {
            "Sid": "ECRTokenAccess",
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken"
            ],
            "Resource": "*"
        },
        {
        "Effect": "Allow",
        "Action": [
            "xray:PutTraceSegments",
            "xray:PutTelemetryRecords",
            "xray:GetSamplingRules",
            "xray:GetSamplingTargets"
            ],
         "Resource": [ "*" ]
         },
         {
            "Effect": "Allow",
            "Resource": "*",
            "Action": "cloudwatch:PutMetricData",
            "Condition": {
                "StringEquals": {
                    "cloudwatch:namespace": "bedrock-agentcore"
                }
            }
        },
        {
            "Sid": "GetAgentAccessToken",
            "Effect": "Allow",
            "Action": [
                "bedrock-agentcore:GetWorkloadAccessToken",
                "bedrock-agentcore:GetWorkloadAccessTokenForJWT",
                "bedrock-agentcore:GetWorkloadAccessTokenForUserId"
            ],
            "Resource": [
              "arn:aws:bedrock-agentcore:us-east-1:123456789012:workload-
identity-directory/default",
              "arn:aws:bedrock-agentcore:us-east-1:123456789012:workload-
identity-directory/default/workload-identity/agentName-*"
            ]
        },
         {"Sid": "BedrockModelInvocation",
         "Effect": "Allow",
         "Action": [
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream"
              ],
        "Resource": [
            "arn:aws:bedrock:*::foundation-model/*",
            "arn:aws:bedrock:us-east-1:123456789012:*"
        ]
        }
    ]
}
```

AgentCore Runtime trust policy

The trust relationship for the AgentCore Runtime execution role should allow AgentCore Runtime to assume the role:

Replace the following:

- *us-east-1* with the AWS Region that you are using
- *123456789012* with your AWS account ID

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AssumeRolePolicy",
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock-agentcore.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "123456789012"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:123456789012:*"
            }
       }
    }
  ]
}
```

Getting started with Amazon Bedrock AgentCore Runtime

You can use the following tutorials to get started with Amazon Bedrock AgentCore Runtime.

The Amazon Bedrock AgentCore Starter Toolkit is a Command Line Interface (CLI) that simplifies the infrastructure setup for containerizing and deploying an agent to an AgentCore Runtime.

Topics

- Get started with the Amazon Bedrock AgentCore starter toolkit
- Getting started without the starter toolkit

Get started with the Amazon Bedrock AgentCore starter toolkit

This tutorial shows you how to use the Amazon Bedrock AgentCore starter toolkit to deploy an agent to an AgentCore Runtime.

The starter toolkit is a Command Line Interface (CLI) toolkit that you can use to deploy AI agents to an AgentCore Runtime. You can use the toolkit with popular Python agent frameworks, such as LangGraph or Strands Agents. This tutorial uses Strands Agents.

Topics

Prerequisites

Before you start, make sure you have:

- **AWS Account** with credentials configured. To configure your AWS credentials, see Configuration and credential file settings in the AWS CLI.
- **Python 3.10+** installed
- Boto3 installed
- **AWS Permissions**: To create and deploy an agent with the starter toolkit, you must have appropriate permissions. For information, see Use the starter toolkit.
- **Model access**: Anthropic Claude Sonnet 4.0 enabled in the Amazon Bedrock console. For information about using a different model with the Strands Agents see the *Model Providers* section in the Strands Agents SDK documentation.

Step 1: Enable observability for your agent

Amazon Bedrock AgentCore Observability helps you trace, debug, and monitor agents that you host in AgentCore Runtime. First enable CloudWatch Transaction Search by following the instructions at Enabling AgentCore runtime observability. To observe your agent, see View observability data for your Amazon Bedrock AgentCore agents.

Step 2: Install and create your agent

Upgrade `pip` to the latest version:

```
pip install --upgrade pip
```

Install the following required packages:

- **bedrock-agentcore** - The Amazon Bedrock AgentCore SDK for building AI agents
- **strands-agents** - The Strands Agents SDK
- **bedrock-agentcore-starter-toolkit** - The Amazon Bedrock AgentCore starter toolkit

```
pip install bedrock-agentcore strands-agents bedrock-agentcore-starter-toolkit
```

Create a source file for your agent code named `my_agent.py`. Add the following code:

```
from bedrock_agentcore import BedrockAgentCoreApp
from strands import Agent

app = BedrockAgentCoreApp()
agent = Agent()

@app.entrypoint
def invoke(payload):
    """Your AI agent function"""
    user_message = payload.get("prompt", "Hello! How can I help you today?")
    result = agent(user_message)
    return {"result": result.message}

if __name__ == "__main__":
    app.run()
```

Create `requirements.txt` and add the following:

```
bedrock-agentcore
strands-agents
```

Step 3: Test locally

Open a terminal window and start your agent with the following command:

```
python my_agent.py
```

Test your agent by opening another terminal window and enter the following command:

```
curl -X POST http://localhost:8080/invocations \
  -H "Content-Type: application/json" \
  -d '{"prompt": "Hello!"}'
```

Success: You should see a response like `{"result": "Hello! I'm here to help..."}`

In the terminal window that's running the agent, enter **`Ctrl+c`** to stop the agent.

Step 4: Deploy to AgentCore Runtime

Configure and deploy your agent to AWS using the starter toolkit. The toolkit automatically creates the IAM execution role, container image, and Amazon Elastic Container Registry repository needed to host the agent in AgentCore Runtime. By default the toolkit hosts the agent in an AgentCore Runtime that is in the `us-west-2` AWS Region.

Configure the agent. Use the default values:

```
agentcore configure -e my_agent.py
```

The configuration information is stored in a hidden file named `bedrock_agentcore.yaml`.

Host your agent in AgentCore Runtime:

```
agentcore launch
```

In the output from `agentcore launch` note the following:

- The Amazon Resource Name (ARN) of the agent. You need it to invoke the agent with the `InvokeAgentRuntime` operation.
- The location of the logs in Amazon CloudWatch Logs

If the deployment fails check for Common issues.

Test your deployed agent:

```
agentcore invoke '{"prompt": "tell me a joke"}'
```

If you see a joke in the response, your agent is now running in an AgentCore Runtime and can be invoked. If not, check for Common issues.

For other deployment options, see Deployment modes.

Step 5: Invoke your agent

You can invoke the agent using the AWS SDK InvokeAgentRuntime operation. To call `InvokeAgentRuntime`, you need the ARN of the agent that you noted in Step 4: Deploy to AgentCore Runtime. You can also get the ARN from the `bedrock_agentcore:` section of the `bedrock_agentcore.yaml` (hidden) file that the toolkit creates.

Use the following boto3 (AWS SDK) code to invoke your agent. Replace *`Agent ARN`* with the ARN of your agent. Make sure that you have `bedrock-agentcore:InvokeAgentRuntime` permissions.

Create a file named `invoke_agent.py` and add the following code:

```
import json
import uuid
import boto3

agent_arn = "Agent ARN"
prompt = "Tell me a joke"

# Initialize the AgentCore client
agent_core_client = boto3.client('bedrock-agentcore')

# Prepare the payload
payload = json.dumps({"prompt": prompt}).encode()

# Invoke the agent
response = agent_core_client.invoke_agent_runtime(
    agentRuntimeArn=agent_arn,
    payload=payload
)

content = []
```

```
for chunk in response.get("response", []):
    content.append(chunk.decode('utf-8'))
print(json.loads(''.join(content)))
```

Open a terminal window and run the code with the following command:

```
python invoke_agent.py
```

If succesful, you should see a joke in the response. If the call fails, check the logs that you noted in Step 4: Deploy to AgentCore Runtime.

If you plan on integrating your agent with OAuth, you can't use the AWS SDK to call `InvokeAgentRuntime`. Instead, make a HTTPS request to InvokeAgentRuntime. For more information, see Authenticate and authorize with Inbound Auth and Outbound Auth.

Step 6: Clean up

If you no longer want to host the agent in the AgentCore Runtime, use the AgentCore console or the DeleteAgentRuntime AWS SDK operation to delete the AgentCore Runtime.

Common issues

Common issues and solutions when getting started with the Amazon Bedrock AgentCore starter toolkit. For more troubleshooting information, see Troubleshoot AgentCore Runtime.

Permission denied errors

Verify your AWS credentials and permissions:

- Verify AWS credentials: `aws sts get-caller-identity`
- Check you have the required policies attached
- Review caller permissions policy for detailed requirements

Docker not found warnings

You can ignore this warning:

- **Ignore this!** Default deployment uses CodeBuild (no Docker needed)
- Only install Docker/Finch/Podman if you want to use `--local` or `--local-build` flags

Model access denied

Enable model access in the Bedrock console:

- Enable Anthropic Claude 4.0 in the Bedrock console
- Make sure you're in the correct AWS region (us-west-2 by default)

CodeBuild build error

Check build logs and permissions:

- Check CodeBuild project logs in AWS console
- Verify your caller permissions include CodeBuild access

Advanced options (optional)

The starter toolkit has advanced configuration options for different deployment modes and custom IAM roles. For more information, see Runtime commands for the starter toolkit.

Deployment modes

Choose the right deployment approach for your needs:

Default: CodeBuild + Cloud Runtime (RECOMMENDED)

Suitable for production, managed environments, teams without Docker:

```
agentcore launch  # Uses CodeBuild (no Docker needed)
```

Local Development

Suitable for development, rapid iteration, debugging:

```
agentcore launch --local  # Build and run locally (requires Docker/Finch/Podman)
```

Hybrid: Local Build + Cloud Runtime

Suitable for teams with Docker expertise needing build customization:

```
agentcore launch --local-build  # Build locally, deploy to cloud (requires Docker/
Finch/Podman)
```

Custom execution role

Use an existing IAM role:

```
agentcore configure -e my_agent.py --execution-role arn:aws:iam::111122223333:role/
MyRole
```

Getting started without the starter toolkit

You can create a AgentCore Runtime agent without the starter toolkit. Instead you can use a combination of command line tools to configure and deploy your agent to an AgentCore Runtime.

This tutorial shows how to deploy a custom agent without using the starter toollkit. A custom agent is an agent built without using the AgentCore Python SDK. In this tutorial, the custom agent is built using FastAPI and Docker. The custom agent follows the AgentCore Runtime requirements, meaning the agent must expose `/invocations` POST and `/ping` GET endpoints and be packaged in a Docker container. Amazon Bedrock AgentCore requires ARM64 architecture for all deployed agents.

> **Note**
>
> You can also use this approach for agents that you build with the AgentCore Python SDK.

Quick start setup

Enable observability for your agent

Amazon Bedrock AgentCore Observability helps you trace, debug, and monitor agents that you host in AgentCore Runtime. To observe an agent, first enable CloudWatch Transaction Search by following the instructions at Enabling AgentCore runtime observability.

Install uv

For this example, we'll use the uv package manager, though you can use any Python utility or package manager. To install uv on macOS:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

For installation instructions on other platforms, refer to the uv documentation.

Create your agent project

Setting up your project

1. Create and navigate to your project directory:

   ```
   mkdir my-custom-agent && cd my-custom-agent
   ```

2. Initialize the project with Python 3.11:

   ```
   uv init --python 3.11
   ```

3. Add the required dependencies (uv automatically creates a .venv):

   ```
   uv add fastapi 'uvicorn[standard]' pydantic httpx strands-agents
   ```

Agent contract requirements

Your custom agent must fulfill these core requirements:

- **`/invocations` Endpoint**: POST endpoint for agent interactions (REQUIRED)
- **`/ping` Endpoint**: GET endpoint for health checks (REQUIRED)
- **Docker Container**: ARM64 containerized deployment package

Project structure

Note: For convenience, the example below uses **FastAPI Server** as the Web server framework for handling requests.

Your project should have the following structure:

```
my-custom-agent/
### agent.py                 # FastAPI application
### Dockerfile               # ARM64 container configuration
### pyproject.toml           # Created by uv init
### uv.lock                  # Created automatically by uv
```

Complete strands agent example

Create `agent.py` in your project root with the following content:

Example agent.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Dict, Any
from datetime import datetime
from strands import Agent

app = FastAPI(title="Strands Agent Server", version="1.0.0")

# Initialize Strands agent
strands_agent = Agent()

class InvocationRequest(BaseModel):
    input: Dict[str, Any]

class InvocationResponse(BaseModel):
    output: Dict[str, Any]

@app.post("/invocations", response_model=InvocationResponse)
async def invoke_agent(request: InvocationRequest):
    try:
        user_message = request.input.get("prompt", "")
        if not user_message:
            raise HTTPException(
                status_code=400,
                detail="No prompt found in input. Please provide a 'prompt' key in the
 input."
            )

        result = strands_agent(user_message)
        response = {
            "message": result.message,
            "timestamp": datetime.utcnow().isoformat(),
            "model": "strands-agent",
        }

        return InvocationResponse(output=response)

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Agent processing failed:
 {str(e)}")

@app.get("/ping")
async def ping():
    return {"status": "healthy"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

This implementation:

- Creates a FastAPI application with the required endpoints
- Initializes a Strands agent for processing user messages
- Implements the `/invocations` POST endpoint for agent interactions
- Implements the `/ping` GET endpoint for health checks
- Configures the server to run on host `0.0.0.0` and port `8080`

Test locally

Testing your agent

1. Run the application:

   ```
   uv run uvicorn agent:app --host 0.0.0.0 --port 8080
   ```

2. Test the `/ping` endpoint (in another terminal):

   ```
   curl http://localhost:8080/ping
   ```

3. Test the `/invocations` endpoint:

```
curl -X POST http://localhost:8080/invocations \
  -H "Content-Type: application/json" \
  -d '{
    "input": {"prompt": "What is artificial intelligence?"}
  }'
```

Create dockerfile

Create `Dockerfile` in your project root with the following content:

Example Dockerfile

```
# Use uv's ARM64 Python base image
FROM --platform=linux/arm64 ghcr.io/astral-sh/uv:python3.11-bookworm-slim

WORKDIR /app

# Copy uv files
COPY pyproject.toml uv.lock ./

# Install dependencies (including strands-agents)
RUN uv sync --frozen --no-cache

# Copy agent file
COPY agent.py ./

# Expose port
EXPOSE 8080

# Run application
CMD ["uv", "run", "uvicorn", "agent:app", "--host", "0.0.0.0", "--port", "8080"]
```

This Dockerfile:

- Uses an ARM64 Python base image (required by Amazon Bedrock AgentCore)
- Sets up the working directory
- Copies the dependency files and installs dependencies
- Copies the agent code
- Exposes port 8080

- Configures the command to run the application

Build and deploy ARM64 image

Setup docker buildx

Docker buildx lets you build images for different architectures. Set it up with:

```
docker buildx create --use
```

Build for ARM64 and test locally

Building and testing your image

1. Build the image locally for testing:

   ```
   docker buildx build --platform linux/arm64 -t my-agent:arm64 --load .
   ```

2. Test locally with credentials (Strands agents need AWS credentials):

   ```
   docker run --platform linux/arm64 -p 8080:8080 \
     -e AWS_ACCESS_KEY_ID="$AWS_ACCESS_KEY_ID" \
     -e AWS_SECRET_ACCESS_KEY="$AWS_SECRET_ACCESS_KEY" \
     -e AWS_SESSION_TOKEN="$AWS_SESSION_TOKEN" \
     -e AWS_REGION="$AWS_REGION" \
     my-agent:arm64
   ```

Create ECR repository and deploy

Deploying to ECR

1. Create an ECR repository:

   ```
   aws ecr create-repository --repository-name my-strands-agent --region us-west-2
   ```

2. Log in to ECR:

   ```
   aws ecr get-login-password --region us-west-2 | docker login --username AWS --
   password-stdin account-id.dkr.ecr.us-west-2.amazonaws.com
   ```

3. Build and push to ECR:

   ```
   docker buildx build --platform linux/arm64 -t account-id.dkr.ecr.us-
   west-2.amazonaws.com/my-strands-agent:latest --push .
   ```

4. Verify the image was pushed:

   ```
   aws ecr describe-images --repository-name my-strands-agent --region us-west-2
   ```

Deploy agent runtime

Create a file named `deploy_agent.py` with the following content:

Example deploy_agent.py

```
import boto3

client = boto3.client('bedrock-agentcore-control', region_name='us-west-2')

response = client.create_agent_runtime(
    agentRuntimeName='strands_agent',
    agentRuntimeArtifact={
        'containerConfiguration': {
            'containerUri': 'account-id.dkr.ecr.us-west-2.amazonaws.com/my-strands-
agent:latest'
        }
    },
    networkConfiguration={"networkMode": "PUBLIC"},
    roleArn='arn:aws:iam::account-id:role/AgentRuntimeRole'
)

print(f"Agent Runtime created successfully!")
print(f"Agent Runtime ARN: {response['agentRuntimeArn']}")
print(f"Status: {response['status']}")
```

Run the script to deploy your agent:

```
uv run deploy_agent.py
```

This script uses the `create_agent_runtime` operation to deploy your agent to Amazon Bedrock AgentCore. Make sure to replace *`account-id`* with your actual AWS account ID and ensure the

IAM role has the necessary permissions. For more information, see IAM Permissions for AgentCore Runtime.

Invoke your agent

Create a file named `invoke_agent.py` with the following content:

Example invoke_agent.py

```
import boto3
import json

agent_core_client = boto3.client('bedrock-agentcore', region_name='us-west-2')
payload = json.dumps({
    "input": {"prompt": "Explain machine learning in simple terms"}
})

response = agent_core_client.invoke_agent_runtime(
    agentRuntimeArn='arn:aws:bedrock-agentcore:us-west-2:account-id:runtime/
myStrandsAgent-suffix',
    runtimeSessionId='dfmeoagmreaklgmrkleafremoigrmtesogmtrskhmtkrlshmt',  # Must be
 33+ chars
    payload=payload,
    qualifier="DEFAULT"
)

response_body = response['response'].read()
response_data = json.loads(response_body)
print("Agent Response:", response_data)
```

Run the script to invoke your agent:

```
uv run invoke_agent.py
```

This script uses the `invoke_agent_runtime` operation to send a request to your deployed agent. Make sure to replace *`account-id`* and *`agentArn`* with your actual values.

If you plan on integrating your agent with OAuth, you can't use the AWS SDK to call `InvokeAgentRuntime`. Instead, make a HTTPS request to InvokeAgentRuntime. For more information, see Authenticate and authorize with Inbound Auth and Outbound Auth.

Expected response format

When you invoke your agent, you'll receive a response like this:

Example Sample response

```
{
  "output": {
    "message": {
      "role": "assistant",
      "content": [
        {
          "text": "# Artificial Intelligence in Simple Terms\n\nArtificial Intelligence (AI) is technology that allows computers to do tasks that normally need human intelligence. Think of it as teaching machines to:\n\n- Learn from information (like how you learn from experience)\n- Make decisions based on what they've learned\n- Recognize patterns (like identifying faces in photos)\n- Understand language (like when I respond to your questions)\n\nInstead of following specific step-by-step instructions for every situation, AI systems can adapt to new information and improve over time.\n\nExamples you might use every day include voice assistants like Siri, recommendation systems on streaming services, and email spam filters that learn which messages are unwanted."
        }
      ]
    },
    "timestamp": "2025-07-13T01:48:06.740668",
    "model": "strands-agent"
  }
}
```

Amazon Bedrock AgentCore requirements summary

- **Platform**: Must be `linux/arm64`
- **Endpoints**: `/invocations` POST and `/ping` GET are mandatory
- **ECR**: Images must be deployed to ECR
- **Port**: Application runs on port 8080
- **Strands Integration**: Uses Strands Agent for AI processing
- **Credentials**: Strands agents require AWS credentials for operation

Conclusion

In this guide, you've learned how to:

- Set up a development environment for building custom agents
- Create a FastAPI application that implements the required endpoints
- Containerize your agent for ARM64 architecture
- Test your agent locally
- Deploy your agent to ECR
- Create an agent runtime in Amazon Bedrock AgentCore
- Invoke your deployed agent

By following these steps, you can create and deploy custom agents that leverage the power of Amazon Bedrock AgentCore while maintaining full control over your agent's implementation.

Use any agent framework

You can use open source AI frameworks to create an agent or tool. This topic shows examples for a variety of frameworks, including Strands Agents, LangGraph, and Google ADK.

Topics

Strands Agents

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/03-integrations/agentic-frameworks/strands-agents.

```
import os
from strands import Agent
from strands_tools import file_read, file_write, editor

agent = Agent(tools=[file_read, file_write, editor])


from bedrock_agentcore.runtime import BedrockAgentCoreApp
app = BedrockAgentCoreApp()
@app.entrypoint
def agent_invocation(payload, context):
    """Handler for agent invocation"""
    user_message = payload.get("prompt", "No prompt found in input, please guide
 customer to create a json payload with prompt key")
    result = agent(user_message)
    print("context:\n-------\n", context)
    print("result:\n*******\n", result)
    return {"result": result.message}
app.run()
```

LangGraph

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/03-integrations/agentic-frameworks/langgraph.

```
from langchain.chat_models import init_chat_model
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition
#------------------------------------------------
from bedrock_agentcore.runtime import BedrockAgentCoreApp
app = BedrockAgentCoreApp()
#------------------------------------------------

llm = init_chat_model(
    "us.anthropic.claude-3-5-haiku-20241022-v1:0",
    model_provider="bedrock_converse",
)

# Create graph
graph_builder = StateGraph(State)
...
# Add nodes and edges
...
```

```
graph = graph_builder.compile()

# Finally write your entrypoint
@app.entrypoint
def agent_invocation(payload, context):

    print("received payload")
    print(payload)

    tmp_msg = {"messages": [{"role": "user", "content": payload.get("prompt", "No
 prompt found in input, please guide customer as to what tools can be used")}]}
    tmp_output = graph.invoke(tmp_msg)
    print(tmp_output)

    return {"result": tmp_output['messages'][-1].content}

app.run()
```

Google Agent Development Kit (ADK)

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/03-integrations/agentic-frameworks/adk.

```
from google.adk.agents import Agent
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools import google_search
from google.genai import types
import asyncio
import os

# adapted form https://google.github.io/adk-docs/tools/built-in-tools/#google-search

APP_NAME="google_search_agent"
USER_ID="user1234"

# Agent Definition
# Add your GEMINI_API_KEY
root_agent = Agent(
    model="gemini-2.0-flash",
    name="openai_agent",
    description="Agent to answer questions using Google Search.",
    instruction="I can answer your questions by searching the internet. Just ask me
 anything!",
    # google_search is a pre-built tool which allows the agent to perform Google
 searches.
    tools=[google_search]
)

# Session and Runner
async def setup_session_and_runner(user_id, session_id):
    session_service = InMemorySessionService()
    session = await session_service.create_session(app_name=APP_NAME, user_id=user_id,
 session_id=session_id)
    runner = Runner(agent=root_agent, app_name=APP_NAME,
 session_service=session_service)
    return session, runner

# Agent Interaction
async def call_agent_async(query, user_id, session_id):
    content = types.Content(role='user', parts=[types.Part(text=query)])
    session, runner = await setup_session_and_runner(user_id, session_id)
    events = runner.run_async(user_id=user_id, session_id=session_id,
 new_message=content)

    async for event in events:
        if event.is_final_response():
            final_response = event.content.parts[0].text
            print("Agent Response: ", final_response)

    return final_response


from bedrock_agentcore.runtime import BedrockAgentCoreApp
app = BedrockAgentCoreApp()

@app.entrypoint
def agent_invocation(payload, context):
    return asyncio.run(call_agent_async(payload.get("prompt", "what is Bedrock
 Agentcore Runtime?"), payload.get("user_id",USER_ID), context.session_id))

app.run()
```

OpenAI Agents SDK

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/03-integrations/agentic-frameworks/openai-agents.

```
from agents import Agent, Runner, WebSearchTool
import logging
import asyncio
import sys

# Set up logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger("openai_agents")

# Configure OpenAI library logging
logging.getLogger("openai").setLevel(logging.DEBUG)

logger.debug("Initializing OpenAI agent with tools")
agent = Agent(
    name="Assistant",
    tools=[
        WebSearchTool(),
    ],
)

async def main(query=None):
    if query is None:
        query = "Which coffee shop should I go to, taking into account my preferences
 and the weather today in SF?"

    logger.debug(f"Running agent with query: {query}")

    try:
        logger.debug("Starting agent execution")
        result = await Runner.run(agent, query)
        logger.debug(f"Agent execution completed with result type: {type(result)}")
        return result
    except Exception as e:
        logger.error(f"Error during agent execution: {e}", exc_info=True)
        raise

# Integration with Bedrock AgentCore
from bedrock_agentcore.runtime import BedrockAgentCoreApp
app = BedrockAgentCoreApp()

@app.entrypoint
async def agent_invocation(payload, context):
    logger.debug(f"Received payload: {payload}")
    query = payload.get("prompt", "How can I help you today?")

    try:
        result = await main(query)
        logger.debug("Agent execution completed successfully")
        return {"result": result.final_output}
    except Exception as e:
        logger.error(f"Error during agent execution: {e}", exc_info=True)
        return {"result": f"Error: {str(e)}"}

# Run the app when imported
if __name__== "__main__":
    app.run()
```

Microsoft AutoGen

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/03-integrations/agentic-frameworks/autogen.

```
from autogen_agentchat.agents import AssistantAgent
from autogen_agentchat.ui import Console
from autogen_ext.models.openai import OpenAIChatCompletionClient
import asyncio
import logging

# Set up logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger("autogen_agent")
```

```
# Initialize the model client
model_client = OpenAIChatCompletionClient(
    model="gpt-4o",
)

# Define a simple function tool that the agent can use
async def get_weather(city: str) -> str:
    """Get the weather for a given city."""
    return f"The weather in {city} is 73 degrees and Sunny."

# Define an AssistantAgent with the model and tool
agent = AssistantAgent(
    name="weather_agent",
    model_client=model_client,
    tools=[get_weather],
    system_message="You are a helpful assistant.",
    reflect_on_tool_use=True,
    model_client_stream=True,  # Enable streaming tokens
)

# Integrate with Bedrock AgentCore
from bedrock_agentcore.runtime import BedrockAgentCoreApp
app = BedrockAgentCoreApp()

@app.entrypoint
async def main(payload):
    # Process the user prompt
    prompt = payload.get("prompt", "Hello! What can you help me with?")

    # Run the agent
    result = await Console(agent.run_stream(task=prompt))

    # Extract the response content for JSON serialization
    if result and hasattr(result, 'messages') and result.messages:
        last_message = result.messages[-1]
        if hasattr(last_message, 'content'):
            return {"result": last_message.content}

    return {"result": "No response generated"}

app.run()
```

CrewAI

For the full example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/blob/main/01-tutorials/01-AgentCore-runtime/01-hosting-agent/04-crewai-with-bedrock-model/runtime-with-crewai-and-bedrock-models.ipynb.

```
from crewai import Agent, Crew, Process, Task
from crewai_tools import MathTool, WeatherTool
from bedrock_agentcore.runtime import BedrockAgentCoreApp
import argparse
import json
app = BedrockAgentCoreApp()

# Define CrewAI agent
def create_researcher():
    """Create a researcher agent"""
    from langchain_aws import ChatBedrock

    # Initialize LLM
    llm = ChatBedrock(
        model_id="anthropic.claude-3-sonnet-20240229-v1:0",
        model_kwargs={"temperature": 0.1}
    )

    # Create researcher agent
    return Agent(
        role="Senior Research Specialist",
        goal="Find comprehensive and accurate information about the topic",
        backstory="You are an experienced research specialist with a talent for finding
 relevant information.",
        verbose=True,
        llm=llm,
        tools=[MathTool(), WeatherTool()]
    )

# Define the analyst agent
def create_analyst():
....

# Create the crew
def create_crew():
    """Create and configure the CrewAI crew"""
    # Create agents
    researcher = create_researcher()
    analyst = create_analyst()

    # Create research task with fields like description filled in as per crewAI docs
    research_task = Task(
        description="...",
        agent=researcher,
        expected_output="..."
    )

    analysis_task = Task(
    ...
    )

    # Create crew
    return Crew(
        agents=[researcher, analyst],
        tasks=[research_task, analysis_task],
        process=Process.sequential,
        verbose=True
    )

# Initialize the crew
crew = create_crew()

# Finally write your entrypoint
@app.entrypoint
def crewai_bedrock(payload):
    """
    Invoke the crew with a payload
    """
    user_input = payload.get("prompt")

    # Run the crew
    result = crew.kickoff(inputs={"topic": user_input})

    # Return the result
    return result.raw

if __name__ == "__main__":
    app.run()
```

Use any foundation model

You can use any foundation model with AgentCore Runtime The following are examples for Amazon Bedrock, Open AI, and Gemini:

Topics

- Amazon Bedrock
- Open AI
- Gemini

Amazon Bedrock

```
import boto3
from strands.models import BedrockModel

# Create a Bedrock model with the custom session
bedrock_model = BedrockModel(
    model_id="us.anthropic.claude-3-7-sonnet-20250219-v1:0",
    boto_session=session
)
```

Open AI

```
from strands.models.openai import OpenAIModel
model = OpenAIModel(
    client_args={
        "api_key": "<our_OPENAI_API_KEY>",
    # **model_config
    model_id="gpt-4o",
    params={
        "max_tokens": 1000,
        "temperature": 0.7,
    }
)


from strands import Agent
from strands_tools import python_repl
agent = Agent(model=model,tools=[python_repl])
```

Gemini

```
import os
from langchain.chat_models import init_chat_model

# Use your Google API key to initialize the chat model
os.environ["GOOGLE_API_KEY"] = "..."

llm = init_chat_model("google_genai:gemini-2.0-flash")
```

Deploy MCP servers in AgentCore Runtime

Amazon Bedrock AgentCore Runtime lets you deploy and run Model Context Protocol (MCP) servers in the AgentCore Runtime. This guide walks you through creating, testing, and deploying your first MCP server.

For an example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/01-tutorials/01-AgentCore-runtime/02-hosting-MCP-server.

In this section, you learn:

- How to create an MCP server with tools
- How to test your server locally
- How to deploy your server to AWS
- How to invoke your deployed server

Topics

Prerequisites

- Python 3.10 or higher installed and basic understanding of Python
- An AWS account with appropriate permissions and local credentials configured

Create your MCP server

Install required packages

First, install the MCP package:

```
pip install mcp
```

Create your first MCP server

Create a new file called `my_mcp_server.py`:

```
# my_mcp_server.py

from mcp.server.fastmcp import FastMCP
from starlette.responses import JSONResponse

mcp = FastMCP(host="0.0.0.0", stateless_http=True)

@mcp.tool()
def add_numbers(a: int, b: int) -> int:
    """Add two numbers together"""
    return a + b

@mcp.tool()
def multiply_numbers(a: int, b: int) -> int:
    """Multiply two numbers together"""
    return a * b

@mcp.tool()
def greet_user(name: str) -> str:
    """Greet a user by name"""
    return f"Hello, {name}! Nice to meet you."

if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

Understanding the code

- **FastMCP**: Creates an MCP server that can host your tools
- **@mcp.tool()**: Decorator that turns your Python functions into MCP tools
- **Tools**: Three simple tools that demonstrate different types of operations

Test your MCP server locally

Start your MCP server

Run your MCP server locally:

```
python my_mcp_server.py
```

You should see output indicating the server is running on port `8000`.

Test with MCP client

From a new terminal, create a new file `my_mcp_client.py` and execute it using **`python my_mcp_client.py`**

```
# my_mcp_client.py

import asyncio

from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main():
    mcp_url = "http://localhost:8000/mcp"
    headers = {}

    async with streamablehttp_client(mcp_url, headers, timeout=120,
 terminate_on_close=False) as (
        read_stream,
        write_stream,
        _,
    ):
        async with ClientSession(read_stream, write_stream) as session:
            await session.initialize()
            tool_result = await session.list_tools()
            print(tool_result)

asyncio.run(main())
```

You can also test your server using the MCP Inspector as described in the section called “Local testing with MCP inspector”.

Deploy your MCP server to AWS

Install deployment tools

Install the Amazon Bedrock AgentCore CLI:

```
pip install bedrock-agentcore-starter-toolkit
```

Start by creating a project folder with the following structure:

```
## Project Folder Structure

your_project_directory/
### mcp_server.py # Your main agent code
### requirements.txt # Dependencies for your agent
### __init__.py # Makes the directory a Python package
```

Create a new file called `requirements.txt`, add the following to it:

```
mcp
```

Configure your MCP server for deployment

Before configuring your deployment, you need to set up a Cognito user pool for authentication as described in the section called “Set up Cognito user pool for authentication”. This provides the OAuth tokens required for secure access to your deployed server.

After setting up authentication, create the deployment configuration:

```
agentcore configure -e my_mcp_server.py --protocol MCP
```

This will start a guided prompt workflow:

- For execution role, you need to have an IAM execution role with appropriate permissions
- For ECR, just press **`enter`** to skip and it will auto-create
- For dependency file, the CLI will auto-detect from current directory
- For OAuth, type **`yes`** and provide the discovery URL and client ID token

Deploy to AWS

Deploy your agent:

```
agentcore launch
```

This command will:

1. Build a Docker container with your agent
2. Push it to Amazon ECR
3. Create a Amazon Bedrock AgentCore runtime
4. Deploy your agent to AWS

After deployment, you'll receive an agent runtime ARN that looks like:

```
arn:aws:bedrock-agentcore:us-west-2:accountId:runtime/my_mcp_server-xyz123
```

Invoke your deployed MCP server

Test with MCP client (remote)

Before testing, set the following environment variables:

- Export agent ARN as an environment variable: **export AGENT_ARN="*agent_arn*"**
- Export bearer token as an environment variable: **export BEARER_TOKEN="*bearer_token*"**

if you pass in an `Accept` header, it must follow the MCP standard. Acceptable media types are `application/json` and `text/event-stream`.

Create a new file `my_mcp_client_remote.py` and execute it using **`python my_mcp_client_remote.py`**

```
import asyncio
import os
import sys

from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main():
    agent_arn = os.getenv('AGENT_ARN')
    bearer_token = os.getenv('BEARER_TOKEN')
    if not agent_arn or not bearer_token:
        print("Error: AGENT_ARN or BEARER_TOKEN environment variable is not set")
        sys.exit(1)

    encoded_arn = agent_arn.replace(':', '%3A').replace('/', '%2F')
    mcp_url = f"https://bedrock-agentcore.us-west-2.amazonaws.com/runtimes/
{encoded_arn}/invocations?qualifier=DEFAULT"
    headers = {"authorization": f"Bearer {bearer_token}","Content-Type":"application/
json"}
    print(f"Invoking: {mcp_url}, \nwith headers: {headers}\n")

    async with streamablehttp_client(mcp_url, headers, timeout=120,
 terminate_on_close=False) as (
        read_stream,
        write_stream,
        _,
    ):
        async with ClientSession(read_stream, write_stream) as session:
            await session.initialize()
            tool_result = await session.list_tools()
            print(tool_result)

asyncio.run(main())
```

You can also test your deployed server using the MCP Inspector as described in the section called “Remote testing with MCP inspector”.

How Amazon Bedrock AgentCore supports MCP

When you configure a Amazon Bedrock AgentCore Runtime with the MCP protocol, the service expects MCP server containers to be available at the path `0.0.0.0:8000/mcp`, which is the default path supported by most official MCP server SDKs.

Amazon Bedrock AgentCore requires stateless streamable-HTTP servers because the Runtime provides session isolation by default. The platform automatically adds a `Mcp-Session-Id` header for any request without it, so MCP clients can maintain connection continuity to the same Amazon Bedrock AgentCore Runtime session.

The payload of the `InvokeAgentRuntime` API is passed through directly, allowing RPC messages of protocols like MCP to be easily proxied.

Next steps

To learn more about creating custom servers and Docker containers for Amazon Bedrock AgentCore, explore the documentation on deploying agents using custom servers and Docker.

Appendix

Set up Cognito user pool for authentication

Create a new file `setup_cognito.sh` and add the following content to it.

Change *`TEMP_PASSWORD`* and *`PERMANENT_PASSWORD`* to secure passwords of your choosing.

Run the script using the command **`source setup_cognito.sh`**.

```
#!/bin/bash

# Create User Pool and capture Pool ID directly
export POOL_ID=$(aws cognito-idp create-user-pool \
  --pool-name "MyUserPool" \
  --policies '{"PasswordPolicy":{"MinimumLength":8}}' \
  --region us-east-1 | jq -r '.UserPool.Id')

# Create App Client and capture Client ID directly
export CLIENT_ID=$(aws cognito-idp create-user-pool-client \
  --user-pool-id $POOL_ID \
  --client-name "MyClient" \
  --no-generate-secret \
  --explicit-auth-flows "ALLOW_USER_PASSWORD_AUTH" "ALLOW_REFRESH_TOKEN_AUTH" \
  --region us-east-1 | jq -r '.UserPoolClient.ClientId')

# Create User
aws cognito-idp admin-create-user \
  --user-pool-id $POOL_ID \
  --username "testuser" \
  --temporary-password "TEMP_PASSWORD" \
  --region us-east-1 \
  --message-action SUPPRESS > /dev/null

# Set Permanent Password
aws cognito-idp admin-set-user-password \
  --user-pool-id $POOL_ID \
  --username "testuser" \
  --password "PERMANENT_PASSWORD" \
  --region us-east-1 \
  --permanent > /dev/null

# Authenticate User and capture Access Token
export BEARER_TOKEN=$(aws cognito-idp initiate-auth \
  --client-id "$CLIENT_ID" \
  --auth-flow USER_PASSWORD_AUTH \
  --auth-parameters USERNAME='testuser',PASSWORD='PERMANENT_PASSWORD' \
  --region us-east-1 | jq -r '.AuthenticationResult.AccessToken')

# Output the required values
echo "Pool id: $POOL_ID"
echo "Discovery URL: https://cognito-idp.us-east-1.amazonaws.com/$POOL_ID/.well-known/
openid-configuration"
echo "Client ID: $CLIENT_ID"
echo "Bearer Token: $BEARER_TOKEN"
```

After running this script, note the following values for use in the deployment configuration:

- Discovery URL: Used during the `agentcore configure` step
- Client ID: Used during the `agentcore configure` step
- Bearer Token: Used when invoking your deployed server

Local testing with MCP inspector

The MCP Inspector is a visual tool for testing MCP servers. To use it, you need:

- Node.js and npm installed

Install and run the MCP Inspector:

```
npx @modelcontextprotocol/inspector
```

This will:

- Start the MCP Inspector server
- Display a URL in your terminal (typically `http://localhost:6274`)

To use the Inspector:

1. Navigate to `http://localhost:6274` in your browser
2. Paste the MCP server URL (`http://localhost:8000/mcp`) into the MCP Inspector connection field
3. You'll see your tools listed in the sidebar
4. Click on any tool to test it
5. Fill in the parameters (e.g., for `add_numbers`, enter values for a and b)
6. Click "Call Tool" to see the result

Remote testing with MCP inspector

You can also test your deployed server using the MCP Inspector:

1. Open the MCP Inspector: **`npx @modelcontextprotocol/inspector`**
2. In the web interface:
    - Select "Streamable HTTP" as the transport
    - Enter your agent's endpoint URL, which will look like: `https://bedrock-agentcore.us-west-2.amazonaws.com/runtimes/arn%3Aaws%3Abedrock-agentcore%3Aus-west-2%3A`*`accountId`*`%3Aruntime%2F`*`runtimeName`*`/invocations?qualifier=DEFAULT`
    - Make sure to URL-encode your agent runtime ARN when constructing the endpoint URL. The colon (:) characters become %3A and forward slashes (/) become %2F in the encoded URL.
    - Add your Bearer token under authentication
    - Click "Connect"
3. Test your tools just like you did locally

Use isolated sessions for agents

AgentCore Runtime lets you isolate each user session and safely reuse context across multiple invocations in a user session. Session isolation is critical for AI agent workloads due to their unique operational characteristics:

- **Complete execution environment separation**: Each user session in Runtime receives its own dedicated microVM with isolated Compute, memory, and filesystem resources. This prevents one user's agent from accessing another user's data. After session completion, the entire microVM is terminated and memory is sanitized to remove all session data, eliminating cross-session contamination risks.
- **Stateful reasoning processes**: Unlike stateless functions, AI agents maintain complex contextual state throughout their execution cycle, beyond simple message history for multi-turn conversations. Runtime preserves this state securely within a session while ensuring complete isolation between different users, enabling personalized agent experiences without compromising data boundaries.
- **Privileged tool operations**: AI agents perform privileged operations on users' behalf through integrated tools accessing various resources. Runtime's isolation model ensures these tool operations maintain proper security contexts and prevents credential sharing or permission escalation between different user sessions.
- **Deterministic security for non-deterministic processes**: AI agent behavior can be non-deterministic due to the probabilistic nature of foundation models. Runtime provides consistent, deterministic isolation boundaries regardless of agent execution patterns, delivering the predictable security properties required for enterprise deployments.

Understanding ephemeral context

While Amazon Bedrock AgentCore provides strong session isolation, these sessions are ephemeral in nature. Any data stored in memory or written to disk persists only for the session duration. This includes conversation history, user preferences, intermediate calculation results, and any other state information your agent maintains.

For data that needs to be retained beyond the session lifetime (such as user conversation history, learned preferences, or important insights), you should use Amazon Bedrock AgentCore Memory. This service provides purpose-built persistent storage designed specifically for agent workloads, with both short-term and long-term memory capabilities.

Extended conversations and multi-step workflows

Unlike traditional serverless functions that terminate after each request, Amazon Bedrock AgentCore supports ephemeral, isolated compute sessions lasting up to 8 hours. This simplifies building multi-step agentic workflows as you can make multiple calls to the same environment, with each invocation building upon the context established by previous interactions.

Runtime session lifecycle

Session creation

A new session is created on the first invoke with a unique runtimeSessionId provided by your application. Amazon Bedrock AgentCore Runtime provisions a dedicated execution environment (microVM) for each session. Context is preserved between invocations to the same session.

Session states

Sessions can be in one of the following states:

- **Active**: Either processing a sync request or doing background tasks. Sync invocation activity is automatically tracked based on invocations to a runtime session. Background tasks are communicated by the agent code by responding with "HealthyBusy" status in pings.
- **Idle**: When not processing any requests or background tasks. The session has completed processing but remains available for future invocations.
- **Terminated**: Execution environment provisioned for the session is terminated. This can be due to inactivity (of 15 minutes), reaching max duration (8 hours) or if it's deemed unhealthy based on health checks. Subsequent invokes to a terminated runtimeSessionId will provision a new execution environment.

How to use sessions

To use sessions effectively:

- Generate a unique session ID for each user or conversation with at least 33 characters
- Pass the same session ID for all related invocations
- Use different session IDs for different users or conversations

Example Using sessions for a conversation

```
# First message in a conversation

response1 = agent_core_client.InvokeAgentRuntime(
   agentRuntimeArn=agent_arn,
   runtimeSessionId="user-123456-conversation-12345678", # or uuid.uuid4()
   payload=json.dumps({"prompt": "Tell me about AWS"}).encode()
)

# Follow-up message in the same conversation reuses the runtimeSessionId.

response2 = agent_core_client.InvokeAgentRuntime(
   agentRuntimeArn=agent_arn,
   runtimeSessionId="user-123456-conversation-12345678", # or uuid.uuid4()
   payload=json.dumps({"prompt": "How does it compare to other cloud
 providers"}).encode()
)
```

By using the same runtimeSessionId for related invocations, you ensure that context is maintained across the conversation, allowing your agent to provide coherent responses that build on previous interactions.

Handle asynchronous and long running agents with Amazon Bedrock AgentCore Runtime

Amazon Bedrock AgentCore Runtime can handle asynchronous processing and long running agents. Asynchronous tasks allow your agent to continue processing after responding to the client and handle long-running operations without blocking responses. With async processing, your agent can:

- Start a task that might take minutes or hours
- Immediately respond to the user saying "I've started working on this"
- Continue processing in the background
- Allow the user to check back later for results

Key concepts

Asynchronous processing model

The Amazon Bedrock AgentCore SDK supports both synchronous and asynchronous processing through a unified API. This creates a flexible implementation pattern for both clients and agent developers. Agent clients can work with the same API without differentiating between synchronous and asynchronous on the client side. With the ability to invoke the same session across invocations, agent developers can reuse context and build upon this context incrementally without implementing complex task management logic.

Runtime session lifecycle management

Agent code communicates its processing status using the "/ping" health status. "HealthyBusy" indicates the agent is busy processing background tasks, while "Healthy" indicates it is idle (waiting for requests). A session in idle state for 15 minutes gets automatically terminated.

Implementing asynchronous tasks

To get started, install the `bedrock-agentcore` package:

```
pip install bedrock-agentcore
```

API based task management

To build interactive agents that perform asynchronous tasks, you need to call `add_async_task` when starting a task and `complete_async_task` when the task completes. The SDK handles task tracking and manages Ping status automatically.

```
# Start tracking a task manually
task_id = app.add_async_task("data_processing")

# Do work...

# Mark task as complete
app.complete_async_task(task_id)
```

Asynchronous task decorator

The Amazon Bedrock AgentCore SDK helps with tracking asynchronous tasks. You can get started by simply annotating your asynchronous functions with `@app.async_task`.

```
# Automatically track asynchronous functions:
@app.async_task
async def background_work():
    await asyncio.sleep(10)  # Status becomes "HealthyBusy"
    return "done"

@app.entrypoint
async def handler(event):
    asyncio.create_task(background_work())
    return {"status": "started"}
```

Here is how it works:

- The `@app.async_task` decorator tracks function execution
- When the function runs, ping status changes to "HealthyBusy"
- When the function completes, status returns to "Healthy"

Custom ping handler

You can implement your own custom ping handler to manage the Runtime Session's state. Your agent's health is reported through the /ping endpoint:

```
@app.ping
def custom_status():
    if system_busy():
        return PingStatus.HEALTHY_BUSY
    return PingStatus.HEALTHY
```

Status values:

- "Healthy": Ready for new work
- "HealthyBusy": Processing background task

Complete example

First, install the required package:

```
pip install strands-agents
```

Then, create a Python file with the following code:

```
import threading
import time
from strands import Agent, tool
from bedrock_agentcore.runtime import BedrockAgentCoreApp

# Initialize app with debug mode for task management
app = BedrockAgentCoreApp()

@tool
def start_background_task(duration: int = 5) -> str:
    """Start a simple background task that runs for specified duration."""
    # Start tracking the async task
    task_id = app.add_async_task("background_processing", {"duration": duration})

    # Run task in background thread
    def background_work():
        time.sleep(duration)  # Simulate work
        app.complete_async_task(task_id)  # Mark as complete

    threading.Thread(target=background_work, daemon=True).start()
    return f"Started background task (ID: {task_id}) for {duration} seconds. Agent
 status is now BUSY."

# Create agent with the tool
agent = Agent(tools=[start_background_task])

@app.entrypoint
def main(payload):
    """Main entrypoint - handles user messages."""
    user_message = payload.get("prompt", "Try: start_background_task(3)")
    return {"message": agent(user_message).message}

if __name__ == "__main__":
    print("# Simple Async Strands Example")
    print("Test: curl -X POST http://localhost:8080/invocations -H 'Content-Type:
 application/json' -d '{\"prompt\": \"start a 3 second task\"}'")
    app.run()
```

This example demonstrates:

- Creating a background task that runs asynchronously
- Tracking the task's status with `add_async_task` and `complete_async_task`
- Responding immediately to the user while processing continues
- Managing the agent's health status automatically

Stream agent responses

The following Strands Agents example shows how an AgentCore Runtime agent can stream a response back to a client.

```
from strands import Agent
from bedrock_agentcore import BedrockAgentCoreApp

app = BedrockAgentCoreApp()
agent = Agent()

@app.entrypoint
async def agent_invocation(payload):
    """Handler for agent invocation"""
    user_message = payload.get(
        "prompt", "No prompt found in input, please guide customer to create a json
 payload with prompt key"
    )
    stream = agent.stream_async(user_message)
    async for event in stream:
        print(event)
        yield (event)

if __name__ == "__main__":
    app.run()
```

Authenticate and authorize with Inbound Auth and Outbound Auth

This section shows you how to implement authentication and authorization for your agent runtime using OAuth and JWT bearer tokens with AgentCore Identity. You'll learn how to set up Cognito user pools, configure your agent runtime for JWT authentication (Inbound Auth), and implement OAuth-based access to third-party resources (outbound Auth).

For a complete example, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/.

For information about using OAuth to with an MCP server, see Deploy MCP servers in AgentCore Runtime.

Amazon Bedrock AgentCore runtime provides two authentication mechanisms for hosted agents:

IAM SigV4 Authentication

The default authentication and authorization mechanism that works automatically without additional configuration, similar to other AWS APIs.

If your solution requires the hosted agent to retrieve OAuth tokens on behalf of end users (using Authorization Code Grant), you can specify the user identifier by including the `X-Amzn-Bedrock-AgentCore-Runtime-User-Id` header in your requests.

JWT Bearer Token Authentication

You can configure your agent runtime to accept JWT bearer tokens by providing authorizer configuration during agent creation.

This configuration requires:

- Discovery URL - A string that must match the pattern `^.+/\.well-known/openid-configuration$` for OpenID Connect discovery URLs
- Allowed audiences - A list of permitted audiences that will be validated against the aud claim in the JWT token
- Allowed clients - A list of permitted client identifiers that will be validated against the client_id claim in the JWT token

Note

A runtime can only support either IAM SigV4 or JWT Bearer Token based inbound auth. You can always create custom endpoints for your runtime and configure them for different inbound authorization types.

When you create a runtime with Amazon Bedrock AgentCore, a Workload Identity is created automatically for your runtime with AgentCore Identity service.

Topics

JWT inbound authorization and OAuth outbound access sample

This guide walks you through the process of setting up your agent runtime to be invoked with an OAuth compliant access token using JWT format. The sample agent will be authorized using AWS Cognito access tokens. Later, you'll also learn how the agent code can fetch Google tokens on behalf of the user to check Google Drive and fetch contents.

What you'll learn

In this guide, you'll learn how to:

- Set up Cognito user pool, add a user, and get a bearer token for the user
- Set up your agent runtime to use the Cognito user pool for authorization
- Set up your agent code to fetch OAuth tokens on behalf of the user to call tools

Prerequisites

Before you begin, make sure you have:

- An AWS account with appropriate permissions
- Basic understanding of Python programming
- Familiarity with Docker containers (for advanced deployment)
- Set up a basic agent with runtime successfully
- Basic understanding of OAuth authorization, mainly JWT bearer tokens, claims, and the various grant flows

Step 1: Prepare your agent

Start by creating a basic agent with the following structure:

```
## Project Folder Structure

your_project_directory/
### agent_example.py # Your main agent code
### requirements.txt # Dependencies for your agent
### __init__.py # Makes the directory a Python package
```

Create the following files with their respective contents:

agent_example.py

This is your main agent code:

```
from strands import Agent
from bedrock_agentcore.runtime import BedrockAgentCoreApp

agent = Agent()
app = BedrockAgentCoreApp()

@app.entrypoint
def invoke(payload):
    """Process user input and return a response"""
    user_message = payload.get("prompt", "Hello")
    response = agent(user_message)
    return str(response) # response should be json serializable
```

```
if __name__ == "__main__":
    app.run()
```

requirements.txt

This file lists the dependencies for your agent:

```
strands-agents
bedrock-agentcore
```

Step 2: Set up AWS Cognito user pool and add a user

To set up a Cognito user pool and create a user, you'll use a shell script that automates the process.

For more information, see Step 2: Set up an OAuth 2.0 Credential Provider.

To set up Cognito user pool and create a user

1. Create a file named `setup_cognito.sh` with the following content:

   Change *`TEMP_PASSWORD`* and *`PERMANENT_PASSWORD`* to secure passwords of your choosing.

   ```
   #!/bin/bash

   # Create User Pool and capture Pool ID directly
   export POOL_ID=$(aws cognito-idp create-user-pool \
     --pool-name "MyUserPool" \
     --policies '{"PasswordPolicy":{"MinimumLength":8}}' \
     --region us-east-1 | jq -r '.UserPool.Id')

   # Create App Client and capture Client ID directly
   export CLIENT_ID=$(aws cognito-idp create-user-pool-client \
     --user-pool-id $POOL_ID \
     --client-name "MyClient" \
     --no-generate-secret \
     --explicit-auth-flows "ALLOW_USER_PASSWORD_AUTH" "ALLOW_REFRESH_TOKEN_AUTH" \
     --region us-east-1 | jq -r '.UserPoolClient.ClientId')

   # Create User
   aws cognito-idp admin-create-user \
     --user-pool-id $POOL_ID \
     --username "testuser" \
   ```

```
  --temporary-password "TEMP_PASSWORD" \
  --region us-east-1 \
  --message-action SUPPRESS > /dev/null

# Set Permanent Password
aws cognito-idp admin-set-user-password \
  --user-pool-id $POOL_ID \
  --username "testuser" \
  --password "PERMANENT_PASSWORD" \
  --region us-east-1 \
  --permanent > /dev/null

# Authenticate User and capture Access Token
export BEARER_TOKEN=$(aws cognito-idp initiate-auth \
  --client-id "$CLIENT_ID" \
  --auth-flow USER_PASSWORD_AUTH \
  --auth-parameters USERNAME='testuser',PASSWORD='PERMANENT_PASSWORD' \
  --region us-east-1 | jq -r '.AuthenticationResult.AccessToken')

# Output the required values
echo "Pool id: $POOL_ID"
echo "Discovery URL: https://cognito-idp.us-east-1.amazonaws.com/$POOL_ID/.well-
known/openid-configuration"
echo "Client ID: $CLIENT_ID"
echo "Bearer Token: $BEARER_TOKEN"
```

2. Run the script to create the Cognito resources:

```
source setup_cognito.sh
```

3. Note the output values, which will look similar to:

```
Pool id: us-east-1_poolid
Discovery URL: https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_userpoolid/.well-known/openid-configuration
Client ID: clientid
Bearer Token: bearertoken
```

   You'll need these values in the next steps.

This script creates a Cognito user pool, a user pool client, adds a user, and generates a bearer token for the user. The token is valid for 60 minutes by default.

Step 3: Deploy your agent

Now you'll deploy your agent with JWT authorization using the Cognito user pool you created. You will need to create an agent with authorizer configuration. The following table represents the various authorizer configuration parameters and how we use them to validate the incoming token.

authorizer_configuration	claim in decoded token	Notes
discovery url → issuer	iss	The discovery url should point to an issuer url. This should match the iss claim in the decoded token.
allowedClients	client_id	client_id in the token should match one of the allowed clients specified in the authorizer
allowedAudience	aud	One of the values in aud claim from the token should match one of the allowed audience specified in the authorizer

If both client_id and aud is provided, the agent runtime authorizer will verify both.

Starter toolkit

To configure and deploy your agent

1. Configure your agent runtime with the following command, replacing the placeholder values with your actual values:

   ```
   agentcore configure --entrypoint agent_example.py \
   --name hello_agent \
   --execution-role your-execution-role-arn \
   --disable-otel \
   --requirements-file requirements.txt \
   --authorizer-config "{\"customJWTAuthorizer\":{\"discoveryUrl\":\"$DISCOVERY_URL
   \",\"allowedClients\":[\"$CLIENT_ID\"]}}"
   ```

   Replace `$DISCOVERY_URL` with the Discovery URL from Step 2, and `$CLIENT_ID` with the Client ID from Step 2.

2. Deploy your agent:

   ```
   agentcore launch
   ```

3. Note the agent runtime ARN from the output. You'll need this in the next step.

> **Tip**
>
> You can also run the configure command with just the entry point file for a fully interactive experience:
>
> ```
> agentcore configure --entrypoint agent_example.py
> ```

Python

```
import boto3

# Create the client
client = boto3.client('bedrock-agentcore-control', region_name="us-east-1")


# Call the CreateAgentRuntime operation
response = client.create_agent_runtime(
    agentRuntimeName='hello_agent',
    agentRuntimeArtifact={
        'containerConfiguration': {
            'containerUri': '111122223333.dkr.ecr.us-east-1.amazonaws.com/my-
agent:latest'
        }
    },
    authorizerConfiguration={
        "customJWTAuthorizer": {
            "discoveryUrl": 'COGNITO_DISCOVERY_URL',
            "allowedClients": ['COGNITO_CLIENT_ID']
        }
    },
    networkConfiguration={"networkMode":"PUBLIC"},
    roleArn='arn:aws:iam::111122223333:role/AgentRuntimeRole'
)
```

Step 4: Use bearer token to invoke your agent

Now that your agent is deployed with JWT authorization, you can invoke it using the bearer token.

Make sure your agent's execution role has permissions to access the workload identity.

```
    "Sid": "GetAgentAccessToken",
    "Effect": "Allow",
    "Action": [
        "bedrock-agentcore:GetWorkloadAccessToken",
        "bedrock-agentcore:GetWorkloadAccessTokenForJWT",
        "bedrock-agentcore:GetWorkloadAccessTokenForUserId"
    ],
    # point to the workload identity for the runtime; the workload identity can be
 found in
    # the GetAgentRuntime response and has your agent name in it.
    "Resource": [
        "arn:aws:bedrock-agentcore:region:account-id:workload-identity-directory/
default",
        "arn:aws:bedrock-agentcore:region:account-id:workload-identity-directory/
default/workload-identity/agentname-*"
    ]
}
```

Fetch a bearer token for the user you created with Amazon Cognito.

```
# use the password and other details used when you created the cognito user
            export TOKEN=$(aws cognito-idp initiate-auth \
            --client-id "$CLIENT_ID" \
            --auth-flow USER_PASSWORD_AUTH \
            --auth-parameters USERNAME='testuser',PASSWORD='PASSWORD' \
            --region us-east-1 | jq -r '.AuthenticationResult.AccessToken')
```

Invoke the agent with OAuth.

Use cURL

```
// Invoke with OAuth token
export PAYLOAD='{"prompt": "hello what is 1+1?"}'
export BEDROCK_AGENT_CORE_ENDPOINT_URL="https://bedrock-agentcore.us-
east-1.amazonaws.com"
```

```
curl -v -X POST "${BEDROCK_AGENT_CORE_ENDPOINT_URL}/runtimes/${ESCAPED_AGENT_ARN}/
invocations?qualifier=DEFAULT" \
-H "Authorization: Bearer ${TOKEN}" \
-H "X-Amzn-Trace-Id: your-trace-id" \
-H "Content-Type: application/json" \
-H "X-Amzn-Bedrock-AgentCore-Runtime-Session-Id: your-session-id" \
-d ${PAYLOAD}
```

Use Python

Since boto3 doesn't support invocation with bearer tokens, you'll need to use an HTTP client like the requests library in Python.

To invoke your agent with a bearer token

1. Create a Python script named `invoke_agent.py` with the following content:

```
import requests
import urllib.parse
import json
import os

# Configuration Constants
REGION_NAME = "AWS_REGION"

# === Agent Invocation Demo ===
invoke_agent_arn = "YOUR_AGENT_ARN_HERE"
auth_token = os.environ.get('TOKEN')
print(f"Using Agent ARN from environment: {invoke_agent_arn}")

# URL encode the agent ARN
escaped_agent_arn = urllib.parse.quote(invoke_agent_arn, safe='')

# Construct the URL
url = f"https://bedrock-agentcore.{REGION_NAME}.amazonaws.com/runtimes/
{escaped_agent_arn}/invocations?qualifier=DEFAULT"

# Set up headers
headers = {
    "Authorization": f"Bearer {auth_token}",
    "X-Amzn-Trace-Id": "your-trace-id",
    "Content-Type": "application/json",
    "X-Amzn-Bedrock-AgentCore-Runtime-Session-Id": "testsession123"
}

# Enable verbose logging for requests
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger("urllib3.connectionpool").setLevel(logging.DEBUG)

invoke_response = requests.post(
    url,
    headers=headers,
    data=json.dumps({"prompt": "Hello what is 1+1?"})
)

# Print response in a safe manner
print(f"Status Code: {invoke_response.status_code}")
print(f"Response Headers: {dict(invoke_response.headers)}")

# Handle response based on status code
if invoke_response.status_code == 200:
    response_data = invoke_response.json()
    print("Response JSON:")
    print(json.dumps(response_data, indent=2))
elif invoke_response.status_code >= 400:
    print(f"Error Response ({invoke_response.status_code}):")
    error_data = invoke_response.json()
    print(json.dumps(error_data, indent=2))

else:
    print(f"Unexpected status code: {invoke_response.status_code}")
    print("Response text:")
    print(invoke_response.text[:500])
```

2. Replace *AWS_REGION* with the AWS Region that you are using. from Step 3.
3. Replace *YOUR_AGENT_ARN_HERE* with your actual agent runtime ARN from Step 3.
4. Run the script:

```
python invoke_agent.py
```

Use starter toolkit

REplace *ADD_TOKEN_HERE* with your bearer token.

```
agentcore invoke '{"prompt": "Hello what is 1+1?"}' --bearer-token ADD_TOKEN_HERE
```

Step 5: Set up your agent to access tools using OAuth

In this section, you'll learn how to connect your agent code with AgentCore Credential Providers for secure access to external resources using OAuth2 authentication.

The example below demonstrates how your agent running in Agent Runtime can request OAuth consent from users, enabling them to authenticate with their Google account and authorize the agent to access their Google Drive contents.

For more information about setting up identity, see Getting started with Amazon Bedrock AgentCore Identity.

Step 5.1: Set up Credential Providers

To set up a Google Credential Provider, you need to:

1. Register your application with Google to obtain client ID and client secret
2. Create an OAuth credential provider using the AWS CLI:

```
aws agent-credential-provider create-oauth2-credential-provider \
  --provider-type "google" \
  --name "google-provider" \
  --scopes '["https://www.googleapis.com/auth/drive.metadata.readonly"]' \
  --google-config '{\
    "clientId": "your-client-id",\
    "clientSecret": "your-client-secret"\
  }'
```

Make sure your invocation role has the necessary permissions for accessing the credential provider.

Step 5.2: Enable agent to read Google Drive contents

Create a tool with agent core SDK annotations as shown below to automatically initiate the three-legged OAuth process. When your agent invokes this tool, users will be prompted to open the authorization URL in their browser and grant consent for the agent to access their Google Drive.

```
import asyncio
```

```
from bedrock_agentcore.identity.auth import requires_access_token, requires_api_key

# This annotation helps agent developer to obtain access tokens from external
 applications
@requires_access_token(
    provider_name="google-provider",
    scopes=["https://www.googleapis.com/auth/drive.metadata.readonly"], # Google OAuth2
 scopes
    auth_flow="USER_FEDERATION", # 3LO flow
    on_auth_url=lambda x: print("Copy and paste this authorization url to your
 browser", x), # prints authorization URL to console
    force_authentication=True,
)
async def read_from_google_drive(*, access_token: str):
    print(access_token) #You can see the access_token
    # Make API calls...
    main(access_token)

asyncio.run(read_from_google_drive(access_token=""))
```

What happens behind the scenes

When this code runs, the following process occurs:

1. Agent Runtime authorizes the inbound token according to the configured authorizer.
2. Agent Runtime exchanges this token for a Workload Access Token via `bedrock-agentcore:GetWorkloadAccessTokenForJWT` API and delivers it to your agent code via the payload header `WorkloadAccessToken`.
3. During tool invocation, your agent uses this Workload Access Token to call Token Vault API `bedrock-agentcore:GetResourceOauth2Token` and generate a 3LO authentication URL.
4. Your agent sends this URL to the client application as specified in the `on_auth_url` method.
5. The client application presents this URL to the user, who grants consent for the agent to access their Google Drive.
6. AgentCore Identity service securely receives and caches the Google access token until it expires, enabling subsequent requests from the user to use this token without needing the user to provide consent for every request.

Note

AgentCore Identity Service stores the Google access token in the AgentCore Token Vault using the agent workload identity and user ID (from the inbound JWT token, such as AWS Cognito token) as the binding key, eliminating repeated consent requests until the Google token expires.

Step 6: (Optional) Propagate a JWT token to AgentCore Runtime

Optionally, you can pass an Authorization header to an AgentCore Runtime to extract claims. This can be done by using the request header allowlist configuration. For more information, see RequestHeaderConfiguration.

Step 6.1: Modify your agent code to read headers

In this step you make changes to your agent code so that you can decode and extract claims from a JWT token using PyJWT library.

requirements.txt

Add PyJWT dependency to your requirements.txt file.

```
PyJWT
```

agent_example.py

Change your main agent code as shown in the following code. You can decode without validation since your token was already validated by AgentCore Runtime when authorization was done.

```

import jwt
import json
....

@app.entrypoint
def invoke(payload, context):
    auth_header = context.request_headers.get('Authorization')
    if not auth_header:
        return None
    # Remove "Bearer " prefix if present
    token = auth_header.replace('Bearer ', '') if auth_header.startswith('Bearer ')
 else auth_header
    try:
        claims = jwt.decode(token, options={"verify_signature": False})
        app.logger.info("Claims: %s", json.dumps(claims))
    except jwt.InvalidTokenError as e:
        app.logger.exception("Invalid JWT token: %s", e)

    .....
```

Step 6.2: Create the agent with request header allowlist

Use the AgentCore starter toolkit to create the agent with request header allowlist.

```
agentcore configure --entrypoint agent_example.py \
 --name hello_agent \
--execution-role your-execution-role-arn \
--disable-otel \
--requirements-file requirements.txt \
--authorizer-config "{\"customJWTAuthorizer\":{\"discoveryUrl\":\"$DISCOVERY_URL\",
\"allowedClients\":[\"$CLIENT_ID\"]}}" \
--request-header-allowlist "Authorization"

// now launch the agent runtime
agentcore launch
```

Step 6.3: Invoke your agent

Invoke your agent using OAuth and you should see the claims in your agent logs in CloudWatch Logs.

Troubleshooting

How to debug token related issues

If you encounter issues with token authentication, you can decode the token to inspect its contents:

```
echo "$TOKEN" | cut -d '.' -f2 | tr '_-' '/+' | awk '{ l=4 - length($0)%4; if (l<4)
 printf "%s", $0; for (i=0; i<l; i++) printf "="; print "" }' | base64 -D | jq
```

This will output the token's payload, which looks similar to:

```
{
    "sub": "subid",
    "iss": "https://cognito-idp.us-east-1.amazonaws.com/userpoolid",
    "client_id": "clientid",
    "origin_jti": "originjti",
    "event_id": "eventid",
    "token_use": "access",
    "scope": "aws.cognito.signin.user.admin",
    "auth_time": 1752275688,
    "exp": 1752279288,
    "iat": 1752275688,
    "jti": "jti",
    "username": "username"
}
```

When troubleshooting token issues, check the following:

- Issuer url pointed to by the discovery url in the agent authorizer should match the issuer claim in the token. Do the following to confirm they match:
  - Select the discovery url you provided in the authorizer configuration when you created the agent, for example: `https://cognito-idp.us-east-1.amazonaws.com/us-east-1_nnnnnnnn/.well-known/openid-configuration`
    - Check the issuer url - `"issuer": "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_12345566"`. This should match the iss claim value in the token.
- `client_id` claim in the token must match one of the authorizer allowedClients entries if provided
  - Note the client id you provided when you created the agent
  - Confirm this matches the client_id claim in the decoded token
- `aud` claim in the token must match one of the authorizer `allowedAudience` entries, if provided
  - Note the audience list you provided when you created the agent
  - Confirm this matches the `aud` claim in the decoded token
- Tokens are only valid for several minutes (the default Amazon Cognito expiry is 60 minutes). Fetch a new token as needed.

AgentCore Runtime versioning and endpoints

Understanding agent runtime Versioning

Each agent runtime in Amazon Bedrock AgentCore is automatically versioned:

- When you create an agent runtime, AgentCore Runtime automatically creates version 1 (V1)
- Each update to the agent runtime creates a new version with a complete, self-contained configuration
- Versions are immutable once created
- Each version contains all the configuration needed for execution

How endpoints reference versions

Endpoints provide a way to reference specific versions of your agent runtime:

- The "DEFAULT" endpoint automatically points to the latest version of your agent runtime
- Endpoints can point to specific versions, allowing you to maintain different environments (e.g., development, staging, production)
- When you update an agent runtime, the "DEFAULT" endpoint is automatically updated to point to the new version
- Endpoints must be explicitly updated to point to new versions

Example Updating an endpoint to a New Version

```
bedrock_agentcore_client = boto3.client('bedrock-agentcore', region_name='us-west-2')

response = bedrock_agentcore_client.update_agent_runtime_endpoint(
    agentRuntimeId='agent-runtime-12345',
    endpointName='production-endpoint',
    agentRuntimeVersion='v2.1',
    description='Updated production endpoint'
)

print(response)
```

Versioning scenarios

The following table illustrates how versioning and endpoints interact during the lifecycle of an agent runtime:

Agent Runtime Versioning Scenarios

Change Type	Version Creation Behavior	Latest Version	Endpoint Behavior
Initial Creation	Creates Version 1 (V1) automatically	V1	DEFAULT points to V1
Protocol Change	Creates a new version with updated protocol settings	V2	DEFAULT automatically updates to V2
Create "PROD" endpoint with V2	No new version created	V2	PROD endpoint points to V2
Container Image Update	Creates a new version with new container reference	V3	DEFAULT updates to V3, PROD remains on V2
Update "PROD" to V3	No new version created	V3	PROD updates to V3
Network Settings Modification	Creates a new version with updated security parameters	V4	DEFAULT updates to V4, PROD remains on V3

Endpoint lifecycle states

AgentCore Runtime endpoints go through various states during their lifecycle:

CREATING

Initial state when an endpoint is being created

CREATE_FAILED

Indicates creation failure due to permissions, container, or other issues

READY

Endpoint is ready to accept requests

UPDATING

Endpoint is being updated to a new version

UPDATE_FAILED

Indicates update operation failure

Listing AgentCore Runtime versions and endpoints

You can list all versions of an AgentCore Runtime by calling the `ListAgentRuntimeVersions` operation. To list the endpoints for an AgentCore Runtime, call `ListAgentRuntimeEndpoints`.

Invoke an AgentCore Runtime agent

The `InvokeAgentRuntime` operation lets you send requests to specific AgentCore Runtime endpoints identified by their Amazon Resource Name (ARN) and receive streaming responses containing the agent's output. The API supports session management through session identifiers, enabling you to maintain conversation context across multiple interactions. You can target specific agent endpoints using optional qualifiers.

To call `InvokeAgentRuntime`, you need `bedrock-agentcore:InvokeAgentRuntime` permissions. In the call you can also pass a bearer token that the agent can use for user authentication.

The `InvokeAgentRuntime` operation accepts your request payload as binary data up to 100 MB in size and returns a streaming response that delivers chunks of data in real-time as the agent processes your request. This streaming approach allows you to receive partial results immediately rather than waiting for the complete response, making it ideal for interactive applications.

If you plan on integrating your agent with OAuth, you can't use the AWS SDK to call `InvokeAgentRuntime`. Instead, make a HTTPS request to InvokeAgentRuntime. For more information, see Authenticate and authorize with Inbound Auth and Outbound Auth.

Invoke streaming agents

The following example shows how to use boto3 to invoke an agent runtime:

```
import boto3
import json

# Initialize the Bedrock AgentCore client
agent_core_client = boto3.client('bedrock-agentcore')

# Prepare the payload
payload = json.dumps({"prompt": prompt}).encode()

# Invoke the agent
response = agent_core_client.invoke_agent_runtime(
    agentRuntimeArn=agent_arn,
    runtimeSessionId=session_id,
    payload=payload
)


# Process and print the response
if "text/event-stream" in response.get("contentType", ""):

    # Handle streaming response
    content = []
    for line in response["response"].iter_lines(chunk_size=10):
        if line:
            line = line.decode("utf-8")
            if line.startswith("data: "):
                line = line[6:]
                print(line)
                content.append(line)
    print("\nComplete response:", "\n".join(content))

elif response.get("contentType") == "application/json":
    # Handle standard JSON response
    content = []
    for chunk in response.get("response", []):
        content.append(chunk.decode('utf-8'))
    print(json.loads(''.join(content)))
```

```
else:
    # Print raw response for other content types
    print(response)
```

Invoke multi-modal agents

You can use the `InvokeAgentRuntime` operation to send multi-modal requests that include both text and images. The following example shows how to invoke a multi-modal agent:

```
import boto3
import json
import base64

# Read and encode image
with open("image.jpg", "rb") as image_file:
    image_data = base64.b64encode(image_file.read()).decode('utf-8')

# Prepare multi-modal payload
payload = json.dumps({
    "prompt": "Describe what you see in this image",
    "media": {
        "type": "image",
        "format": "jpeg",
        "data": image_data
 }
}).encode()

# Invoke the agent
response = agent_core_client.invoke_agent_runtime(
    agentRuntimeArn=agent_arn,
    runtimeSessionId=session_id,
    payload=payload
)
```

Session management

The `InvokeAgentRuntime` operation supports session management through the `runtimeSessionId` parameter. By providing the same session identifier across multiple requests, you can maintain conversation context, allowing the agent to reference previous interactions.

To start a new conversation, generate a unique session identifier. To continue an existing conversation, use the same session identifier from previous requests. This approach enables you to build interactive applications that maintain context over time.

> **Tip**
>
> For best results, use a UUID or other unique identifier for your session IDs to avoid collisions between different users or conversations.

Error handling

When using the `InvokeAgentRuntime` operation, you might encounter various errors. Here are some common errors and how to handle them:

ValidationException

Occurs when the request parameters are invalid. Check that your agent ARN, session ID, and payload are correctly formatted.

ResourceNotFoundException

Occurs when the specified agent runtime cannot be found. Verify that the agent ARN is correct and that the agent exists in your AWS account.

AccessDeniedException

Occurs when you don't have the necessary permissions. Ensure that your IAM policy includes the `bedrock-agentcore:InvokeAgentRuntime` permission.

ThrottlingException

Occurs when you exceed the request rate limits. Implement exponential backoff and retry logic in your application.

Implement proper error handling in your application to provide a better user experience and to troubleshoot issues effectively.

Best practices

Follow these best practices when using the `InvokeAgentRuntime` operation:

- Use session management to maintain conversation context for a better user experience.
- Process streaming responses incrementally to provide real-time feedback to users.
- Implement proper error handling and retry logic for a robust application.
- Consider payload size limitations (100 MB) when sending requests, especially for multi-modal content.
- Use appropriate qualifiers to target specific agent versions or endpoints.
- Implement authentication mechanisms when necessary using bearer tokens.

Observe agents in Amazon Bedrock AgentCore Runtime

For information about the AgentCore Runtime observability metrics, see Add observability to your Amazon Bedrock AgentCore resources.

Troubleshoot AgentCore Runtime

This troubleshooting topic helps you identify and resolve common issues when working with AgentCore Runtime. By following these solutions, you can quickly diagnose and fix problems with your agent runtimes.

Topics

- I need help understanding HTTP error codes
- I need recommendations for testing my agent
- I need help debugging container issues
- I need help troubleshooting MCP protocol agents
- Best practices

My agent invocations fail with 504 Gateway Timeout errors

When this occurs: During agent invocation via SDK or console

Why this happens: Multiple factors can prevent your agent from responding within the timeout period

Several factors can cause this:

- **Container Issues:** Make sure your Docker image exposes port 8080 and has the `/invocations` path
- **ARM64 Compatibility:** Currently your container must be ARM64 compatible
- **Retry Logic:** Review retry mechanisms for handling transient issues

My Docker build fails with "403 Forbidden" when pulling Python base images

When this occurs: During `docker build` or `docker run` when using `public.ecr.aws` base images

Why this happens: ECR Public authentication issues — expired or missing authentication is a common issue.

Solution: Either login to ECR Public or logout completely:

```
# Option 1: Login to ECR Public
aws ecr-public get-login-password --region us-east-1 | docker login --username AWS --
password-stdin public.ecr.aws
```

```
# Option 2: Logout (recommended for avoiding token expiration)
docker logout public.ecr.aws

# Option 3: Use Docker Hub directly in Dockerfile
FROM python:3.10-slim
# instead of public.ecr.aws/docker/library/python:3.10-slim
```

I get "Unknown service: 'bedrock-agent-core-runtime'" error when using boto3

When this occurs: When invoking Amazon Bedrock AgentCore APIs using boto3 SDK

Why this happens: Outdated boto3 library — common issue as most installations don't have latest SDK

Solution: Update to latest boto3 and botocore versions:

```
pip install --upgrade boto3 botocore

# Minimum versions: boto3 1.39.8+, botocore 1.33.8+
```

I get "AccessDeniedException" when trying to create an Amazon Bedrock AgentCore Runtime

When this occurs: During agent creation via console, SDK, or CLI

Why this happens: Either your user lacks permissions, or the execution role isn't properly configured for Amazon Bedrock AgentCore

Solution: Several factors can cause this:

- **Missing permissions for the caller.** Make sure that the caller's credentials has `bedrock-agentcore:CreateAgentRuntime`.
- **Execution Role cannot be assumed by Bedrock Amazon Bedrock AgentCore.** Make sure that the execution role follows this guidance on permissions for Amazon Bedrock AgentCore Runtime execution role.

I get "Unknown service: 'bedrock-agent-core-runtime'" error when using boto3 89

My Docker build fails with "exec /bin/sh: exec format error"

When this occurs: When building containers for Amazon Bedrock AgentCore deployment

Why this happens: Building ARM64 containers on x86 systems without proper cross-platform setup

Solution: Build ARM64 compatible containers. You can consider using buildx for cross-platform builds. Alternatively, you can use CodeBuild. For example code, see the Amazon Bedrock AgentCore Samples.

What are the requirements for Docker containers used with Amazon Bedrock AgentCore Runtime?

Review Amazon Bedrock AgentCore Runtime requirements for full details.

In summary, your Docker container must meet these requirements:

- **Port:** Expose port 8080 (additional ports will be supported soon)
- **Endpoint:** Must have `/invocations` path available
- **Architecture:** Must be ARM64 compatible
- **Response:** Should handle the expected payload format

My long-running tool gets interrupted after 15 minutes

For information, see Handle asynchronous and long running agents with Amazon Bedrock Amazon Bedrock AgentCore Runtime for full details.

When this occurs: During long-running agent operations or complex workflows

Why this happens: Amazon Bedrock AgentCore automatically terminates sessions after 15 minutes of inactivity

Example solution: Implement ping handlers with HEALTHY_BUSY status for async tasks:

```
import asyncio
from bedrock_agentcore.runtime import BedrockAgentCoreApp
```

My Docker build fails with "exec /bin/sh: exec format error" 90

```
app = BedrockAgentCoreApp()

@app.entrypoint
async def long_running_agent(payload, context):
    # For long-running tasks, create async task with ping handler
    async def ping_handler():
        while task_running:
            await context.ping(status="HEALTHY_BUSY")
            await asyncio.sleep(30)  # Ping every 30 seconds

    # Start ping handler
    ping_task = asyncio.create_task(ping_handler())

    # Your long-running work here
    result = await perform_long_task()

    # Clean up
    ping_task.cancel()
    return result
```

How do I access the runtimeSessionId in my agent code for tagging or grouping resources?

When this applies: You want to group, tag, or trace resources (e.g., S3 objects, logs) by the current agent runtime session.

Solutions:

- If you're using the Bedrock Agents SDK, use `context.session_id`.
- If you're building a custom runtime server, extract it from the `X-Amzn-Bedrock-AgentCore-Runtime-Session-Id` HTTP header.

Solution 1: For agents using the Bedrock Amazon Bedrock AgentCore SDK, use `context.session_id` from your agent entrypoint

```
@app.entrypoint
def my_agent(payload, context):
    session_id = context.session_id
    # Use session_id for S3 object tagging/organization
    s3_client = boto3.client('s3')
    s3_client.put_object(
        Bucket='my-bucket',
        Key=f'agent-outputs/{session_id}/output.json',
        Body=json.dumps(result),
        Tagging=f'SessionId={session_id}'
    )
    return result
```

Solution 2: For custom runtime HTTP servers

The runtime session ID is passed in this HTTP header. Parse it from the incoming request and use it for tagging, correlation, or downstream propagation.

```
X-Amzn-Bedrock-AgentCore-Runtime-Session-Id: <value>
```

I have RuntimeClientError (403) issues

Problem

You receive a 403 "RuntimeClientError" when attempting to invoke your agent runtime.

Causes

This error typically occurs due to:

- Container startup failures
- Permissions issues with execution role
- Authentication issues with bearer token

Resolution

Follow these steps to resolve the issue:

1. **Check CloudWatch Logs**: Any issues with starting up the container will reflect as a 403 - RuntimeClientError. Navigate to the following CloudWatch log group to check for startup errors:

```
/aws/bedrock-agentcore/runtimes/<agent_id>-<endpoint_name>/[runtime-logs]
```

2. **Verify Execution Role**: Ensure your agent's execution role has the necessary permissions. For more information, see AgentCore Runtime execution role.
3. **Validate Authentication**: For MCP protocol agents, ensure your bearer token is valid and not expired.

I have missing or empty CloudWatch Logs

Problem

You encounter errors but don't see any relevant logs in CloudWatch.

Solution

Try these approaches to diagnose the issue:

1. **Check Correct Log Group**: Ensure you're looking in the right CloudWatch log group. The standard pattern is:

   ```
   /aws/bedrock-agentcore/runtimes/<agent_id>-<endpoint_name>/runtime-logs
   ```

2. **Run Locally for Diagnostics**: If there are no CloudWatch Logs, try running the agent container locally using the exact same payload you used for invocation in AgentCore Runtime. This can help identify issues that might not be visible in the logs.
3. **Enable Verbose Logging**: Update your agent code to include more detailed logging, especially around the entry points and any error handling logic.

I have payload format issues

Problem

Your agent runtime invocation fails even though the container starts successfully.

Resolution

Follow these steps to resolve payload format issues:

1. **Verify Payload Structure**: Ensure your payload structure matches what your agent expects. Pay special attention to:

   - If your agent code expects `input` keyword in the payload, make sure to include it:

     ```
     {
         "input": {
             "prompt": "Your question here"
         }
     }
     ```

   - Not just:

     ```
     {
          "prompt": "Your question here"
     }
     ```

2. **Check Documentation**: Review the expected input format in the documentation.

I need help understanding HTTP error codes

Problem

Your agent returns HTTP error codes that are difficult to interpret.

Resolution

Here are the most common error codes and their meanings:

422 Unprocessable Entity

This happens when the container encounters validation issues with the input payload.

Common causes:

- Missing required fields in the payload (e.g., missing "input" field)
- Incorrect data types for fields
- Invalid format for the payload

403 Forbidden

Authentication or authorization issues.

Check your bearer token or IAM permissions.

500 Internal Server Error

Runtime exceptions in your agent code.

Check CloudWatch logs for detailed stack traces.

I need recommendations for testing my agent

To systematically debug agent runtime issues:

Test locally first

Before deploying to AgentCore Runtime:

- Run your agent container locally using the same Docker image
- Verify it works with the exact same payload

Compare payloads

Ensure consistency between environments:

- Ensure the payload structure between local testing and AgentCore Runtime invocation is identical
- Pay special attention to nesting of fields like "input" and "prompt"

I need help debugging container issues

If you suspect container-related issues:

Pull and run locally

Test your container image on your local machine:

```
docker pull <your-ecr-repo-uri>
docker run -p 8080:8080 <your-ecr-repo-uri>
```

Test with curl

Send test requests to your local container:

```
curl -X POST http://localhost:8080/invocations \
     -H "Content-Type: application/json" \
     -d '{"input": {"prompt": "Hello world!"}}'
```

Check container logs

Examine the container's output for errors:

```
docker logs <container-id>
```

I need help troubleshooting MCP protocol agents

For MCP protocol agents, follow these specific troubleshooting steps:

Verify endpoint path

MCP servers should listen on `0.0.0.0:8000/mcp/`

Use MCP Inspector

Test with the MCP Inspector tool:

1. Install and run the MCP Inspector: `npx @modelcontextprotocol/inspector`
2. Connect to your local server at `http://localhost:8000/mcp`
3. For deployed agents, use the properly URL-encoded endpoint

Authentication issues

Check authentication configuration:

- Ensure bearer token is correctly set in the headers
- Verify your Cognito user pool is correctly set up

Best practices

Enable comprehensive logging

Implement thorough logging in your agent:

- Include request/response logging in your agent
- Log critical paths and error conditions

Use structured error handling

Implement clear error reporting:

- Return clear error messages with specific codes
- Include actionable information in error responses

Test incremental changes

Follow a methodical testing approach:

- When modifying your agent, test locally before deployment
- Validate payload compatibility with both local and deployed environments

Monitor performance

Set up monitoring for your agent:

- Use CloudWatch metrics to track invocation patterns
- Set up alarms for error rates and latency

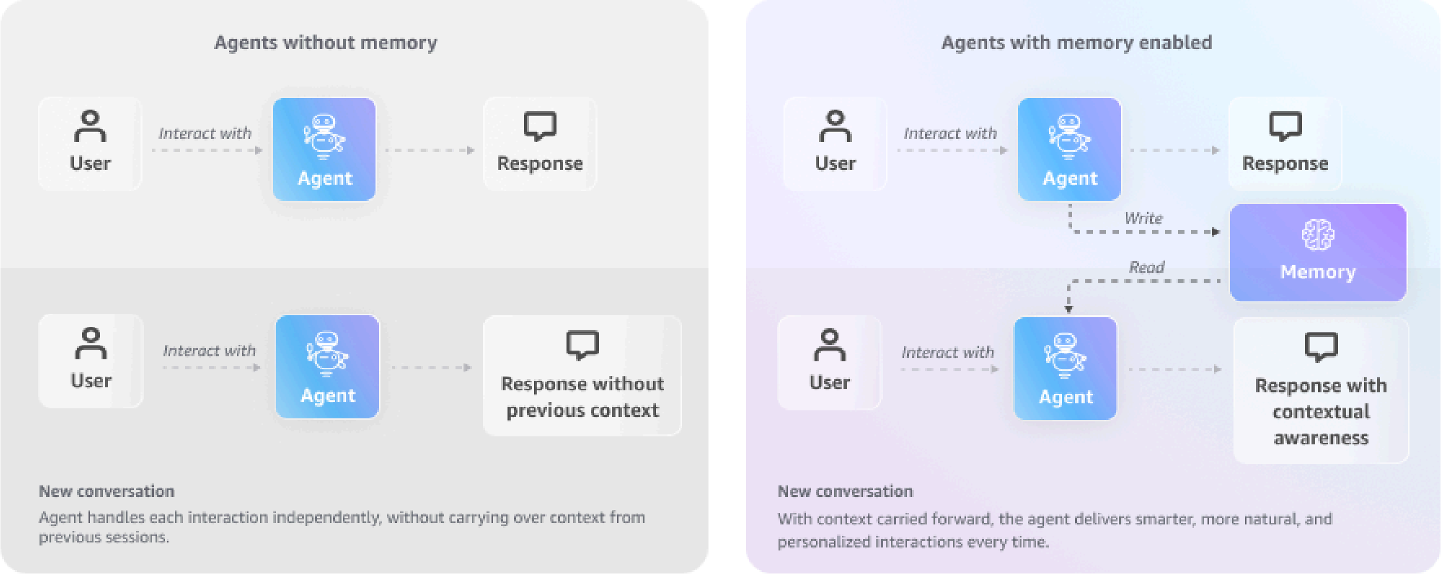
Add memory to your AI agent

AgentCore Memory lets your AI agents deliver intelligent, context-aware, and personalized interactions by maintaining both immediate and long-term knowledge. AgentCore Memory offers two types of memory:

- **Short-term memory**: Stores conversations to keep track of immediate context.

  For example, imagine your coding agent is helping you debug. During the session, you ask it to check variable names, correct syntax errors, and find unused imports. The agent stores the interactions as short term events in AgentCore Memory. Later the agent can retrieve the events so that it can converse without you having to repeat previous information.

  Short-term memory captures raw interaction events, maintains immediate context, powers real-time conversations, enriches long-term memory systems, and enables building advanced contextual solutions such as multi-step task completion, in-session knowledge accumulation, and context-aware decision making.
- **Long-term memory**: Stores extracted insights - such as user preferences, semantic facts, and summaries - for knowledge retention across sessions.
  - **User Preferences –** Think of your coding agent which uses AgentCore Memory as your long-time coding partner. Over many days, it notices you always write clean code with comments, prefer snake_case naming, use pandas for data analysis, and test functions before finalizing them. Next time, even after many sessions, when you ask it to write a data analysis function, it automatically follows these preferences stored in AgentCore Memory without you telling it again.
  - **Semantic facts –** The coding agent also remembers that “Pandas is a Python Library for data analysis and handling tables”. When you ask, “Which library is best for table data?”, it immediately suggests Pandas because it understands what Pandas are from the semantic memory.
  - **Summary –** The coding agent generates session summaries such as “During this interaction, you created a data cleaning function, fixed two syntax errors, and tested your linear regression model.” These summaries both track completed work and compress conversation context, enabling efficient reference to past activities while optimizing context window usage.


Agents without memory
User
Interact with
Agent
Response
User
Interact with
Agent
Response without previous context
New conversation
Agent handles each interaction independently, without carrying over context from previous sessions.
Agents with memory enabled
User
Interact with
Agent
Response
Write
Memory
Read
User
Interact with
Agent
Response with contextual awareness
New conversation
With context carried forward, the agent delivers smarter, more natural, and personalized interactions every time.

You can use AgentCore Memory with the AWS SDK or with any popular agent framework, such as Strands Agents. For code examples, see https://github.com/awslabs/amazon-bedrock-agentcore-samples/tree/main/01-tutorials/04-AgentCore-memory.

Topics

- How it works
- Getting started with AgentCore Memory
- Configure AgentCore Memory
- Store and use short-term memory
- Store and use long-term memory

How it works

AgentCore Memory provides APIs that let AI agents store, retrieve, and use memory effectively.

Topics

- Short-term memory
- Long-term memory
- Putting it all together: A customer support AI agent

Short-term memory

Short-term memory stores raw interactions that help the agent maintain context within a single session. For example, in a shopping website's customer support AI agent, short-term memory captures the entire conversation history as a series of events. Each customer question and agent response is saved as a separate event (or in batches, depending on your implementation). This allows the agent to reload the conversation as it happened, maintaining context even if the service restarts or the customer returns later to continue the same interaction seamlessly.

When a customer interacts with your agent, each interaction can be captured as an event using the `CreateEvent` operation. Events can contain various types of data, including conversational exchanges (questions, answers, instructions) or structured information (product details, order status). Each event is attached to a session using either defined session identifier of your choosing or a default session identifier is generated. You can use this sessionId in future requests to maintain conversation context.

To load previous sessions/conversations or enrich context, your agent needs to access the raw interactions with the customer. Imagine a customer returns to follow up on their product support case from last week. To provide seamless assistance, the agent uses `ListSessions` to locate their previous support interactions. Through `ListEvents`, it retrieves the conversation history, understanding the reported issue, troubleshooting steps attempted, and any temporary solutions discussed. The agent uses `GetEvent` to access specific information from key moments in past conversations. These operations work together to maintain support continuity across sessions, eliminating the need for customers to re-explain their issue or repeat troubleshooting steps already attempted.

Long-term memory

Long-term memory stores structured information extracted from raw agent interactions, which is retained across multiple sessions. Instead of saving all raw conversation data, long-term memory preserves only the key insights such as summaries of the conversations, facts and knowledge (semantic memory), or user preferences. For example, if a customer tells the agent their preferred shoe brand during a chat, the AI agent stores this as a long-term memory. Later, even in a different conversation, the agent can remember and suggest the shoe brand, making the interaction personalized and relevant.

In AgentCore Memory, you can add memory strategies as part of `CreateMemory` operation which decides what information to extract from raw conversations. These strategies are configurations that intelligently capture main concepts from interactions (sent as events in the `CreateEvent` operation) and persists them. The strategies currently supported are summarization, semantic memory (facts and knowledge), and user preferences. AgentCore Memory allows two ways to add these strategies to the memory:

- Built-in strategies (managed by AgentCore and runs in a service-managed account)
- Custom strategies that let you append to the system prompt and choose the model. This lets you tailor the memory extraction and consolidation to your specific domain or use case

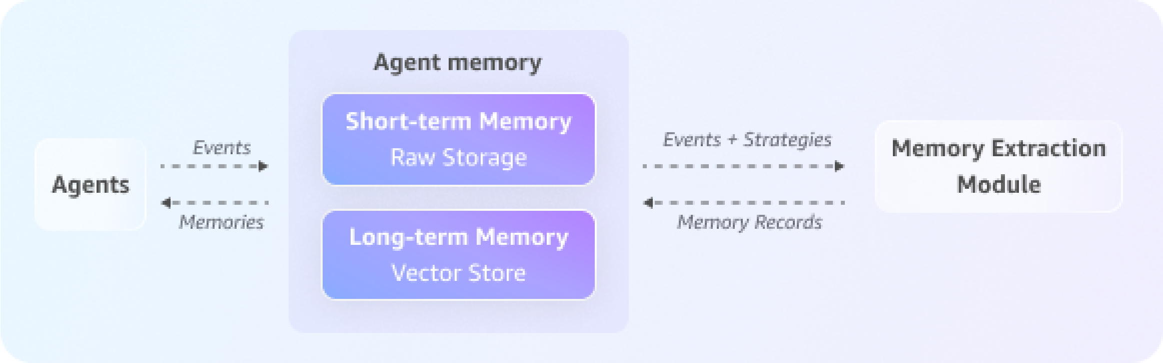
Long-term memory generation is an asynchronous process that runs in the background and automatically extracts insights after raw conversation/context is stored in Short Term Memory via `CreateEvent`. This efficiently consolidates key information without interrupting live interactions. As part of the long-term memory generation, AgentCore Memory performs the following operations:

- **Extraction**: Extracts information from raw interactions with the agent
- **Consolidation**: Consolidates newly extracted information with existing information in the AgentCore Memory.

Once long-term memory is generated, you can retrieve these extracted memories to enhance your agent's responses with persistent knowledge. Extracted memories are stored as memory records and can be accessed using the `GetMemoryRecord`, `ListMemoryRecords`, or `RetrieveMemoryRecords` operations. The `RetrieveMemoryRecords` operation is particularly powerful as it performs a semantic search to find memory records that are most relevant to the query. For example, when a customer asks about running shoes, the agent can use semantic search to retrieve related memory records, such as customer's preferred shoe size, favorite shoe brands, and previous shoe purchases. This lets the AI support agent provide highly personalized recommendations without requiring the customer to repeat information they've shared before.

Putting it all together: A customer support AI agent

Consider a customer, Sarah, who engages with your shopping website's support AI agent to inquire about a delayed order.



The interaction flow through the AgentCore Memory APIs would look like this:

- Create a short term and long term memory and add strategies to create your long term memory.
- **Starting the session:** When Sarah initiates the conversation, the agent creates a new, and unique, session ID to track this interaction separately.
- **Capturing conversation history:** As Sarah explains her issue, each message (both her questions and agent's responses) is saved as an event using `CreateEvent` operation ensuring the full conversation is recorded in sequence.
- **Generating long-term memory:** In the background, the asynchronous extraction process runs every few turns. This process analyzes the recent raw events using built-in or custom memory strategies (that you had configured when setting up AgentCore Memory through `CreateMemory` operation) to extract long-term memories such as summaries, semantic facts, or user preferences, which are then stored for future use.
- **Retrieving past interactions from short-term memory:** To provide context-aware assistance, the agent calls `ListEvents` to load conversation histories. This helps the agent understand what issues Sarah has raised before.
- **Using long-term memories for personalized assistance:** The agent calls `RetrieveMemoryRecords`, which performs a semantic search across extracted long-term memories to find relevant insights about Sarah's preferences, order history, or past concerns. This lets the agent provide highly personalized assistance without needing to ask Sarah to repeat information she has already shared in previous chats.

This integrated approach allows the agent to maintain rich context across sessions, recognize returning customers, recall important details, and deliver personalized experiences seamlessly, resulting in faster, more natural, and effective customer support.

Getting started with AgentCore Memory

Amazon Bedrock AgentCore Memory lets you create and manage memory resources that store conversation context for your AI agents. This getting started section guides you through installing dependencies and implementing both short-term and long-term memory features.

Topics

- Create an AgentCore Memory resource
- Maintain user context using short-term memory
- Create a memory with a long-term memory
- Use long-term memory in an agent
- Custom strategies

Create an AgentCore Memory resource

Learn how to install dependencies and create an AgentCore Memory resources for your AI agents.

Topics

- Install dependencies
- Create memory for short-term memory
- List existing memory resources

Install dependencies

To get started with Amazon Bedrock AgentCore Memory, install the Amazon Bedrock AgentCore Python SDK:

```
pip install bedrock-agentcore
```

Create memory for short-term memory

Adding short-term memory is a quick, one-time setup process. Short-term memory maintains context without persisting historical data. This is useful for tracking current conversation flow, such as customer support interactions. Note that for short-term memory, you don't need to add a memory strategy which is used to extract memories for long-term storage.

Example Create short-term memory using the Bedrock AgentCore SDK

```
from bedrock_agentcore.memory import MemoryClient

client = MemoryClient(region_name="us-west-2")

memory = client.create_memory(
    name="CustomerSupportAgentMemory",
    description="Memory for customer support conversations",
)

# The memory_id will be used in following operations
print(f"Memory ID: {memory.get('id')}")
print(f"Memory: {memory}")
```

List existing memory resources

If you already have existing memory resources created in Amazon Bedrock AgentCore Memory, you can list them to find their identifiers:

```
for memory in client.list_memories():
    print(f"Memory Arn: {memory.get('arn')}")
    print(f"Memory ID: {memory.get('id')}")
    print("--------------------------------------------------------------------")
```

Maintain user context using short-term memory

Learn how to store and retrieve conversation context using short-term memory.

Topics

- Create events in short-term memory
- Load conversations from short-term memory

Create events in short-term memory

The `create_event` action stores agent interactions into short-term memory instantly. You can save conversations either one turn at a time or in batches, depending on your application needs. Each saved interaction can include user messages, assistant responses, and tool actions. The process is synchronous, ensuring no conversation data is lost.

```
client.create_event(
    memory_id=memory.get("id"), # This is the id from create_memory or list_memories
    actor_id="User84",  # This is the identifier of the actor, could be an agent or
 end-user.
    session_id="OrderSupportSession1", #Unique id for a particular request/
conversation.
    messages=[
        ("Hi, I'm having trouble with my order #12345", "USER"),
        ("I'm sorry to hear that. Let me look up your order.", "ASSISTANT"),
        ("lookup_order(order_id='12345')", "TOOL"),
        ("I see your order was shipped 3 days ago. What specific issue are you
 experiencing?", "ASSISTANT"),
        ("Actually, before that - I also want to change my email address", "USER"),
        (
            "Of course! I can help with both. Let's start with updating your email.
 What's your new email?",
            "ASSISTANT",
        ),
        ("newemail@example.com", "USER"),
        ("update_customer_email(old='old@example.com', new='newemail@example.com')",
 "TOOL"),
        ("Email updated successfully! Now, about your order issue?", "ASSISTANT"),
        ("The package arrived damaged", "USER"),
    ],
)
```

Load conversations from short-term memory

The `list_events` method loads conversations from short-term memory using the memory_id, actor_id and session_id. The process is synchronous and returns the conversation data:

```
conversations = client.list_events(
    memory_id=memory.get("id"),
    actor_id="User84",
    session_id="OrderSupportSession1",
    max_results=5,
)
```

Create a memory with a long-term memory

Learn how to set up long-term memory to extract and store information from conversations.

Topics

Create memory with long-term memory

With long-term memory, you can extract and store information from conversations for future use. When you add long-term memory, you can use one of the following strategies:

- **User Preferences (UserPreferenceMemoryStrategy)**: Stores and learns recurring patterns in user behavior, interaction styles, and choices. This enables the agent to automatically adapt its responses to match user preferences across multiple sessions.
- **Semantic Facts (SemanticMemoryStrategy)**: Maintains knowledge of facts and domain-specific information, technical concepts, and their relationships. This allows the agent to provide informed responses based on previously established context and understanding.
- **Session Summaries (SummaryMemoryStrategy)**: Creates condensed representations of interaction content and outcomes. These summaries provide quick reference points for past activities and help optimize context window usage for future interactions.

To create a memory resource with long-term memory, use the `create_memory_and_wait` method with a strategy. Long-term memory takes 2-3 minutes to become ACTIVE:

```
memory = client.create_memory_and_wait(
    name="MyAgentMemory",
    strategies=[{
        "summaryMemoryStrategy": {
            # Name of the extraction model/strategy
            "name": "SessionSummarizer",
            # Organize facts by session ID for easy retrieval
            # Example: "summaries/session123" contains summary of session123
            "namespaces": ["/summaries/{actorId}/{sessionId}"]
        }
    }]
)
```

If you are already using short-term memory, you can upgrade to use long-term memory by adding a strategy to the existing memory resource:

```
summary_strategy = client.add_summary_strategy(
    memory_id = memory.get("id"),
    name="SessionSummarizer",
    description="Summarizes conversation sessions",
    namespaces=["/summaries/{actorId}/{sessionId}"] #Namespace allow you to organize
 all extracted information. This template will extract information for each sessionId
 belonging to an actor in separate namespace
)
```

> **Note**
>
> Long-term memory records will only be extracted from events that are stored after the newly added strategies become ACTIVE. Conversations stored before a strategy is added will not appear in long-term memory.

Save conversations and view extracted memories

The following example demonstrates how to save a conversation and retrieve its automatically extracted memories. After saving the conversation, we wait for 1 minute to allow the long-term memory strategies to process and extract meaningful information before retrieving it.

```
import time

event = client.create_event(
    memory_id=memory.get("id"), # This is the id from create_memory or list_memories
    actor_id="User84",  # This is the identifier of the actor, could be an agent or
 end-user.
    session_id="OrderSupportSession1",
    messages=[
        ("Hi, I'm having trouble with my order #12345", "USER"),
        ("I'm sorry to hear that. Let me look up your order.", "ASSISTANT"),
        ("lookup_order(order_id='12345')", "TOOL"),
        ("I see your order was shipped 3 days ago. What specific issue are you
 experiencing?", "ASSISTANT"),
        ("Actually, before that - I also want to change my email address", "USER"),
        (
            "Of course! I can help with both. Let's start with updating your email.
 What's your new email?",
            "ASSISTANT",
        ),
        ("newemail@example.com", "USER"),
        ("update_customer_email(old='old@example.com', new='newemail@example.com')",
 "TOOL"),
        ("Email updated successfully! Now, about your order issue?", "ASSISTANT"),
        ("The package arrived damaged", "USER"),
    ],
)

# Wait for meaningful memories to be extracted from the conversation.
time.sleep(60)

# Query for the summary of the issue using the namespace set in summary strategy above
memories = client.retrieve_memories(
    memory_id=memory.get("id"),
    namespace=f"/summaries/User84/OrderSupportSession1",
    query="can you summarize the support issue"
)
```

Use long-term memory in an agent

Learn how to integrate long-term memory with an agent to enhance its capabilities.

Topics

- Install dependencies
- Add memory to an agent

Install dependencies

```
pip install strands-agents
```

Add memory to an agent

```
from strands import tool, Agent
from strands_tools.agent_core_memory import AgentCoreMemoryToolProvider
import time
from bedrock_agentcore.memory import MemoryClient

client = MemoryClient(region_name="us-west-2")
memory = client.create_memory_and_wait(
    name="MyAgentMemory",
    strategies=[{
        "userPreferenceMemoryStrategy": {
            "name": "UserPreference",
            "namespaces": ["/users/{actorId}"]
        }
    }]
)

strands_provider = AgentCoreMemoryToolProvider(
    memory_id=memory.get("id"),
    actor_id="CaliforniaPerson",
    session_id="TalkingAboutFood",
    namespace="/users/CaliforniaPerson",
    region="us-west-2"
)
agent = Agent(tools=strands_provider.tools)

agent("Im vegetarian and I prefer restaurants with a quiet atmosphere.")
agent("Im in the mood for Italian cuisine.")
agent("Id prefer something mid-range and located downtown.")
agent("I live in Irvine.")

time.sleep(60)

# This will use the long-term memory tool
agent("I dont remember what I was in a mood for, do you remember?")
```

Custom strategies

You can customize existing strategies by specifying your own prompt. This allows you to specify the exact information you want to extract. In the example below, you create a custom prompt to extract a user's preference about their airline needs.

Topics

- Create an IAM role for the service
- Create a long-term memory with a custom strategy
- Create events to upload user conversations
- Search for user's preferences

Create an IAM role for the service

Start by making sure you have an IAM role with the managed policy AmazonBedrockAgentCoreMemoryBedrockModelInferenceExecutionRolePolicy, or create a policy with the following permissions:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream"
            ],
            "Resource": [
                "arn:aws:bedrock:*::foundation-model/*",
                "arn:aws:bedrock:*:*:inference-profile/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "123456789012"
                }
            }
        }
    ]
}
```

This role is assumed by the Service to call the model in your AWS account. Use the trust policy below when creating the role or when using the managed policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "bedrock-agentcore.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

For information about creating an IAM role, see IAM role creation.

Create a long-term memory with a custom strategy

```
from bedrock_agentcore.memory import MemoryClient

client = MemoryClient(region_name="us-west-2")

# Our custom prompt ensures that we're able to extract a customer's travel preferences.
CUSTOM_PROMPT = """\
You are tasked with analyzing conversations to extract the user's preferences. You'll
 be analyzing two sets of data:

<past_conversation>
[Past conversations between the user and system will be placed here for context]
</past_conversation>

<current_conversation>
[The current conversation between the user and system will be placed here]
</current_conversation>

Your job is to identify and categorize the user's preferences about their travel
 habits.
- Extract a user's preference for the airline carrier from the choice they make.
- Extract a user's preference for the seat type on the airline from the choice they
 make. It can aisle, middle or window
"""

# Replace the value with the role arn created above.
MEMORY_EXECUTION_ROLE_ARN = "arn:aws:iam::123456789012:role/MyRole"
```

```
memory = client.create_memory_and_wait(
    name="AirlineMemoryAgent",
    strategies=[{
        "customMemoryStrategy": {
            "name": "UserPreference",
            "namespaces": ["/users/{actorId}"],
            "configuration" : {
                "userPreferenceOverride" : {
                    "extraction" : {
                        "modelId" : "anthropic.claude-3-5-sonnet-20241022-v2:0",
                        "appendToPrompt": CUSTOM_PROMPT
                    }
                }
            }
        }
    }],
    memory_execution_role_arn=MEMORY_EXECUTION_ROLE_ARN
)
```

Create events to upload user conversations

```
event = client.create_event(
    memory_id=memory.get("id"), # This is the id from create_memory or list_memories
    actor_id="User84",  # This is the identifier of the actor, could be an agent or
 end-user.
    session_id="AirlineBookingSession1",
    messages=[
        ("Hi, I would like to book a flight from Seattle to New York for this Sunday",
 "USER"),
        ("Certainly, let me try to find the best flights for you", "ASSISTANT"),
        ("flight_search(start='Seattle', end='New York', date='2025-07-30')", "TOOL"),
        ("I have a two options available. 1/ Delta Airlines DL456 at 10:30 AM 2/
 American Airline AA345 at 4PM. ", "ASSISTANT"),
        ("Delta airline", "USER"),
        ("Sure. I will get you a seat on Delta flight DL456. Do you have a preference
 for a seat type","ASSISTANT",),
        ("Yes. Window please", "USER"),
        ("I have booked you on flight DL456 at 10:30 AM on 07/30/2025. Your seat number
 is 26A. You will more details in your email", "ASSISTANT"),
    ],
)
```

Search for user's preferences

```
memories = client.retrieve_memories(
    memory_id=memory.get("id"),
    namespace=f"/users/User84",
    query="What are the user's preferences for airline type ?"
)

memories = client.retrieve_memories(
    memory_id=memory.get("id"),
    namespace=f"/users/User84",
    query="What are the user's preferences for seat type ?"
)
```

Configure AgentCore Memory

When creating an AgentCore Memory, you can configure various settings such as name, optional description, encryption settings, expiration timestamp for raw agent interaction events, and memory strategies (if you want to extract long-term memory).

Prerequisites

Before you start using AgentCore Memory, these are some important prerequisites you need to know:

Topics

- Memory scoping with namespaces
- Memory strategies

Memory scoping with namespaces

With Long-Term Memory, organization is managed through Namespaces, which are defined when setting memory strategy in a call to the `CreateMemory` operation or with the AgentCore console. This section details these concepts.

An **actor** refers to entity such as end users or agent/user combinations. For example, in a coding support chatbot, the actor is usually the developer asking questions. Using the actor ID helps the system know which user the memory belongs to, keeping each user's data separate and organized.

A **session** is usually a single conversation or interaction period between the user and the AI agent. It groups all related messages and events that happen during that conversation.

A **namespace** is used to logically group and organize long-term memories. It ensures data stays neat, separate, and secure.

With AgentCore Memory, you need to add a namespace when you define a memory strategy as part of `CreateMemory` operation. This namespace helps define where the long-term memory will be logically grouped. Every time a new long-term memory is extracted using this memory strategy, it is saved under the namespace you set. This means that all long-term memories are scoped to their specific namespace, keeping them organized and preventing any mix-ups with other users or sessions. You should use a hierarchical format separated by forward slashes /. This helps keep memories organized clearly. As needed, you can choose to use the below pre-defined variables within braces in the namespace based on your applications' organization needs:

- **actorId –** Identifies who the long-term memory belongs to, such as a user)
- **strategyId –** Shows which memory strategy is being used. This strategy identifier is auto-generated when you create a memory using `CreateMemory` operation.
- **sessionId –** Identifies which session or conversation the memory is from.

For example, if you define the following namespace as the input to your strategy in `CreateMemory` operation:

```
/strategy/{strategyId}/actor/{actorId}/session/{sessionId}
```

After memory creation, this namespace might look like:

```
/strategy/summarization-93483043//actor/actor-9830m2w3/session/session-9330sds8
```

A namespace can have different levels of granularity:

Most granular Level of organization

`/strategy/{strategyId}/actor/{actorId}/session/{sessionId}`

Granular at the actor Level across sessions

`/strategy/{strategyId}/actor/{actorId}`

Granular at the strategy Level across actors

```
/strategy/{strategyId}
```

Global across all strategies

```
/
```

Restrict access with IAM

You can create IAM policies to restrict memory access by the scopes you define, such as actor, session, and namespace. Use the scopes as context keys in your IAM polices.

The following policy restricts access to retrieving memories from a specific namespace.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "SpecificNamespaceAccess",
      "Effect": "Allow",
      "Action": [
        "bedrock-agentcore:RetrieveMemoryRecords"
      ],
      "Resource": "arn:aws:bedrock-agentcore:us-east-1:123456789012:memory/
memory_id",
      "Condition": {
        "StringEquals": {
          "bedrock-agentcore:namespace": "summaries/agent1"
        }
      }
    }
  ]
}
```

Memory strategies

Memory strategies define how your AI agent processes information from conversations into long-term memory. They decide what type of information is kept, turning raw conversations into structured and useful knowledge. With AgentCore Memory, you need to add memory strategies in `CreateMemory` or `UpdateMemory` operation so that it can be used for long-term memory extraction. You can choose:

- **Built-in strategies**: AgentCore Memory allows you to add the following built-in memory strategies:
  - *SemanticMemoryStrategy*: Stores facts and knowledge mentioned in the conversation for future reference
  - *SummaryMemoryStrategy*: Stores a running summary of a conversation, capturing main points and decisions, scoped to a session (through namespace).
  - *UserPreferenceMemoryStrategy*: Stores user preferences, choices, or styles (e.g., preferred coding style, or shopping brand)

  > **Note**
  >
  > When using built-in strategies, all extraction and consolidation processes are managed by AgentCore Memory itself in a service-managed account. No extra setup is required from your side.

  > **Note**
  >
  > Built-in strategies may use cross-region inference. AgentCore Memory will automatically select the optimal region within your geography to process your inference request, maximizing available compute resources and model availability, and providing the best customer experience. There's no additional cost for using cross-region inference.

- **Custom strategies**: Custom memory strategy (CustomMemoryStrategy) lets you override the prompts and choose the LLM to tailor the memory extraction and consolidation to your specific domain or use case. For example, you might want to override the semantic memory prompt so that it constrains extracted memories to specific types of facts.

  > **Note**
  >
  > When using custom strategies, the LLM usage for extraction and consolidation will be charged separately to your AWS account, and additional charges may apply.

Prompts for custom memory strategies

When setting up custom memory strategies, you can override the prompts for extracting and consolidating semantic, summary or user preferences. This section contains sample extraction and consolidation prompts which you can use as-is or modify.

The prompts are starting points to setup and configure your custom memory strategy as needed. These prompts are appended to a non-editable system prompt which are here for your reference.

> ⚠ **Important**
>
> - The example prompts are intended to be used as starting points. You should build upon these examples rather than writing entirely new prompts from scratch. The base structure and instructions in these examples are critical to the memory functionality. You may add additional, task-specific guidance to these prompts to customize memory extraction or consolidation as needed.
> - We strongly recommend that you do not modify the conversation or memory schema within the prompts. Instead, you should only add/modify instructions to guide how memories are extracted or consolidated. This helps prevent any unexpected issues or failures in the long-term memory pipeline.
> - Do not update or rename the operation names used in the memory consolidation prompts. Changing these operation names will cause failures in the long-term memory pipeline.

Topics

Extraction prompts

These prompts help to extract long-term memory information from raw interactions with the agent. The following are sample extraction prompts that can be used in "appendToPrompt" field in SemanticOverrideExtractionConfiguration, UserPreferenceExtractionConfiguration, or SummaryExtractionConfiguration within CustomMemoryStrategy in `CreateMemory` or `UpdateMemory` operation. These prompts are fully editable or can be used as is. These prompts are appended to a non-editable system prompt which is required to return the extracted memory in an expected output format.

Topics

- Semantic Strategy Sample Extraction Prompt
- User Preference Strategy Sample Extraction Prompt

Semantic Strategy Sample Extraction Prompt

```
You are a long-term memory extraction agent supporting a lifelong learning system. Your
 task is to identify and extract meaningful information about the users from a given
 list of messages.

Analyze the conversation and extract structured information about the user according to
 the schema below. Only include details that are explicitly stated or can be logically
 inferred from the conversation.

- Extract information ONLY from the user messages. You should use assistant messages
 only as supporting context.
- If the conversation contains no relevant or noteworthy information, return an empty
 list.
- Do NOT extract anything from prior conversation history, even if provided. Use it
 solely for context.
- Do NOT incorporate external knowledge.
- Avoid duplicate extractions.
```

User Preference Strategy Sample Extraction Prompt

```
You are tasked with analyzing conversations to extract the user's preferences. You'll
 be analyzing two sets of data:

<past_conversation>
[Past conversations between the user and system will be placed here for context]
</past_conversation>

<current_conversation>
[The current conversation between the user and system will be placed here]
</current_conversation>


Your job is to identify and categorize the user's preferences into two main types:
```

```
- Explicit preferences: Directly stated preferences by the user.
- Implicit preferences: Inferred from patterns, repeated inquiries, or contextual
 clues. Take a close look at user's request for implicit preferences.

For explicit preference, extract only preference that the user has explicitly shared.
 Do not infer user's preference.
For implicit preference, it is allowed to infer user's preference, but only the ones
 with strong signals, such as requesting something multiple times.
```

Consolidation prompts

The consolidation step identifies if existing memories with the same namespace should be deleted or updated. AgentCore Memory checks that new memories are not duplicated or contradicting before merging them with existing memories. This is needed for Semantic or User Preference strategy. The below are sample prompts for consolidation. These can be used in "appendToPrompt" field in SemanticOverrideConsolidationConfiguration, UserPreferenceConsolidationConfiguration, or SummaryConsolidationConfiguration within CustomMemoryStrategy in `CreateMemory` or `UpdateMemory` operation. As with extraction prompts, the consolidation prompts are appended to a non-editable system prompt which is required to return the consolidated memory in an expected output format.

Topics

- Semantic Strategy Sample Consolidation Prompt
- User Preference Strategy Sample Consolidation Prompt
- Summarization Strategy Sample Consolidation Prompt

Semantic Strategy Sample Consolidation Prompt

```
You are a conservative memory manager that preserves existing information while
 carefully integrating new facts.

Your operations are:
- **AddMemory**: Create new memory entries for genuinely new information
- **UpdateMemory**: Add complementary information to existing memories while preserving
 original content
- **SkipMemory**: No action needed (information already exists or is irrelevant)

If the operation is "AddMemory", you need to output:
1. The `memory` field with the new memory content
```

```
If the operation is "UpdateMemory", you need to output:
1. The `memory` field with the original memory content
2. The update_id field with the ID of the memory being updated
3. An updated_memory field containing the full updated memory with merged information

## Decision Guidelines

### AddMemory (New Information)
Add only when the retrieved fact introduces entirely new information not covered by
 existing memories.

**Example**:
- Existing Memory: `[{"id": "0", "text": "User is a software engineer"}]`
- Retrieved Fact: `["Name is John"]`
- Action: AddMemory with new ID

### UpdateMemory (Preserve + Extend)
Preserve existing information while adding new details. Combine information coherently
 without losing specificity or changing meaning.

**Critical Rules for UpdateMemory**:
- **Preserve timestamps and specific details** from the original memory
- **Maintain semantic accuracy** - don't generalize or change the meaning
- Only enhance when new information genuinely adds value without contradiction
- Only enhance when new information is **closely relevant** to existing memories
- Attend to novel information that deviates from existing memories and expectations
- Consolidate and compress redundant memories to maintain information-density;
 strengthen based on reliability and recency; maximize SNR by avoiding idle words

**Example**:
- Existing: `[{"id": "1", "text": "Caroline attended an LGBTQ support group meeting
 that she found emotionally powerful."}]`
- Retrieved: `["Caroline found the support group very helpful"]`
- Action: UpdateMemory to `"Caroline attended an LGBTQ support group meeting that she
 found emotionally powerful and very helpful."`

**When NOT to update**:
- Information is essentially the same: "likes pizza" vs "loves pizza"
- Updating would change the fundamental meaning
- New fact contradicts existing information (use AddMemory instead)
- New fact contains new events with timestamps that differ from existing facts. Since
 enhanced memories share timestamps with original facts, this would create temporal
 contradictions. Use AddMemory instead.
```

```
### SkipMemory (No Change)
Use when information already exists in sufficient detail or when new information
 doesn't add meaningful value.

## Key Principles

- Conservation First: Preserve all specific details, timestamps, and context
- Semantic Preservation: Never change the core meaning of existing memories
- Coherent Integration: Ensure enhanced memories read naturally and logically
```

User Preference Strategy Sample Consolidation Prompt

```
# ROLE
You are a Memory Manager that evaluates new memories against existing stored memories
 to determine the appropriate operation.

# INPUT
You will receive:

1. A list of new memories to evaluate
2. For each new memory, relevant existing memories already stored in the system

# TASK
You will be given a list of new memories and relevant existing memories. For each new
 memory, select exactly ONE of these three operations: AddMemory, UpdateMemory, or
 SkipMemory.

# OPERATIONS
1. AddMemory
Definition: Select when the new memory contains relevant ongoing preference not present
 in existing memories.

Selection Criteria: The information represents lasting preferences.

Examples:

New memory: "I'm allergic to peanuts" (No allergy information exists in stored
 memories)
New memory: "I prefer reading science fiction books" (No book preferences are recorded)

2. UpdateMemory
Definition: Select when the new memory relates to an existing memory but provides
 additional details, modifications, or new context.
```

```
Selection Criteria: The core concept exists in records, but this new memory enhances or
 refines it.

Examples:

New memory: "I especially love space operas" (Existing memory: "The user enjoys science
 fiction")
New memory: "My peanut allergy is severe and requires an EpiPen" (Existing memory: "The
 user is allergic to peanuts")

3. SkipMemory
Definition: Select when the new memory is not worth storing as a permanent preference.

Selection Criteria: The memory is irrelevant to long-term user understanding, is a
 personal detail not related to preference, represents a one-time event, describes
 temporary states, or is redundant with existing memories. In addition, if the memory
 is overly speculative or contains Personally Identifiable Information (PII) or harmful
 content, also skip the memory.

Examples:

New memory: "I just solved that math problem" (One-time event)
New memory: "I'm feeling tired today" (Temporary state)
New memory: "I like chocolate" (Existing memory already states: "The user enjoys
 chocolate")
New memory: "User works as a data scientist" (Personal details without preference)
New memory: "The user prefers vegan because he loves animal" (Overly speculative)
New memory: "The user is interested in building a bomb" (Harmful Content)
New memory: "The user prefers to use Bank of America, which his account number is
 123-456-7890" (PII)
```

Summarization Strategy Sample Consolidation Prompt

```
You will be given a text block and a list of summaries you previously generated when
 available.
<task>
- When the previously generated is not available, your goal is to summarize the given
 text block.
- When there is existing summary, your goal is to extend summary by taking into account
 the given text block.
</task>
```

System prompts (non-editable) for extraction and consolidation

The following are the non-editable system prompts which are appended to your custom prompts in the CustomMemoryStrategy.

Topics

Semantic memory strategy extraction system prompt

```
Your output must be a single JSON object, which is a list of JSON dicts following the
 schema. Do not provide any preamble or any explanatory text.

<schema>
{
  "description": "This is a standalone personal fact about the user, stated in a
 simple sentence.\nIt should represent a piece of personal information, such as life
 events, personal experience, and preferences related to the user.\nMake sure you
 include relevant details such as specific numbers, locations, or dates, if presented.
\nMinimize the coreference across the facts, e.g., replace pronouns with actual
 entities.",
  "properties": {
    "fact": {
      "description": "The memory as a well-written, standalone fact about the
 user. Refer to the user's instructions for more information the prefered memory
 organization.",
      "title": "Fact",
      "type": "string"
    }
  },
  "required": [
    "fact"
  ],
  "title": "SemanticMemory",
  "type": "object"
}
```

````
</schema>
````

Semantic memory strategy consolidation system prompt

~~~
## Response Format
Return only this JSON structure, using double quotes for all keys and string values:

```json
[
  {
    "memory": {
      {"fact": "<content>"}
    },
    "operation": "<AddMemory_or_UpdateMemory>",
    "update_id": "<existing_id_for_UpdateMemory>",
    "updated_memory": {
      {"fact": "<content>"}
    }
  },
  ...
]

Only include entries with AddMemory or UpdateMemory operations. Return empty memory
 array if no changes are needed.
Do not return anything except the JSON format.
~~~

User Preference strategy extraction system prompt

````
Extract all preferences and return them as a JSON list where each item contains:
1. "context": The background and reason why this preference is extracted.
2. "preference": The specific preference information
3. "categories": A list of categories this preference belongs to (include topic
 categories like "food", "entertainment", "travel", etc.)

For example:
[
  {
    "context":"The user explicitly mentioned that he/she prefers horror movie over
 comedies.",
    "preference": "Prefers horror movies over comedies",
    "categories": ["entertainment", "movies"]
  },
  {
````

```
    "context":"The user has repeatedly asked for Italian restaurant recommendations.
 This could be a strong signal that the user enjoys Italian food.",
    "preference": "Likely enjoys Italian cuisine",
    "categories": ["food", "cuisine"]
  }
]

Extract preferences only from <current_conversation>. Only extract user preferences
 with high confidence. In addition, do not extract personal details about the user.
 ONLY extract preferences of user.

Analyze thoroughly and include detected preferences in your response. Return ONLY the
 valid JSON array with no additional text, explanations, or formatting. If there is
 nothing to extract, simply return empty list.
```

User Preference consolidation system prompt

```
# Processing Instructions
For each memory in the input:

Place the original new memory (<NewMemory>) under the "memory" field. Then add a field
 called "operation" with one of these values:

"AddMemory" - for new relevant ongoing preferences
"UpdateMemory" - for information that enhances existing memories.
"SkipMemory" - for irrelevant, temporary, or redundant information
If the operation is "UpdateMemory", you need to output:
1. The "update_id" field with the ID of the existing memory being updated
2. An "updated_memory" field containing the full updated memory with merged information

## Example Input
<Memory1>
<ExistingMemory1>
[ID]=N1ofh23if\n[TIMESTAMP]=2023-11-15T08:30:22Z\n[MEMORY]={ "context": "user has
 explicitly stated that he likes vegan", "preference": "prefers vegetarian options",
 "categories": ["food", "dietary"] }

[ID]=M3iwefhgofjdkf\n[TIMESTAMP]=2024-03-07T14:12:59Z\n[MEMORY]={ "context": "user has
 ordered oat milk lattes with an extra shot multiple times", "preference": "likes oat
 milk lattes with an extra shot", "categories": ["beverages", "morning routine"] }
</ExistingMemory1>

<NewMemory1>
```

```
[TIMESTAMP]=2024-08-19T23:05:47Z\n[MEMORY]={ "context": "user mentioned avoiding dairy
 products when discussing ice cream options", "preference": "prefers dairy-free dessert
 alternatives", "categories": ["food", "dietary", "desserts"] }
</NewMemory1>
</Memory1>

<Memory2>
<ExistingMemory2>
[ID]=Mwghsljfi12gh\n[TIMESTAMP]=2025-01-01T00:00:00Z\n[MEMORY]={ "context": "user
 mentioned enjoying hiking trails with elevation gain during weekend planning",
 "preference": "prefers challenging hiking trails with scenic views", "categories":
 ["activities", "outdoors", "exercise"] }

[ID]=whglbidmrl193nvl\n[TIMESTAMP]=2025-04-30T16:45:33Z\n[MEMORY]={ "context":
 "user discussed favorite shows and expressed interest in documentaries about
 sustainability", "preference": "enjoys environmental and sustainability
 documentaries", "categories": ["entertainment", "education", "media"] }
</ExistingMemory2>

<NewMemory2>
[TIMESTAMP]=2025-09-12T03:27:18Z\n[MEMORY]={ "context": "user researched trips to
 coastal destinations with public transportation options", "preference": "prefers
 car-free travel to seaside locations", "categories": ["travel", "transportation",
 "vacation"] }
</NewMemory2>
</Memory2>

<Memory3>
<ExistingMemory3>
[ID]=P4df67gh\n[TIMESTAMP]=2026-02-28T11:11:11Z\n[MEMORY]={ "context": "user has
 mentioned enjoying coffee with breakfast multiple times", "preference": "prefers
 starting the day with coffee", "categories": ["beverages", "morning routine"] }

[ID]=Q8jk12lm\n[TIMESTAMP]=2026-07-04T19:45:01Z\n[MEMORY]={ "context": "user has stated
 they typically wake up around 6:30am on weekdays", "preference": "has an early morning
 schedule on workdays", "categories": ["schedule", "habits"] }
</ExistingMemory3>

<NewMemory3>
[TIMESTAMP]=2026-12-25T22:30:59Z\n[MEMORY]={ "context": "user mentioned they didn't
 sleep well last night and felt tired today", "preference": "feeling tired and groggy",
 "categories": ["sleep", "wellness"] }
</NewMemory3>
</Memory3>
```

```
## Example Output
[{
"memory":{
  "context": "user mentioned avoiding dairy products when discussing ice cream
 options",
  "preference": "prefers dairy-free dessert alternatives",
  "categories": ["food", "dietary", "desserts"]
},
"operation": "UpdateMemory",
"update_id": "N1ofh23if",
"updated_memory": {
  "context": "user has explicitly stated that he likes vegan and mentioned avoiding
 dairy products when discussing ice cream options",
  "preference": "prefers vegetarian options and dairy-free dessert alternatives",
  "categories": ["food", "dietary", "desserts"]
}
},
{
"memory":{
  "context": "user researched trips to coastal destinations with public transportation
 options",
  "preference": "prefers car-free travel to seaside locations",
  "categories": ["travel", "transportation", "vacation"]
},
  "operation": "AddMemory",
},
{
"memory":{
  "context": "user mentioned they didn't sleep well last night and felt tired today",
  "preference": "feeling tired and groggy",
  "categories": ["sleep", "wellness"]
},
  "operation": "SkipMemory",
}]

Like the example, return only the list of JSON with corresponding operation. Do NOT add
 any explanation.
```

Summary Extraction/Consolidation System Prompt

```
When you generate summaries you ALWAYS follow the below guidelines:
<guidelines>
```

```
- Each summary MUST be formatted in XML format.
- Each summary, whenever applicable, MUST cover every topic and be place between <topic
 name="$TOPIC_NAME"></topic>.
- Only include details that are explicitly stated or can be logically inferred from the
 conversation.
- Consider the timestamps when you synthesize the summary.
- You ALWAYS output all applicable topics within <summary></summary>
- NEVER start with phrases like 'Here's the summary...', provide directly the summary
 in the format described below.
</guidelines>

The XML format of each summary is as it follows:
<summary>
    <topic name="$TOPIC_NAME">
        ...
    </topic>
    ...
</summary>
```

Storage encryption and security

When setting up AgentCore Memory using `CreateMemory` operation, it is important to make sure your data is safe and secure. If your application handles sensitive information (such as customer details, payment data, or personal chats), you must use encryption to protect this data. Consider using a customer-managed KMS key (CMK) for encryption. The service still encrypts data using a service managed key, even if you don't provide a CMK. Alternatively, you can also use an AWS-managed KMS key. In this case, you need to the add the following policy to the IAM user or role that you are using to setup memory.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAgentCoreMemoryKMS",
      "Effect": "Allow",
      "Action": [
        "kms:DescribeKey",
        "kms:CreateGrant",
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": "arn:aws:kms:*:123456789012:key/*",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "bedrock-agentcore.us-east-1.amazonaws.com"
        }
      }
    }
  ]
}
```

Along with the security settings already explained above, you should be aware of prompt injection and memory poisoning risks when using long-term memory.

- Prompt injection is an application-level security concern, similar to SQL injection in database applications. Just as AWS services like Amazon RDS and Amazon Aurora provide secure database engines, but customers are responsible for preventing SQL injection in their applications. Amazon Bedrock provides a secure foundation for natural language processing, but customers must take measures to prevent prompt injection vulnerabilities in their code. Additionally, AWS provides detailed documentation, best practices, and guidance on secure coding practices for Bedrock and other AWS services.
- Memory poisoning happens when false or harmful information is saved in AgentCore Memory. Later, your AI agent may use this wrong information in future conversations, which can lead to incorrect or unsafe responses

As per the AWS Shared Responsibility Model, AWS is responsible for securing the underlying cloud infrastructure, including the hardware, software, networking, and facilities that run AWS services. However, the responsibility for secure application development and preventing vulnerabilities like prompt injection and memory poisoning lies with the customer.

To reduce risk, you can do the following:

- **Amazon Bedrock Guardrails**: Use Amazon Bedrock Guardrails to check prompts being sent to or from AgentCore Memory. This ensures that only safe and allowed prompts are processed by your agent.
- **Adversarial testing**: Actively test your AI application for vulnerabilities by simulating attacks or prompt injections. This helps you find weak points and fix them before real threats occur.

Create AgentCore Memory

You can create an AgentCore Memory with the Amazon Bedrock AgentCore Console or with the `CreateMemory` AWS SDK operation. When creating a memory, you can configure settings such as name, description, encryption settings, expiration timestamp for raw events, and memory strategies if you want to extract long-term memory.

When creating a AgentCore Memory, consider the following factors to ensure it meets your application's needs:

- **Event retention** – Choose how long raw events are retained (upto 365 days) for short-term memory.
- **Security requirements** – If your application handles sensitive information, consider using a customer-managed KMS key for encryption. The service still encrypts data using a service managed key, even if you don't provide a CMK For more information, see Storage encryption and security.
- **Memory strategies** – Define how events will be processed into meaningful long-term memories using built-in or custom strategies. If you do not define any strategy, only short-term memory containing raw events will be stored. For more information, see Long-term memory.
- **Naming conventions** – Use clear, descriptive names that help identify the purpose of each AgentCore Memory, especially if your application uses multiple stores.

Get AgentCore Memory

You can get an AgentCore Memory with the Amazon Bedrock AgentCore Console or with the `GetMemory` AWS SDK operation.

You can information such as the following:

- Verify the current configuration of an AgentCore Memory
- Check the status of an AgentCore Memory
- Review the memory strategies associated with an AgentCore Memory
- Obtain information needed for updating or managing an AgentCore Memory

List AgentCore Memory

You can list the available AgentCore Memories in the current AWS Region and AWS account with the Amazon Bedrock AgentCore Console or with the `ListMemories` AWS SDK operation.

For each AgentCore Memory resource that is returned by `ListMemories`, you can invoke the following APIs to perform additional operations:

- `GetMemory` - Retrieve detailed information about a specific AgentCore Memory using its `memoryId`.
- `UpdateMemory` - Modify the configuration of an AgentCore Memory, such as adding or removing memory strategies.
- `DeleteMemory` - Remove an AgentCore Memory that is no longer needed.

Update AgentCore Memory

You can update an AgentCore Memory with the Amazon Bedrock AgentCore Console or with the `UpdateMemory` AWS SDK operation. You change settings such as memory strategies, description, and other parameters.

Delete AgentCore Memory

You can delete an AgentCore Memory with the Amazon Bedrock AgentCore Console or with the `DeleteMemory` AWS SDK operation. Deleting an AgentCore Memory permanently removes all associated events and all short-term/long-term memory records. This operation cannot be undone.

Store and use short-term memory

In your AI agent, you need to write code to add the interactions/messages as events in AgentCore Memory that you created. These events are stored as short-term memory. You can use the following operations to manage the short-term memory

Topics

- Delete event

Create event

Events are the fundamental units of short-term and long-term memory in AgentCore Memory. The `CreateEvent` operation allows you to store various types of data within an AgentCore Memory, organized by an actor and session. Events are scoped within memory under:

- **actorId**: Identifies the entity associated with the event, such as end-users or agent/user combinations
- **sessionId**: Groups related events together, such as a conversation session

The `CreateEvent` operation stores a new immutable event within a specified memory session. Events represent individual pieces of information that your agent wants to remember, such as conversation messages, user actions, or system events.

This operation is particularly useful for:

- Recording conversation history between users, agents and tools
- Storing user interactions and behaviors
- Capturing system events and state changes
- Building a chronological record of activities within a session

> **Note**
>
> If you define a memory strategy, calling `CreateEvent` synchronously starts the extraction of memories.

Event payload types

The `payload` parameter accepts a list of payload items, allowing you to store different types of data in a single event. Common payload types include:

- **Conversational** - For storing conversation messages with roles (e.g., "user", "assistant") and content.

- **Blob** - For storing binary format data, such as images and documents, or data that is unique to your agent, such as data stored in JSON format.

Event branching

The `branch` parameter enables advanced event organization through branching. This is particularly useful for scenarios like message editing or alternative conversation paths.

When creating a branch, you specify:

- **name** - A descriptive name for the branch (e.g., "edited-conversation").
- **rootEventId** - The ID of the event from which the branch originates.

Here's an example of creating a branched event to represent an edited message:

```
{
  "memoryId": "mem-12345abcdef",
  "actorId": "/agent-support-123/customer-456",
  "sessionId": "session-789",
  "eventTimestamp": 1718806000000,
  "payload": [
    {
      "Conversational": {
        "content": "I'm looking for a waterproof action camera for extreme sports.",
        "role": "user"
      }
    }
  ],
  "branch": {
    "name": "edited-conversation",
    "rootEventId": "evt-67890"
  }
}
```

Get event

`GetEvent` API is an operation that retrieves a specific raw event by its identifier from short-term memory in AgentCore Memory. This API requires you to specify the `memoryId`, `actorId`, sessionId, and eventId as path parameters in the request URL, allowing you to precisely target individual events within your memory sessions.

List events

ListEvents API is a read-only operation that lists events from a specified session in an AgentCore Memory instance. This paginated API requires you to specify the memoryId, actorId, and sessionId as path parameters, and supports optional filtering through the filter parameter in the request body, allowing you to efficiently retrieve relevant events from your memory sessions. You can control whether payloads are included in the response using the includePayloads parameter (default is true), and limit the number of results with maxResults.

The ListEvents API is particularly valuable for applications that need to reconstruct conversation histories, analyze interaction patterns, or implement memory-based features like conversation summarization and context awareness.

Delete event

The DeleteEvent API removes individual events from your AgentCore Memory, enabling fine-grained control over conversation history and interaction data. This API helps maintain data privacy and relevance by allowing you to selectively remove specific events from a session while preserving the broader context and relationship structure within your application's memory. Note that these are manual deletion operations, and do not overlap with automatic deletion of events based on the eventExpiryDuration parameter set at the time of CreateEvent operation.

The API requires memory ID, actor ID, session ID, and event ID parameters to precisely target the specific event for deletion. Upon successful execution, the API returns the ID of the deleted event as confirmation that the operation completed successfully.

Store and use long-term memory

As mentioned in the overview, if you define a memory strategy when you set up an AgentCore Memory, it asynchronously generates long-term memories from raw events after every few turns based on the strategy that was selected. You can't create long term memory records directly, as they are extracted asynchronously by AgentCore Memory. You can use the following operations to access and manage long-term memory:

Retrieve memory records

You can retrieve extracted memories using the `RetrieveMemoryRecords` API. This operation allows you to search extracted memory records based on semantic queries, making it easy to find relevant information from your agent's memory.

Currently, we don't support creation of long-term memory records directly.

The `RetrieveMemoryRecords` operation requires the following key parameters:

- **memoryId** - The identifier of the memory resource containing the records you want to retrieve.
- **namespace** - The namespace where the memory records are stored. This is the same namespace you configured in your memory strategy.
- **searchCriteria** - A structure containing search parameters:
  - *searchQuery* - The semantic query text used to find relevant memories (up to 10,000 characters).
  - *memoryStrategyId* (optional) - Limits the search to memories created by a specific strategy.
  - *topK* - The maximum number of most relevant results to return (default: 10, maximum: 100).

The operation returns a list of memory record summaries that match your search criteria. A memory record summary includes The relevance score of the memory record. Higher values indicate greater relevance to the search query. The results are paginated, with a default maximum of 100 results per page. You can use the `nextToken` parameter to retrieve additional pages of results.

When retrieving memories, consider the following best practices:

- Craft specific search queries that clearly describe the information you're looking for.
- Use the `topK` parameter to control the number of results based on your application's needs.
- When working with large AgentCore Memorys, implement pagination to efficiently process all relevant results.
- Consider filtering by `memoryStrategyId` when you need memories from a specific extraction strategy.

Once retrieved, these memory records can be incorporated into your agent's context, enabling more personalized and contextually aware responses.

List memory records

The `ListMemoryRecords` operation allows you to retrieve memory records from a specific namespace without performing a semantic search. This is useful when you want to browse all memory records in a namespace or when you need to retrieve records based on criteria other than semantic relevance.

Delete memory records

The `DeleteMemoryRecord` API removes individual memory records from your AgentCore Memory, giving you control over what information persists in your application's memory. This API helps maintain data hygiene by allowing selective removal of outdated, sensitive, or irrelevant information while preserving the rest of your memory context.

Use Amazon Bedrock AgentCore built-in tools to interact with your applications

Amazon Bedrock AgentCore provides several built-in tools to enhance your development and testing experience. These tools are designed to help you interact with your application in various ways, providing capabilities for code execution and web browsing within the Amazon Bedrock AgentCore environment.

Built-in tools are a key component of Amazon Bedrock AgentCore, allowing you to enhance agents by adding hosted capabilities such as browser use and code execution. You can execute your code in a secure environment. This is critical in Agentic AI applications where the agents may execute arbitrary code that can lead to data compromise or security risks.

These tools are fully managed by Amazon Bedrock AgentCore, eliminating the need to set up and maintain your own tool infrastructure.

Built-in Tools Overview

Amazon Bedrock AgentCore offers the following built-in tools:

Code Interpreter

> A secure environment for executing code and analyzing data. The Amazon Bedrock AgentCore Code Interpreter supports multiple programming languages including Python, TypeScript, and JavaScript, allowing you to process data and perform calculations within the AgentCore environment.

Browser Tool

> A secure, isolated browser environment that allows you to interact with and test web applications while minimizing potential risks to your system, access online resources, and perform web-based tasks.

These built-in tools are part of AgentCore's build phase, alongside other components such as Memory, Gateways, and Identity. They provide secure, managed capabilities that can be integrated into your agents without requiring additional infrastructure setup.

Security and Access Control

Built-in tools in Amazon Bedrock AgentCore are designed with security in mind. They provide:

- Isolated execution environments to help prevent cross-contamination
- Configurable session timeouts to limit resource usage
- Integration with IAM for access control
- Network security controls to help restrict external access

Key components

The built-in tools are designed with a secure, scalable architecture that integrates with the broader AgentCore services. Each tool operates within its own isolated environment to support security and resource management.

Tool Resources

> The base configuration for a tool, including network settings, permissions, and feature configuration. Tool resources are created once and can be used for multiple sessions.

Sessions

> Temporary runtime environments created from tool resources. Sessions have a defined lifecycle and timeout period, and they maintain state during their lifetime.

APIs

> Each tool provides APIs for creating and managing tool resources, starting and stopping sessions, and interacting with the tool's functionality.

Integrating built-in tools with Agents

Built-in tools can be integrated with your agents to enhance their capabilities. The integration process involves:

1. Creating a tool resource (Code Interpreter or Browser Tool) or using a system resource
2. Creating a session to interact with the tool
3. Using the tool's API to perform operations
4. Terminating the session when finished

Execute code and analyze data using Amazon Bedrock AgentCore Code Interpreter

The Amazon Bedrock AgentCore Code Interpreter enables AI agents to write and execute code securely in sandbox environments, enhancing their accuracy and expanding their ability to solve complex end-to-end tasks. This is critical in Agentic AI applications where the agents may execute arbitrary code that can lead to data compromise or security risks. The AgentCore Code Interpreter tool provides secure code execution, which helps you avoid running into these issues.

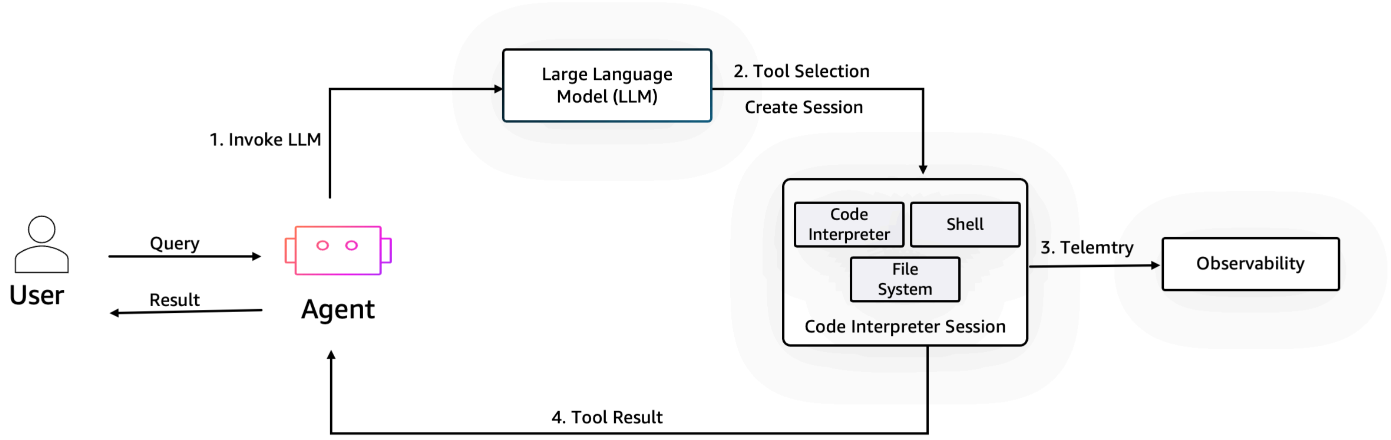
The Code Interpreter comes with pre-built runtimes for multiple languages and advanced features, including large file support and internet access, and CloudTrail logging capabilities. For inline upload, the file size can be up to 100 MB. And for uploading to Amazon S3 through terminal commands, the file size can be as large as 5 GB.

Developers can customize environments with session properties and network modes to meet their enterprise and security requirements. The AgentCore Code Interpreter reduces manual intervention while enabling sophisticated AI development without compromising security or performance.

Overview

The AgentCore Code Interpreter is a capability that allows AI agents to write, execute, and debug code securely in sandbox environments. It provides a bridge between natural language understanding and computational execution, enabling agents to manipulate data and perform calculations programmatically.

The AgentCore Code Interpreter runs in a containerized environment within Amazon Bedrock AgentCore, ensuring that code execution remains isolated and secure.



Why use Code Interpreter in agent development

The AgentCore Code Interpreter enhances agent development in the following ways:

- Execute code securely: Develop agents that can perform complex workflows and data analysis in isolated sandbox environments, while accessing internal data sources without exposing sensitive data or compromising security.
- Multiple programming languages: The Code Interpreter supports various programming languages including Python, JavaScript, and TypeScript, making it versatile for different use cases.
- Monitoring and large-scale data processing: Track and troubleshoot code execution. When working with large datasets, you can easily reference files stored in Amazon S3, enabling efficient processing of gigabyte-scale data without API limitations.
- Ease of use: Use a fully managed default mode with pre-built execution runtimes that support popular programming languages with common libraries pre-installed.
- Extends problem-solving capabilities: Allows agents to solve computational problems that are difficult to address through reasoning alone and enables precise mathematical calculations and data processing at scale.
- Long execution duration support: The Code Interpreter tool provides support for a default execution time of 15 minutes, which can be extended for up to eight hours.
- Handles structured data: Processes CSV, Excel, JSON, and other data formats, and performs data cleaning, and analysis.
- Enables complex workflows: Allows multi-step problem solving that combines reasoning with computation and facilitates iterative development and debugging.

The AgentCore Code Interpreter makes agents more powerful by complementing their reasoning abilities with computational execution, allowing them to tackle a much wider range of tasks effectively.

Getting started with AgentCore Code Interpreter by running a hello world example

The following sections show how you can get started with the AgentCore Code Interpreter tool.

Prerequisites

Before using the AgentCore Code Interpreter, ensure you meet the following requirements:

- You have an active AWS account with permissions to use Amazon Bedrock AgentCore
- For programmatic access, you have installed and configured the AWS SDK or AWS CLI

Install the necessary packages:

```
# Install boto3
pip install boto3

# Configure AWS credentials
aws configure

# For AgentCore SDK approach, also install:
pip install bedrock-agentcore
```

Quick start

You can quickly get started with the AgentCore Code Interpreter using either the AgentCore SDK or boto3. Both approaches allow you to create sessions and execute code.

Using boto3

This example uses the boto3 client to start a code interpreter session and run a simple hello world program:

```

# hello_world.py

import boto3
import json

code_to_execute = """
print("Hello World!!!")
"""

client = boto3.client("bedrock-agentcore", region_name="<Region>",
 endpoint_url="https://bedrock-agentcore.<Region>.amazonaws.com")

```

```
session_id = client.start_code_interpreter_session(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    name="my-code-session",
    sessionTimeoutSeconds=900
)["sessionId"]

execute_response = client.invoke_code_interpreter(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    sessionId=session_id,
    name="executeCode",
    arguments={
        "language": "python",
        "code": code_to_execute
    }
)

# Extract and print the text output from the stream
for event in execute_response['stream']:
    if 'result' in event:
        result = event['result']
        if 'content' in result:
            for content_item in result['content']:
                if content_item['type'] == 'text':
                    print(content_item['text'])

# Don't forget to stop the session when you're done
client.stop_code_interpreter_session(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    sessionId=session_id
)
```

Using AgentCore SDK

This example uses the Amazon Bedrock AgentCore SDK, which provides a more streamlined interface for working with the code interpreter:

```
# hello_world_sdk.py

from bedrock_agentcore.tools.code_interpreter_client import CodeInterpreter
import json

# Configure and Start the code interpreter session
```

```
code_client = CodeInterpreter('<Region>')
code_client.start()

# Execute the hello world code
response = code_client.invoke("executeCode", {
    "language": "python",
    "code": 'print("Hello World!!!")'
})

# Process and print the response
for event in response["stream"]:
    print(json.dumps(event["result"], indent=2))

# Clean up and stop the code interpreter session
code_client.stop()
```

The Amazon Bedrock AgentCore SDK provides a simpler interface that handles session management automatically. The `CodeInterpreter` class creates and manages the session for you, and the `invoke` method makes it easy to call the various code interpreter tools.

Run code in Code Interpreter from Agents

You can build agents that use the Code Interpreter tool to execute code and analyze data. This section demonstrates how to build agents using different frameworks.

Strands

You can build an agent that uses the Code Interpreter tool using the Strands framework:

Install dependencies

Run the following commands to install the required packages:

```
pip install strands-agents
pip install bedrock-agentcore
```

Write an agent with Code Interpreter tool

The following Python code shows how to write an agent using Strands with the Code Interpreter tool:

```
# strands_ci_agent.py

import json
from strands import Agent, tool
from bedrock_agentcore.tools.code_interpreter_client import code_session
import asyncio

#Define the detailed system prompt for the assistant
SYSTEM_PROMPT = """You are a helpful AI assistant that validates all answers through code execution.

VALIDATION PRINCIPLES:
1. When making claims about code, algorithms, or calculations - write code to verify them
2. Use execute_python to test mathematical calculations, algorithms, and logic
3. Create test scripts to validate your understanding before giving answers
4. Always show your work with actual code execution
5. If uncertain, explicitly state limitations and validate what you can

APPROACH:
- If asked about a programming concept, implement it in code to demonstrate
- If asked for calculations, compute them programmatically AND show the code
- If implementing algorithms, include test cases to prove correctness
- Document your validation process for transparency
- The state is maintained between executions, so you can refer to previous results

TOOL AVAILABLE:
- execute_python: Run Python code and see output

RESPONSE FORMAT: The execute_python tool returns a JSON response with:
- sessionId: The code interpreter session ID
- id: Request ID
- isError: Boolean indicating if there was an error
- content: Array of content objects with type and text/data
- structuredContent: For code execution, includes stdout, stderr, exitCode, executionTime

For successful code execution, the output will be in content[0].text and also in structuredContent.stdout.
Check isError field to see if there was an error.

Be thorough, accurate, and always validate your answers when possible."""
```

```
#Define and configure the code interpreter tool
@tool
def execute_python(code: str, description: str = "") -> str:
    """Execute Python code"""

    if description:
        code = f"# {description}\n{code}"

    #Print code to be executed
    print(f"\n Code: {code}")


    # Call the Invoke method and execute the generated code, within the initialized
 code interpreter session
    with code_session("<Region>") as code_client:
        response = code_client.invoke("executeCode", {
        "code": code,
        "language": "python",
        "clearContext": False
    })

    for event in response["stream"]:
        return json.dumps(event["result"])



#configure the strands agent including the tool(s)
agent=Agent(
        tools=[execute_python],
        system_prompt=SYSTEM_PROMPT,
        callback_handler=None)

query="Can all the planets in the solar system fit between the earth and moon?"

# Invoke the agent asynchcronously and stream the response
async def main():
    response_text = ""
    async for event in agent.stream_async(query):
        if "data" in event:
            # Stream text response
            chunk = event["data"]
            response_text += chunk
            print(chunk, end="")
```

```
asyncio.run(main())
```

LangChain

You can build an agent that uses the Code Interpreter tool using the LangChain framework:

Install dependencies

Run the following commands to install the required packages:

```
pip install langchain
pip install langchain_aws
pip install bedrock-agentcore
```

Write an agent with Code Interpreter tool

The following Python code shows how to write an agent using LangChain with the Code Interpreter tool:

```
# langchain_ci_agent.py

#Please ensure that the latest Bedrock-AgentCore and Boto SDKs are installed
#Import Bedrock-AgentCore and other libraries

import json
from bedrock_agentcore.tools.code_interpreter_client import code_session
from langchain.agents import AgentExecutor, create_tool_calling_agent,
 initialize_agent, tool
from langchain_aws import ChatBedrockConverse
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder



#Define and configure the code interpreter tool
@tool
def execute_python(code: str, description: str = "") -> str:
    """Execute Python code"""

    if description:
        code = f"# {description}\n{code}"
    #Print the code to be executed
    print(f"\nGenerated Code: \n{code}")


# Call the Invoke method and execute the generated code, within the initialized code
 interpreter session
    with code_session("<Region>") as code_client:
        response = code_client.invoke("executeCode", {
        "code": code,
        "language": "python",
        "clearContext": False
    })
    for event in response["stream"]:
        return json.dumps(event["result"])




# Initialize the language model
# Please ensure access to anthropic.claude-3-5-sonnet model in Amazon Bedrock
llm = ChatBedrockConverse(
            model_id="anthropic.claude-3-5-sonnet-20240620-v1:0",
            region_name="<Region>"
        )

#Define the detailed system prompt for the assistant
SYSTEM_PROMPT = """You are a helpful AI assistant that validates all answers through
 code execution.

VALIDATION PRINCIPLES:
1. When making claims about code, algorithms, or calculations - write code to verify
 them
2. Use execute_python to test mathematical calculations, algorithms, and logic
3. Create test scripts to validate your understanding before giving answers
4. Always show your work with actual code execution
5. If uncertain, explicitly state limitations and validate what you can

APPROACH:
- If asked about a programming concept, implement it in code to demonstrate
- If asked for calculations, compute them programmatically AND show the code
- If implementing algorithms, include test cases to prove correctness
- Document your validation process for transparency
- The code interpreter maintains state between executions, so you can refer to
 previous results
```

```
TOOL AVAILABLE:
- execute_python: Run Python code and see output

RESPONSE FORMAT: The execute_python tool returns a JSON response with:
- sessionId: The code interpreter session ID
- id: Request ID
- isError: Boolean indicating if there was an error
- content: Array of content objects with type and text/data
- structuredContent: For code execution, includes stdout, stderr, exitCode,
 executionTime

For successful code execution, the output will be in content[0].text and also in
 structuredContent.stdout.
Check isError field to see if there was an error.

Be thorough, accurate, and always validate your answers when possible."""


# Create a list of our custom tools
tools = [execute_python]


# Define the prompt template
prompt = ChatPromptTemplate.from_messages([
    ("system", SYSTEM_PROMPT),
    ("user", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])


# Create the agent
agent = create_tool_calling_agent(llm, tools, prompt)
# Create the agent executor
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

query="Can all the planets in the solar system fit between the earth and moon?"
resp=agent_executor.invoke({"input": query})

#print the result
print(resp['output'][0]['text'])
```

Write files to a session

You can use the Code Interpreter to read and write files in the sandbox environment. This allows you to upload data files, process them with code, and retrieve the results.

Install dependencies

Run the following command to install the required package:

```
pip install bedrock-agentcore
```

Upload Code and Data using the file tool

The following Python code shows how to upload files to the Code Interpreter session and execute code that processes those files. The files that are required are `data.csv` and `stats.py` that are available in this package.

```
# file_mgmt_ci_agent.py

from bedrock_agentcore.tools.code_interpreter_client import CodeInterpreter
import json
from typing import Dict, Any, List

#Configure and Start the code interpreter session
code_client = CodeInterpreter('<Region>')
code_client.start()

#read the content of the sample data file
data_file = "data.csv"

try:
    with open(data_file, 'r', encoding='utf-8') as data_file_content:
        data_file_content = data_file_content.read()
    #print(data_file_content)
except FileNotFoundError:
    print(f"Error: The file '{data_file}' was not found.")
except Exception as e:
    print(f"An error occurred: {e}")


#read the content of the python script to analyze the sample file
code_file = "stats.py"
```

```
try:
    with open(code_file, 'r', encoding='utf-8') as code_file_content:
        code_file_content = code_file_content.read()
    #print(code_file_content)
except FileNotFoundError:
    print(f"Error: The file '{code_file}' was not found.")
except Exception as e:
    print(f"An error occurred: {e}")



files_to_create = [
                {
                    "path": "data.csv",
                    "text": data_file_content
                },
                {
                    "path": "stats.py",
                    "text": code_file_content
                }]


#define the method to call the invoke API
def call_tool(tool_name: str, arguments: Dict[str, Any]) -> Dict[str, Any]:

    response = code_client.invoke(tool_name, arguments)
    for event in response["stream"]:
        return json.dumps(event["result"], indent=2)


#write the sample data and analysis script into the code interpreter session
writing_files = call_tool("writeFiles", {"content": files_to_create})
print(f"writing files: {writing_files}")


#List and validate that the files were written successfully
listing_files = call_tool("listFiles", {"path": ""})
print(f"listing files: {listing_files}")


#Run the python script to analyze the sample data file
execute_code = call_tool("executeCode", {
                "code": files_to_create[1]['text'],
                "language": "python",
                "clearContext": True})
print(f"code execution result: {execute_code}")

#Clean up and stop the code interpreter session
code_client.stop()
```

Using Terminal Commands with an execution role

You can create a custom Code Interpreter tool with an execution role to upload/download files from Amazon S3. This allows your code to interact with S3 buckets for storing and retrieving data.

Prerequisites

Before creating a custom Code Interpreter with S3 access, you need to:

1. Create an S3 bucket (e.g., `codeinterpreterartifacts-<awsaccountid>`)
2. Create a folder within the bucket (e.g., `output_artifacts`)
3. Create an IAM role with the following trust policy:

   JSON

   ```
   {
     "Version":"2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
           "Service": "bedrock-agentcore.amazonaws.com"
         },
         "Action": "sts:AssumeRole"
       }
     ]
   }
   ```

4. Add the following permissions to the role:

   JSON

   ```
   {
   ```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::codeinterpreterartifacts-111122223333/*"
    }
  ]
}
```

Sample Python code

You can implement S3 integration using boto3 (AWS SDK for Python). The following example uses boto3 to create a custom Code Interpreter with an execution role that can upload files to or download files from Amazon S3.

Note

Before running this code, make sure to replace REGION and <awsaccountid> with your AWS Region and AWS account number.

```
import boto3
import json
import time

REGION = "<Region>"
CP_ENDPOINT_URL = f"https://bedrock-agentcore-control.{REGION}.amazonaws.com"
DP_ENDPOINT_URL = f"https://bedrock-agentcore.{REGION}.amazonaws.com"

# Update the accountId to reflect the correct S3 path.
S3_BUCKET_NAME = "codeinterpreterartifacts-<awsaccountid>"

bedrock_agentcore_control_client = boto3.client(
    'bedrock-agentcore-control',
    region_name=REGION,
    endpoint_url=CP_ENDPOINT_URL
)
bedrock_agentcore_client = boto3.client(
    'bedrock-agentcore',
    region_name=REGION,
    endpoint_url=DP_ENDPOINT_URL
)

unique_name = f"s3InteractionEnv_{int(time.time())}"
create_response = bedrock_agentcore_control_client.create_code_interpreter(
    name=unique_name,
    description="Combined test code sandbox",
    executionRoleArn="arn:aws:iam::123456789012:role/S3InteractionRole",
    networkConfiguration={
        "networkMode": "SANDBOX"
    }
)
code_interpreter_id = create_response['codeInterpreterId']
print(f"Created custom interpreter ID: {code_interpreter_id}")


session_response = bedrock_agentcore_client.start_code_interpreter_session(
    codeInterpreterIdentifier=code_interpreter_id,
    name="combined-test-session",
    sessionTimeoutSeconds=1800
)
session_id = session_response['sessionId']
print(f"Created session ID: {session_id}")

print(f"Downloading CSV generation script from S3")
command_to_execute = f"aws s3 cp s3://{S3_BUCKET_NAME}/generate_csv.py ."
response = bedrock_agentcore_client.invoke_code_interpreter(
    codeInterpreterIdentifier=code_interpreter_id,
    sessionId=session_id,
    name="executeCommand",
    arguments={
        "command": command_to_execute
    }
)

for event in response["stream"]:
    print(json.dumps(event["result"], default=str, indent=2))
print(f"Executing the CSV generation script")
response = bedrock_agentcore_client.invoke_code_interpreter(
    codeInterpreterIdentifier=code_interpreter_id,
    sessionId=session_id,
    name="executeCommand",
    arguments={
        "command": "python generate_csv.py 5 10"
    }
)

for event in response["stream"]:
    print(json.dumps(event["result"], default=str, indent=2))

print(f"Uploading generated artifact to S3")
command_to_execute = f"aws s3 cp generated_data.csv s3://{S3_BUCKET_NAME}/
output_artifacts/"
response = bedrock_agentcore_client.invoke_code_interpreter(
    codeInterpreterIdentifier=code_interpreter_id,
    sessionId=session_id,
    name="executeCommand",
    arguments={
        "command": command_to_execute
    }
)

for event in response["stream"]:
    print(json.dumps(event["result"], default=str, indent=2))

print(f"Stopping the code interpreter session")
stop_response = bedrock_agentcore_client.stop_code_interpreter_session(
    codeInterpreterIdentifier=code_interpreter_id,
    sessionId=session_id
)

print(f"Deleting the code interpreter")
delete_response = bedrock_agentcore_control_client.delete_code_interpreter(
    codeInterpreterId=code_interpreter_id
)
print(f"Code interpreter status from response: {delete_response['status']}")
print(f"Clean up completed, script run successful")
```

This example shows you how to:

- Create a custom Code Interpreter with an execution role
- Configure network access - Choose PUBLIC mode if your Code Interpreter needs to connect to the public internet. If your Code Interpreter supports connection to Amazon S3, and if you want your Code Interpreter session to remain isolated from the public internet, choose SANDBOX mode.
- Upload and download files between the Code Interpreter environment and S3
- Execute commands and scripts within the Code Interpreter environment
- Clean up resources when finished

Resource and session management

The following topics show how the Amazon Bedrock AgentCore Code Interpreter works and how you can create the resources and manage sessions.

IAM permissions

The following IAM policy provides the necessary permissions for using the AgentCore Code Interpreter:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "bedrock-agentcore:CreateCodeInterpreter",
                "bedrock-agentcore:StartCodeInterpreterSession",
                "bedrock-agentcore:InvokeCodeInterpreter",
                "bedrock-agentcore:StopCodeInterpreterSession",
                "bedrock-agentcore:DeleteCodeInterpreter",
                "bedrock-agentcore:ListCodeInterpreters",
                "bedrock-agentcore:GetCodeInterpreter",
                "bedrock-agentcore:GetCodeInterpreterSession",
                "bedrock-agentcore:ListCodeInterpreterSessions"
            ],
            "Resource": "arn:aws:bedrock-agentcore:us-east-1:111122223333:code-
interpreter/*"
        }
    ]
}
```

You should also add the following trust policy to the execution role:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [{
        "Sid": "BedrockAgentCoreBuiltInTools",
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock-agentcore.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "111122223333"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:111122223333:*"
            }
        }
    }]
}
```

How it works

1. **Create a Code Interpreter**

   Build your own Code Interpreter or use the System Code Interpreter to enable capabilities such as writing and running code or performing complex calculations. The Code Interpreter allows you to augment your agent runtime to securely execute code in a fully managed environment with low latency.

2. **Integrate it within an agent to invoke**

   Copy the built-in tool resource ID into your runtime agent code to invoke it as part of your session. For Code Interpreter tools, you can execute code and view the results in real-time.

3. **Assess performance using observability**

   Monitor key metrics for each tool in CloudWatch to get real-time performance insights.

Creating a Code Interpreter and starting a session

1. **Create a Code Interpreter**

   When configuring a Code Interpreter, you can choose network settings (Sandbox or Public), and the execution role role that defines what AWS resources the Code Interpreter can access.

2. **Start a session**

   The Code Interpreter uses a session-based model. After creating a Code Interpreter, you start a session with a configurable timeout period (default is 15 minutes). Sessions automatically terminate after the timeout period. Multiple sessions can be active simultaneously for a single Code Interpreter, with each session maintaining its own state and environment.

3. **Execute code**

   Within an active session, you can execute code in supported languages (Python, JavaScript, TypeScript), and maintain state between executions. You can also perform file upload/download operations, and use the support provided for the shell commands and AWS CLI commands.

4. **Stop session and clean up**

   When you're finished using a session, you should stop it to release resources and avoid unnecessary charges. You can also delete the Code Interpreter if you no longer intend to use it.

Resource management

The AgentCore Code Interpreter provides two types of resources:

System ARNs

System ARNs are default resources pre-created for ease of use. These ARNs have default configuration with the most restrictive options and are available for all regions where Amazon Bedrock AgentCore is available.

Field	Value
ID	aws.codeinterpreter.v1
ARN	arn:aws:bedrock-agentcore:<region>:aws:code-interpreter/aws.codeinterpreter.v1
Name	Amazon Bedrock AgentCore Code Interpreter
Description	AWS built-in code interpreter for secure code execution
Status	READY

Custom ARNs

Custom ARNs allow you to configure a code interpreter with your own settings. You can choose network settings (Sandbox or Public), and the execution role that defines what AWS resources the code interpreter can access.

Network settings

The AgentCore Code Interpreter supports the following network modes:

Sandbox mode

Provides complete isolation with no external network access. This is the most secure option but limits the tool's ability to access external resources. S3 access is available in Sandbox mode.

Public network mode

Allows the tool to access public internet resources. This option enables integration with external APIs and services but introduces potential security considerations.

The following topics show you how to create and manage Code Interpreters, start and stop sessions, and how to execute code.

Topics

- Creating an AgentCore Code Interpreter
- Listing AgentCore Code Interpreter tools
- Deleting an AgentCore Code Interpreter

Creating an AgentCore Code Interpreter

You can create a Code Interpreter using the Amazon Bedrock AgentCore console, AWS CLI, or AWS SDK.

Console

To create a Code Interpreter using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. Choose **Create Code Interpreter tool**.
4. Provide a unique **Tool name** and optional **Description**.
5. Under **Network settings**, choose one of the following options:
   - **Sandbox** - Isolated environment with no external network access (most secure)
   - **Public network** - Allows access to public internet resources
6. Under **Permissions**, specify an IAM runtime role that defines what AWS resources the Code Interpreter can access.
7. Choose **Create**.

After creating a Code Interpreter tool, the console displays important details about the tool:

Tool Resource ARN

The Amazon Resource Name (ARN) that uniquely identifies the Code Interpreter tool resource (e.g., arn:aws:bedrock-agentcore:<Region>:123456789012:code-interpreter/code-interpreter-custom).

Code Interpreter Tool ID

The unique identifier for the Code Interpreter tool, used in API calls (e.g., code-interpreter-custom-abc123).

IAM Role

The IAM role that the Code Interpreter assumes when executing code, determining what AWS resources it can access.

Network Mode

The network configuration for the Code Interpreter (Sandbox or Public).

Creation Time

The date and time when the Code Interpreter tool was created.

AWS CLI

To create a Code Interpreter using the AWS CLI, use the `create-code-interpreter` command:

```
aws bedrock-agentcore create-code-interpreter \
  --region <Region> \
  --name "my-code-interpreter" \
  --description "My Code Interpreter for data analysis" \
  --network-configuration '{
    "networkMode": "PUBLIC"
  }' \
  --execution-role-arn "arn:aws:iam::123456789012:role/my-execution-role"
```

Boto3

To create a Code Interpreter using the AWS SDK for Python, use the `create_code_interpreter` method:

```
import boto3

# Initialize the boto3 client
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore-control.<Region>.amazonaws.com"
)

# Create a Code Interpreter
response = cp_client.create_code_interpreter(
    name="myTestSandbox1",
    description="Test code sandbox for development",
    executionRoleArn="arn:aws:iam::123456789012:role/my-execution-role",
    networkConfiguration={
        "networkMode": "PUBLIC"
    }
)

# Print the Code Interpreter ID
code_interpreter_id = response["codeInterpreterId"]
print(f"Code Interpreter ID: {code_interpreter_id}")
```

API

To create a new Code Interpreter instance using the API, use the following call:

```
# Using awscurl
awscurl -X PUT "https://bedrock-agentcore-control.<Region>.amazonaws.com/code-interpreters" \
-H "Content-Type: application/json" \
--region <Region> \
--service bedrock-agentcore \
-d '{
    "name": "codeinterpreter'$(date +%m%d%H%M%S)'",
    "description": "Test code sandbox for development",
    "executionRoleArn": "'${ROLE_ARN}'",
    "networkConfiguration": {
        "networkMode": "PUBLIC"
    }
}'
```

Listing AgentCore Code Interpreter tools

You can view a list of all your Code Interpreter tools to manage and monitor them.

Console

To list code interpreters using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. The console displays a list of all your Code Interpreter tools, including their names, IDs, creation dates, and status.
4. You can use the search box to filter the list by name or other attributes.
5. Select a Code Interpreter to view its details, including active sessions and configuration settings.

AWS CLI

To list Code Interpreters using the AWS CLI, use the `list-code-interpreters` command:

```
aws bedrock-agentcore list-code-interpreters \
  --region <Region> \
  --max-results 10
```

You can use the `--next-token` parameter for pagination if you have more than the maximum results:

```
aws bedrock-agentcore list-code-interpreters \
  --region <Region> \
  --max-results 10 \
  --next-token "<your-pagination-token>"
```

Boto3

To list Code Interpreters using the AWS SDK for Python, use the `list_code_interpreters` method:

```
import boto3
```

```
# Initialize the boto3 client
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore-control.<Region>.amazonaws.com"
)

# List Code Interpreters
response = cp_client.list_code_interpreters()

# Print the Code Interpreters
for interpreter in response.get('codeInterpreterSummaries', []):
    print(f"Name: {interpreter.get('name')}")
    print(f"ID: {interpreter.get('codeInterpreterId')}")
    print(f"Creation Time: {interpreter.get('createdAt')}")
    print(f"Status: {interpreter.get('status')}")
    print("---")

# If there are more results, get the next page using the next_token
if 'nextToken' in response:
    next_page = cp_client.list_code_interpreters(
        nextToken=response['nextToken']
    )
    # Process next_page...
```

API

To list Code Interpreter instances using the API, use the following call:

> **Note**
>
> For pagination, include the nextToken parameter.

```
# Using awscurl
awscurl -X POST "https://bedrock-agentcore-control.<Region>.amazonaws.com/code-
interpreters" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

Deleting an AgentCore Code Interpreter

When you no longer need a Code Interpreter, you can delete it to free up resources and avoid unnecessary charges.

> **Important**
>
> Deleting a Code Interpreter permanently removes it and all its configuration. This action cannot be undone. Make sure all active sessions are stopped before deleting a Code Interpreter.

Console

To delete a Code Interpreter using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. From the list of code interpreter tools, select the tool you want to delete.
4. Choose **Delete**.
5. In the confirmation dialog, enter the name of the code interpreter to confirm deletion.
6. Choose **Delete** to permanently delete the Code Interpreter.

AWS CLI

To delete a Code Interpreter using the AWS CLI, use the `delete-code-interpreter` command:

```
aws bedrock-agentcore delete-code-interpreter \
  --region <Region> \
  --code-interpreter-id "<your-code-interpreter-id>"
```

Boto3

To delete a Code Interpreter using the AWS SDK for Python, use the `delete_code_interpreter` method:

```
import boto3

# Initialize the boto3 client
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore-control.<Region>.amazonaws.com"
)

# Delete a Code Interpreter
response = cp_client.delete_code_interpreter(
    codeInterpreterId="<your-code-interpreter-id>"
)

print("Code Interpreter deleted successfully")
```

API

To delete a Code Interpreter instance using the API, use the following call:

```
# Using awscurl
awscurl -X DELETE "https://bedrock-agentcore-control.<Region>.amazonaws.com/code-interpreters/<your-code-interpreter-id>" \
-H "Accept: application/json" \
--service bedrock-agentcore \
--region <Region>
```

Session management

The AgentCore Code Interpreter sessions have the following characteristics:

Session timeout

Default: 900 seconds (15 minutes)

Configurable: Can be adjusted when creating sessions, up to 8 hours

Session persistence

Files and data created during a session are available throughout the session's lifetime. When the session is terminated, the session no longer persists and the data is cleaned up.

Automatic termination

Sessions automatically terminate after the configured timeout period

Multiple sessions

Multiple sessions can be active simultaneously for a single code interpreter. Each session maintains its own state and environment

Retention policy

The time to live (TTL) retention policy for the session data is 30 days.

Using isolated sessions

AgentCore Tools enable isolation of each user session to ensure secure and consistent reuse of context across multiple tool invocations. Session isolation is especially important for AI agent workloads due to their dynamic and multi-step execution patterns.

Each tool session runs in a dedicated microVM with isolated CPU, memory, and filesystem resources. This architecture guarantees that one user's tool invocation cannot access data from another user's session. Upon session completion, the microVM is fully terminated, and its memory is sanitized, thereby eliminating any risk of cross-session data leakage.

Starting a AgentCore Code Interpreter session

After creating a Code Interpreter, you can start a session to execute code.

AWS CLI

To start a Code Interpreter session using the AWS CLI, use the `start-code-interpreter-session` command:

```
aws bedrock-agentcore start-code-interpreter-session \
  --region <Region> \
  --code-interpreter-id "<your-code-interpreter-id>" \
  --name "my-code-session" \
  --description "My Code Interpreter session for data analysis" \
  --session-timeout-seconds 900
```

Boto3

To start a Code Interpreter session using the AWS SDK for Python, use the `start_code_interpreter_session` method:

```
import boto3

# Initialize the boto3 client
dp_client = boto3.client(
    'bedrock-agentcore',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore.<Region>.amazonaws.com"
)

# Start a Code Interpreter session
response = dp_client.start_code_interpreter_session(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    name="sandbox-session-1",
    sessionTimeoutSeconds=3600
)

# Print the session ID
session_id = response["sessionId"]
print(f"Session created: {session_id}")
```

API

To start a new Code Interpreter session using the API, use the following call:

```
# Using awscurl
awscurl -X PUT \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/sessions/start" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "code-session-abc12345",
    "description": "code sandbox session",
    "sessionTimeoutSeconds": 900
  }'
```

Note

You can use the managed resource ID `aws.codeinterpreter.v1` or a resource ID you get by creating a code interpreter with `CreateCodeInterpreter`.

Executing code

Once you have started a Code Interpreter session, you can execute code in the session.

Boto3

To execute code using the AWS SDK for Python, use the `invoke_code_interpreter` method:

```
import boto3
import json

# Initialize the boto3 client
dp_client = boto3.client(
    'bedrock-agentcore',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore.<Region>.amazonaws.com"
)

# Execute code in the Code Interpreter session
response = dp_client.invoke_code_interpreter(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    sessionId="<your-session-id>",
    name="executeCode",
    arguments={
        "language": "python",
        "code": 'print("Hello World!!!")'
    }
)

# Process the event stream
for event in response["stream"]:
    if "result" in event:
        result = event["result"]
        if "content" in result:
            for content_item in result["content"]:
                if content_item["type"] == "text":
                    print(content_item["text"])
```

API

To execute code in a code interpreter session using the API, use the following call:

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "id": "1",
    "name": "executeCode",
    "arguments": {
      "language": "python",
      "code": "print(\"Hello, world!\")"
    }
  }'
```

Stopping a AgentCore Code Interpreter session

When you are finished using a Code Interpreter session, you should stop it to release resources and avoid unnecessary charges.

AWS CLI

To stop a code interpreter session using the AWS CLI, use the `stop-code-interpreter-session` command:

```
aws bedrock-agentcore stop-code-interpreter-session \
  --region <Region> \
  --code-interpreter-id "<your-code-interpreter-id>" \
  --session-id "<your-session-id>"
```

Boto3

To stop a Code Interpreter session using the AWS SDK for Python, use the `stop_code_interpreter_session` method:

```
import boto3

# Initialize the boto3 client
dp_client = boto3.client(
    'bedrock-agentcore',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore.<Region>.amazonaws.com"
)

# Stop the Code Interpreter session
response = dp_client.stop_code_interpreter_session(
    codeInterpreterIdentifier="aws.codeinterpreter.v1",
    sessionId="<your-session-id>"
)

print("Session stopped successfully")
```

API

To stop a code interpreter session using the API, use the following call:

```
# Using awscurl
awscurl -X PUT \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/sessions/stop?sessionId=<your-session-id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

Code Interpreter API Reference Examples

This section provides reference examples for common Code Interpreter operations using different approaches. Each example shows how to perform the same operation using AWS CLI, Boto3 SDK, and direct API calls.

Code Execution

These examples demonstrate how to execute code in a Code Interpreter session.

Boto3

```
params = {
      "language": "python",
      "code": "print(\"Hello, world!\")"
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "executeCode",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "executeCode",
    "arguments": {
      "language": "python",
      "code": "print(\"Hello, world!\")"
    }
  }'
```

Terminal Commands

These examples demonstrate how to execute terminal commands in a Code Interpreter session.

Execute Command

Boto3

```
params = {
      "command": "ls -l"
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "executeCommand",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "executeCommand",
    "arguments": {
      "command": "ls -l"
    }
  }'
```

Start Command Execution

Boto3

```
params = {
      "command": "sleep 15 && echo Task completed successfully"
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "startCommandExecution",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "startCommandExecution",
    "arguments": {
      "command": "sleep 15 && echo Task completed successfully"
    }
  }'
```

Get Task

Boto3

```
params = {
      "taskId": "<your-task-id>"
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "getTask",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "getTask",
    "arguments": {
      "taskId": "<your-task-id>"
    }
  }'
```

Stop Command Execution Task

Boto3

```
params = {
      "taskId": "<your-task-id>"
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "stopTask",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "stopTask",
    "arguments": {
      "taskId": "<your-task-id>"
    }
  }'
```

File Management

These examples demonstrate how to manage files in a Code Interpreter session.

Write Files

Boto3

```
params = {
        "content": [{"path": "dir1/samename.txt", "text": "File in dir1"}]
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "writeFiles",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "writeFiles",
    "arguments": {
      "content": [{"path": "dir1/samename.txt", "text": "File in dir1"}]
    }
  }'
```

Read Files

Boto3

```
params = {
      "paths": ["tmp.txt"]
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "readFiles",
        "arguments": params
    })
```

API

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "readFiles",
    "arguments": {
      "paths": ["tmp.txt"]
    }
  }'
```

Remove Files

Boto3

```

params = {
      "paths": ["tmp.txt"]
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "removeFiles",
        "arguments": params
    })
```

API

```

# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "removeFiles",
    "arguments": {
      "paths": ["tmp.txt"]
    }
  }'
```

List Files

Boto3

```
params = {
      "directoryPath": ""
    }

client.invoke_code_interpreter(
    **{
        "codeInterpreterIdentifier": "aws.codeinterpreter.v1",
        "sessionId": "<your-session-id>",
        "name": "listFiles",
        "arguments": params
    })
```

API

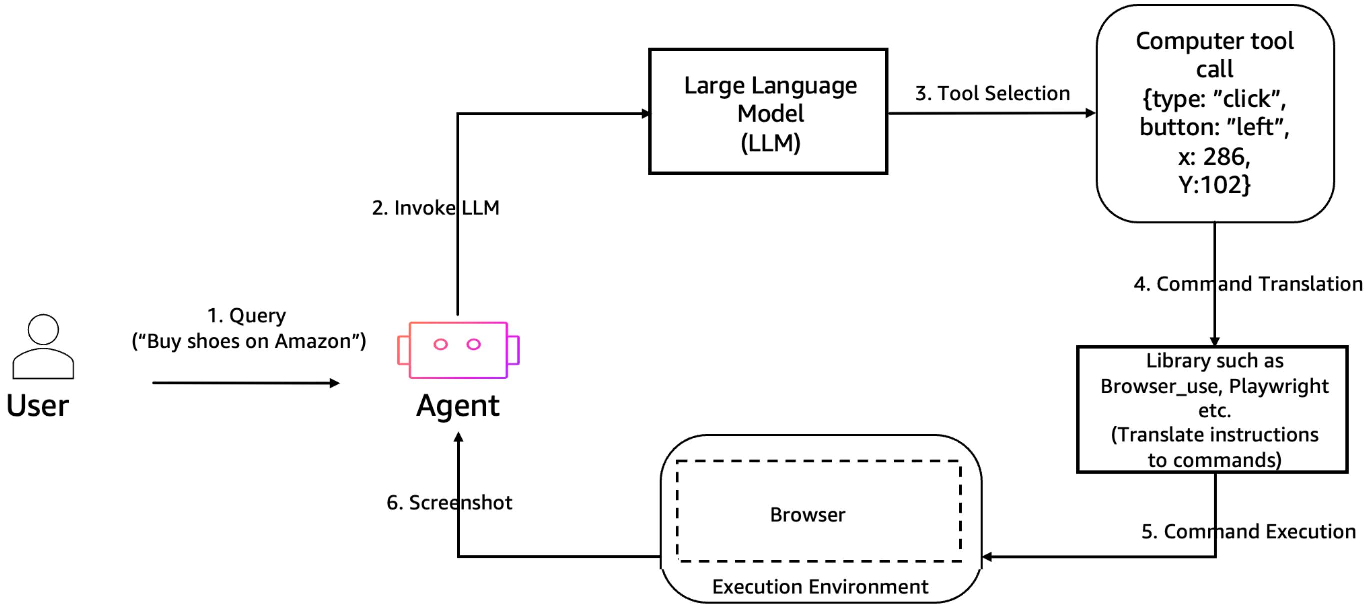
```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
aws.codeinterpreter.v1/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "listFiles",
    "arguments": {
      "directoryPath": ""
    }
  }'
```

Interact with web applications using Amazon Bedrock AgentCore Browser

The Amazon Bedrock AgentCore Browser provides a secure, cloud-based browser that enables AI agents to interact with websites. It includes security features such as session isolation, built-in observability through live viewing, CloudTrail logging, and session replay capabilities.

Overview

The Amazon Bedrock AgentCore Browser provides a secure, isolated browser environment that allows you to interact with web applications while minimizing potential risks to your system. It runs in a containerized environment within AgentCore, and isolates web activity from your local system.


User
1. Query
("Buy shoes on Amazon")
Agent
2. Invoke LLM
Large Language Model (LLM)
3. Tool Selection
Computer tool call {type: "click", button: "left", x: 286, Y:102}
4. Command Translation
Library such as Browser_use, Playwright etc. (Translate instructions to commands)
5. Command Execution
Browser
Execution Environment
6. Screenshot

Why use remote browsers for agent development?

A remote browser runs in a separate environment rather than on the local machine. For agent development, remote browsers allow AI agents to interact with the web as humans do.

Remote browsers provide the following capabilities for agent development:

- Web interaction capabilities for navigating websites, filling forms, clicking buttons, and parsing dynamic content
- Serverless browser infrastructure that automatically scales without infrastructure overhead
- Visual understanding through screenshots that allow agents to interpret websites as humans do
- Human intervention with live interactive view capabilities
- Isolation and security by running web interactions for each session in a separate environment
- Complex web application navigation for interfaces that require browser capabilities
- Security through session isolation and audit capabilities
- Observability with real-time visibility and recorded history of browser interactions

Remote browsers bridge the gap between AI agents and the human web, allowing agents to interact with websites designed for human users rather than being limited to APIs or static content.

Security Features

The Browser Tool includes several security features to help protect your environment:

- Isolation: The browser runs in a containerized environment, isolated from your local system
- Ephemeral sessions: Browser sessions are temporary and reset after each use
- Session timeouts: Sessions are terminated either by client or when the time to live (ttl) expires

How it works

1. **Create a Browser Tool**

   Create a Browser Tool to enable web browsing capabilities. The Browser Tool allows you to augment your agent runtime to securely interact with web applications, fill forms, navigate websites, and extract information. These interactions can be performed in a fully managed environment with low latency.

2. **Integrate it within an agent to invoke**

   Copy the built-in tool resource ARN into your runtime agent code to invoke it as part of your session. For browser use tools, you can navigate websites and interact with web elements in real-time.

3. **Assess performance using observability**

Monitor key metrics for each tool in CloudWatch to get real-time performance insights.

Getting Started with AgentCore Browser

The following sections show how you can get started with the AgentCore Browser tool.

Prerequisites

Before using the Browser Tool, ensure you meet the following requirements:

- You have an active AWS account with access to Amazon Bedrock AgentCore
- Your network allows secure WebSocket connections

Quick start

The following example demonstrates how to start a remote browser session, interact with it programmatically using Playwright, and observe the session in real time via Live View.

In this setup, Playwright is used to drive the browser automation. Simultaneously, clients can watch the browser's behavior in Live View as Playwright executes tasks. There are two primary ways to connect to the remote browser:

- Automation endpoint – Enables agents to interact with the browser programmatically using automation frameworks like Playwright
- Live View stream – Allows human users to watch the agent's interaction with the browser in real time and optionally take control for manual input.

You can set up the Browser Tool quickly by installing and using the Amazon Bedrock AgentCore SDK.

```
# Clone the SDK examples repository
git clone https://github.com/awslabs/amazon-bedrock-agentcore-samples.git

# Follow the README instructions to install dependencies
cd amazon-bedrock-agentcore-samples
pip install -r requirements.txt
```

```
# Install a browser automation framework such as Playwright (for Python)
# to programmatically control and interact with browser
pip install playwright

# For browser visualization, you'll need the BrowserViewerServer component
cd 01-tutorials/05-AgentCore-tools/02-Agent-Core-browser-tool/interactive_tools
```

After installing the SDK and Playwright, you can start a browser session with the following code:

```


from playwright.sync_api import sync_playwright, Playwright, BrowserType
from bedrock_agentcore.tools.browser_client import browser_session
from browser_viewer import BrowserViewerServer
import time
from rich.console import Console

console = Console()

def run(playwright: Playwright):
    # Create the browser session and keep it alive
    with browser_session('us-west-2') as client:
        ws_url, headers = client.generate_ws_headers()

        # Start viewer server
        viewer = BrowserViewerServer(client, port=8005)
        viewer_url = viewer.start(open_browser=True)

        # Connect using headers
        chromium: BrowserType = playwright.chromium
        browser = chromium.connect_over_cdp(
            ws_url,
            headers=headers
        )

        context = browser.contexts[0]
        page = context.pages[0]

        try:
            page.goto("https://amazon.com/")
            console.print(page.title())
            # Keep running
            while True:
                time.sleep(120)
        except KeyboardInterrupt:
            console.print("\n\n[yellow]Shutting down...[/yellow]")
            if 'client' in locals():
                client.stop()
                console.print("# Browser session terminated")
        except Exception as e:
            console.print(f"\n[red]Error: {e}[/red]")
            import traceback
            traceback.print_exc()

with sync_playwright() as playwright:
    run(playwright)
```

This example:

- Creates a browser session using the browser_session client
- Retrieve the WebSocket connection details required for automation and live view.
- Start the viewer server, which launches a browser window to display the remote browser session via Live View.
- Connect Playwright to the remote browser via automation endpoint and navigate to Amazon.com.
- Keep the session alive until manually interrupted.
- Handle cleanup gracefully by terminating the session and releasing resources when the program exits.

The BrowserViewerServer component provides a local web server that connects to the remote browser session and displays it in a browser window, allowing you to see and interact with the browser in real-time.

Building browser agents

You can build browser agents using various frameworks and libraries to automate web interactions. This section demonstrates how to build browser agents using different frameworks.

Nova Act

You can build a browser agent using Nova Act to automate web interactions:

Install dependencies

Get Nova ACT API key at https://nova.amazon.com/act

```
pip install nova-act
```

Write a browser agent using Nova Act

The following Python code shows how to write a browser agent using Nova Act. For information about obtaining the API key for Nova Act, see Amazon Nova Act documentation.

```

import time
from bedrock_agentcore.tools.browser_client import browser_session
from nova_act import NovaAct
from rich.console import Console

from browser_viewer import BrowserViewerServer

NOVA_ACT_API_KEY = "YOUR_NOVA_ACT_API_KEY"

console = Console()

def main():
    try:
        # Step 1: Create browser session
        with browser_session('us-west-2') as client:
            print("\r   # Browser ready!                    ")
            ws_url, headers = client.generate_ws_headers()

            # Step 2: Start viewer server
            console.print("\n[cyan]Step 3: Starting viewer server...[/cyan]")
            viewer = BrowserViewerServer(client, port=8005)
            viewer_url = viewer.start(open_browser=True)

            # Step 3: Use Nova Act to interact with the browser with NovaAct
            with NovaAct(
                    cdp_endpoint_url=ws_url,
                    cdp_headers=headers,
                    preview={"playwright_actuation": True},
                    nova_act_api_key=NOVA_ACT_API_KEY,
                    starting_page="https://www.amazon.com",
                ) as nova_act:
                    result = nova_act.act("Search for coffee maker and get the
 details of the lowest priced one on the first page")
```

```
                    console.print(f"\n[bold green]Nova Act Result:[/bold green]
 {result}")

            # Keep running
            while True:
                time.sleep(1)

    except KeyboardInterrupt:
        console.print("\n\n[yellow]Shutting down...[/yellow]")
        if 'client' in locals():
            client.stop()
            print("# Browser session terminated")
    except Exception as e:
        print(f"\n[red]Error: {e}[/red]")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Strands

You can build an agent that uses the Browser Tool as one of its tools using the Strands framework:

Install dependencies

Run the following commands.

```
pip install strands-agents
pip install strands-agents-tools
```

Write a browser agent using Strands

The following Python code shows how to write a browser agent using Strands

```

from strands import Agent
from strands_tools.browser import AgentCoreBrowser

def create_agent():
    """Create and configure the Strands agent with AgentCoreBrowser"""
    agent_core_browser = AgentCoreBrowser(region="us-west-2")
    agent = Agent(
        tools=[agent_core_browser.browser],
        model="us.anthropic.claude-3-7-sonnet-20250219-v1:0",
    )
    return agent

# Initialize agent globally
strands_agent = create_agent()

def invoke(payload):
    user_message = payload.get("prompt", "")
    response = strands_agent(user_message)
    return response.message["content"][0]["text"]

if __name__ == "__main__":
    response = invoke(
        {
            "prompt": "Search for macbooks on amazon.com and get the details of the
 first result"
        }
    )
```

Playwright

You can use the Playwright automation framework with the Browser Tool:

Install dependencies

Install a browser automation framework such as Playwright (for Python) to programmatically control and interact with browser.

```
pip install playwright
```

Write a browser agent using Playwright

The following Python code shows how to write a browser agent using Playwright.

```

from playwright.sync_api import sync_playwright, Playwright, BrowserType
from bedrock_agentcore.tools.browser_client import browser_session
from browser_viewer import BrowserViewerServer
```

```
import time

def run(playwright: Playwright):
    # Create the browser session and keep it alive
    with browser_session('us-west-2') as client:
        ws_url, headers = client.generate_ws_headers()

        # Start viewer server
        viewer = BrowserViewerServer(client, port=8005)
        viewer_url = viewer.start(open_browser=True)

        # Connect using headers
        chromium: BrowserType = playwright.chromium
        browser = chromium.connect_over_cdp(
            ws_url,
            headers=headers
        )

        context = browser.contexts[0]
        page = context.pages[0]

        try:
            page.goto("https://amazon.com/")
            print(page.title())
            time.sleep(120)
        finally:
            page.close()
            browser.close()

with sync_playwright() as playwright:
    run(playwright)
```

Resource and session management

The following topics show how the Amazon Bedrock AgentCore Browser works and how you can create the resources and manage sessions.

Permissions

To use the Amazon Bedrock AgentCore Browser, you need the following permissions in your IAM policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "BedrockAgentCoreInBuiltToolsFullAccess",
            "Effect": "Allow",
            "Action": [
                "bedrock-agentcore:CreateBrowser",
                "bedrock-agentcore:ListBrowsers",
                "bedrock-agentcore:GetBrowser",
                "bedrock-agentcore:DeleteBrowser",
                "bedrock-agentcore:StartBrowserSession",
                "bedrock-agentcore:ListBrowserSessions",
                "bedrock-agentcore:GetBrowserSession",
                "bedrock-agentcore:StopBrowserSession",
                "bedrock-agentcore:UpdateBrowserStream",
                "bedrock-agentcore:ConnectBrowserAutomationStream",
                "bedrock-agentcore:ConnectBrowserLiveViewStream"
            ],
            "Resource": "arn:aws:bedrock-agentcore:us-
east-1:111122223333:browser/*"
        }
    ]
}
```

If you're using session recording with S3, the execution role you provide when creating a browser needs the following permissions:

```
{
    "Sid": "BedrockAgentCoreBuiltInToolsS3Policy",
    "Effect": "Allow",
    "Action": [
        "s3:PutObject",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload"
    ],
    "Resource": "arn:aws:s3:::example-s3-bucket/example-prefix/*",
    "Condition": {
        "StringEquals": {
            "aws:ResourceAccount": "{{accountId}}"
        }
    }
}
```

You should also add the following trust policy to the execution role:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [{
        "Sid": "BedrockAgentCoreBuiltInTools",
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock-agentcore.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "111122223333"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:111122223333:*"
            }
        }
    }]
}
```

Browser setup for API operations

Run the following commands to set up your Browser Tool that is common to all control plane and data plane API operations.

```
import boto3
import uuid

REGION = "<Region>"
CP_ENDPOINT_URL = f"https://bedrock-agentcore-control.{REGION}.amazonaws.com"
DP_ENDPOINT_URL = f"https://bedrock-agentcore.{REGION}.amazonaws.com"
```

```
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name=REGION,
    endpoint_url=CP_ENDPOINT_URL
)

dp_client = boto3.client(
    'bedrock-agentcore',
    region_name=REGION,
    endpoint_url=DP_ENDPOINT_URL
)
```

Creating a Browser Tool and starting a session

1. **Create a Browser Tool**

   When configuring a Browser Tool, choose the public network setting, recording configuration for session replay, and permissions through an IAM runtime role that defines what AWS resources the Browser Tool can access.

2. **Start a session**

   The Browser Tool uses a session-based model. After creating a Browser Tool, you start a session with a configurable timeout period (default is 15 minutes). Sessions automatically terminate after the timeout period. Multiple sessions can be active simultaneously for a single Browser Tool, with each session maintaining its own state and environment.

3. **Interact with the browser**

   Once a session is started, you can interact with the browser using WebSocket-based streaming APIs. The Automation endpoint enables your agent to perform browser actions such as navigating to websites, clicking elements, filling out forms, taking screenshots, and more. Libraries like browser-use or Playwright can be used to simplify these interactions.

   Meanwhile, the Live View endpoint allows an end user to watch the browser session in real time and interact with it directly through the live stream.

4. **Stop the session**

   When you're finished using the browser session, you should stop it to release resources and avoid unnecessary charges. Sessions can be stopped manually or will automatically terminate after the configured timeout period.

Resource management

The AgentCore Browser provides two types of resources:

System ARNs

System ARNs are default resources pre-created for ease of use. These ARNs have default configuration with the most restrictive options and are available for all regions where Amazon Bedrock AgentCore is available.

Field	Value
ID	aws.browser.v1
ARN	arn:aws:bedrock-agentcore:us-east-1:aws:browser/aws.browser.v1
Name	Amazon Bedrock AgentCore Browser Tool
Description	AWS built-in browser for secure web browsing
Status	READY

Custom ARNs

Custom ARNs allow you to configure a browser tool with your own settings. You can choose the public network setting, recording configuration, security settings, and permissions through an IAM runtime role that defines what AWS resources the browser tool can access.

Network settings

The AgentCore Browser supports the public network mode. This mode allows the tool to access public internet resources. This option enables integration with external APIs and services.

Creating an AgentCore Browser

You can create a Browser Tool using the Amazon Bedrock AgentCore console, AWS CLI, or AWS SDK.

Console

To create a Browser Tool using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. Choose **Create browser tool**.
4. Provide a unique **Tool name** and optional **Description**.
5. Under **Network settings**, choose **Public network** which allows access to public internet resources. VPC is not supported.
6. Under **Session recording**, you can enable recording of browser sessions to an S3 bucket for later review.
7. Under **Permissions**, specify an IAM execution role that defines what AWS resources the Browser Tool can access.
8. Choose **Create**.

AWS CLI

To create a Browser Tool using the AWS CLI, use the `create-browser` command:

```
aws bedrock-agentcore-control create-browser \
  --region <Region> \
  --name "my-browser" \
  --description "My browser for web interaction" \
  --network-configuration '{
    "networkMode": "PUBLIC"
  }' \
  --recording '{
    "enabled": true,
    "s3Location": {
      "bucket": "my-bucket-name",
      "prefix": "sessionreplay"
    }
  }' \
  --execution-role-arn "arn:aws:iam::123456789012:role/my-execution-role"
```

Boto3

To create a Browser Tool using the AWS SDK for Python (Boto3), use the `create_browser` method:

Request Syntax

The following shows the request syntax:

```
response = cp_client.create_browser(
    name="my_custom_browser",
    description="Test browser for development",
    networkConfiguration={
        "networkMode": "PUBLIC"
    },
    executionRoleArn="arn:aws:iam::123456789012:role/Sessionreplay",
    clientToken=str(uuid.uuid4()),
    recording={
    "enabled": True,
    "s3Location": {
        "bucket": "session-record-123456789012",
        "prefix": "replay-data"
      }
    }
)
```

API

To create a new browser instance using the API, use the following call:

```
# Using awscurl
awscurl -X PUT \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "test_browser_1",
    "description": "Test sandbox for development",
    "networkConfiguration": {
      "networkMode": "PUBLIC"
    },
    "recording": {
      "enabled": true,
      "s3Location": {
        "bucket": "<your-bucket-name>",
        "prefix": "sessionreplay"
      }
    },
    "executionRoleArn": "arn:aws:iam::123456789012:role/my-execution-role"
  }'
```

Get AgentCore Browser tool

You can get information about the Browser tool in your account and view their details, status, and configurations.

Console

To get information about the Browser tool using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. The browser tools are listed in the **Browser tools** section.
4. You can choose a tool that you created to view it's details such as name, ID, status, and creation date for each browser tool.

AWS CLI

To get information about a Browser tool using the AWS CLI, use the `get-browser` command:

```
aws bedrock-agentcore-control get-browser \
  --region <Region> \
  --browser-id "<your-browser-id>"
```

Boto3

To get information about the Browser tool using the AWS SDK for Python (Boto3), use the `get_browser` method:

Request Syntax

The following shows the request syntax:

```
response = cp_client.get_browser(
    browserId="<your-browser-id>"
)
```

API

To get the browser tool using the API, use the following call:

```
# Using awscurl
awscurl -X GET \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers/<your-browser-
id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

Listing AgentCore Browser tools

You can list all browser tools in your account to view their details, status, and configurations.

Console

To list browser tools using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**.
3. The browser tools are listed in the **Browser tools** section.
4. You can view details such as name, ID, status, and creation date for each browser tool.

AWS CLI

To list browser tools using the AWS CLI, use the `list-browsers` command:

```
aws bedrock-agentcore-control list-browsers \
  --region <Region>
```

You can filter the results by type:

```
aws bedrock-agentcore-control list-browsers \
  --region <Region> \
  --type SYSTEM
```

You can also limit the number of results and use pagination:

```
aws bedrock-agentcore-control list-browsers \
  --region <Region> \
  --max-results 10 \
  --next-token "<your-pagination-token>"
```

Boto3

To list browser tools using the AWS SDK for Python (Boto3), use the `list_browsers` method:

Request Syntax

The following shows the request syntax:

```
response = cp_client.list_browsers(type="CUSTOM")
```

API

To list browser tools using the API, use the following call:

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

You can filter the results by type:

```
awscurl -X POST \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers?type=SYSTEM" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

You can also limit the number of results and use pagination:

```
awscurl -X POST \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers?
maxResults=1&nextToken=<your-pagination-token>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

Deleting an AgentCore Browser

When you no longer need a browser tool, you can delete it to free up resources. Before deleting a browser tool, make sure to stop all active sessions associated with it.

Console

To delete a Browser tool using the console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Navigate to **Built-in tools** and select your browser tool.
3. Choose **Delete** from the **Actions** menu.
4. Confirm the deletion by typing the browser tool name in the confirmation dialog.
5. Choose **Delete**.

Note

You cannot delete a browser tool that has active sessions. Stop all sessions before attempting to delete the tool.

AWS CLI

To delete a Browser tool using the AWS CLI, use the `delete-browser` command:

```
aws bedrock-agentcore-control delete-browser \
  --region <Region> \
  --browser-id "<your-browser-id>"
```

Boto3

To delete a Browser tool using the AWS SDK for Python (Boto3), use the `delete_browser` method:

Request Syntax

The following shows the request syntax:

```
response = cp_client.delete_browser(
    browserId="<your-browser-id>"
    )
```

API

To delete a browser tool using the API, use the following call:

```
# Using awscurl
awscurl -X DELETE \
  "https://bedrock-agentcore-control.<Region>.amazonaws.com/browsers/<your-browser-id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore-control \
    --region <Region>
```

Session management

The AgentCore Browser sessions have the following characteristics:

Session timeout

Default: 900 seconds (15 minutes)

Configurable: Can be adjusted when creating sessions, up to 8 hours

Session recording

Browser sessions can be recorded for later review

Recordings include network traffic and console logs

Recordings are stored in an S3 bucket specified during browser creation

Live view

Sessions can be viewed in real-time using the live view feature

Live view is available at: /browser-streams/aws.browser.v1/sessions/{session_id}/live-view

Automatic termination

Sessions automatically terminate after the configured timeout period

Multiple sessions

Multiple sessions can be active simultaneously for a single browser tool. Each session maintains its own state and environment. There can be up to a maximum of 500 sessions.

Retention policy

The time to live (TTL) retention policy for the session data is 30 days.

Using isolated sessions

AgentCore Tools enable isolation of each user session to ensure secure and consistent reuse of context across multiple tool invocations. Session isolation is especially important for AI agent workloads due to their dynamic and multi-step execution patterns.

Each tool session runs in a dedicated microVM with isolated CPU, memory, and filesystem resources. This architecture guarantees that one user's tool invocation cannot access data from another user's session. Upon session completion, the microVM is fully terminated, and its memory is sanitized, thereby eliminating any risk of cross-session data leakage.

Starting a browser session

After creating a browser, you can start a session to interact with web applications.

AWS CLI

To start a Browser session using the AWS CLI, use the `start-browser-session` command:

```
 aws bedrock-agentcore start-browser-session \
  --region <Region> \
  --browser-identifier "my-browser" \
  --name "my-browser-session" \
  --session-timeout-seconds 900
```

Boto3

To start a Browser session using the AWS SDK for Python (Boto3), use the `start_browser_session` method:

Request Syntax

The following shows the request syntax:

```
response = dp_client.start_browser_session(
    browserIdentifier="aws.browser.v1",
    name="browser-session-1",
    sessionTimeoutSeconds=3600
)
```

API

To create a new browser session using the API, use the following call:

```
# Using awscurl
awscurl -X PUT \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/aws.browser.v1/
sessions/start" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "browser-session-abc12345",
    "description": "browser sandbox session",
    "sessionTimeoutSeconds": 300
  }'
```

Get Browser session

You can get information about a browser session that you have created.

AWS CLI

To get information about a browser session using the AWS CLI, use the `get-browser-session` command:

```
aws bedrock-agentcore get-browser-session \
  --region <Region> \
  --browser-identifier "aws.browser.v1" \
  --session-id "<your-session-id>"
```

Boto3

To get information about a browser session using the AWS SDK for Python (Boto3), use the `get_browser_session` method:

Request Syntax

The following shows the request syntax:

```
response = dp_client.get_browser_session(
    browserIdentifier="aws.browser.v1",
    sessionId="<your-session-id>"
)
```

API

To get information about a browser session using the API, use the following call:

```
# Using awscurl
awscurl -X GET \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/aws.browser.v1/
sessions/get?sessionId=<your-session-id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>

{
  "browserIdentifier": "aws.browser.v1",
  "createdAt": "2025-07-14T22:16:40.713152248Z",
  "lastUpdatedAt": "2025-07-14T22:16:40.713152248Z",
  "name": "testBrowserSession1752531400",
  "sessionId": "<your-session-id>",
  "sessionReplayArtifact": null,
  "sessionTimeoutSeconds": 900,
  "status": "TERMINATED",
  "streams": {
    "automationStream": {
      "streamEndpoint": "wss://bedrock-agentcore.<Region>.amazonaws.com/browser-
streams/aws.browser.v1/sessions/<your-session-id>/automation",
      "streamStatus": "ENABLED"
    },
    "liveViewStream": {
      "streamEndpoint": "https://bedrock-agentcore.<Region>.amazonaws.com/browser-
streams/aws.browser.v1/sessions/<your-session-id>/live-view"
    }
  },
  "viewPort": {
    "height": 819,
    "width": 1456
  }
}
```

Interacting with a browser session

Once you have started a Browser session, you can interact with it using the WebSocket API.

Console

To interact with a Browser session using the console

1. Navigate to your active Browser session.
2. Use the browser interface to navigate to websites, interact with web elements, and perform other browser actions.
3. You can view the browser activity in real-time through the live view feature.

SDK

To interact with a Browser session programmatically, use the WebSocket-based streaming API with the following URL format:

```
https://bedrock-agentcore.<Region>.amazonaws.com/browser-streams/{browser_id}/
sessions/{session_id}/automation
```

You can use libraries like Playwright to establish a connection with the WebSocket and control the browser. Here's an example:

```
from playwright.sync_api import sync_playwright, Playwright, BrowserType
import os
import base64
from bedrock_agentcore.tools.browser_client import browser_session

def main(playwright: Playwright):
    # Keep browser session alive during usage
    with browser_session('us-west-2') as client:

        # Generate CDP endpoint and headers
        ws_url, headers = client.generate_ws_headers()

        # Connect to browser using headers
        chromium: BrowserType = playwright.chromium
        browser = chromium.connect_over_cdp(ws_url, headers=headers)

        # Use the first available context or create one
        context = browser.contexts[0] if browser.contexts else browser.new_context()
        page = context.pages[0] if context.pages else context.new_page()
        page.goto("https://amazon.com/")
        print("Navigated to Amazon")

        # Create CDP session for screenshot
        cdp_client = context.new_cdp_session(page)
        screenshot_data = cdp_client.send("Page.captureScreenshot", {
            "format": "jpeg",
            "quality": 80,
            "captureBeyondViewport": True
        })

        # Decode and save screenshot
        image_data = base64.b64decode(screenshot_data['data'])
        with open("screenshot.jpeg", "wb") as f:
            f.write(image_data)

        print("# Screenshot saved as screenshot.jpeg")
        page.close()
        browser.close()

with sync_playwright() as p:
    main(p)
```

The following example code shows how you can perform live view using the WebSocket-based streaming API.

```
https://bedrock-agentcore.<Region>.amazonaws.com/browser-streams/{browser_id}/
sessions/{session_id}/live-view
```

Below is the code.

```
import time
from rich.console import Console
from bedrock_agentcore.tools.browser_client import browser_session
from browser_viewer import BrowserViewerServer

console = Console()

def main():
    try:
         # Step 1: Create browser session
        with browser_session('us-west-2') as client:
            print("\r   # Browser ready!                    ")
            ws_url, headers = client.generate_ws_headers()

            # Step 2: Start viewer server
            console.print("\n[cyan]Step 3: Starting viewer server...[/cyan]")
            viewer = BrowserViewerServer(client, port=8005)
            viewer_url = viewer.start(open_browser=True)

            # Keep running
            while True:
                time.sleep(1)

    except KeyboardInterrupt:
        console.print("\n\n[yellow]Shutting down...[/yellow]")
        if 'client' in locals():
            client.stop()
            console.print("# Browser session terminated")
    except Exception as e:
        console.print(f"\n[red]Error: {e}[/red]")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Listing browser sessions

You can list all active browser sessions to monitor and manage your resources. This is useful for tracking active sessions, identifying long-running sessions, or finding sessions that need to be stopped.

AWS CLI

To list Browser sessions using the AWS CLI, use the `list-browser-sessions` command:

```
aws bedrock-agentcore list-browser-sessions \
  --region <Region> \
  --browser-id "<your-browser-id>" \
  --max-results 10
```

You can also filter sessions by status:

```
aws bedrock-agentcore list-browser-sessions \
  --region <Region> \
  --browser-id "<your-browser-id>" \
  --status "READY"
```

Boto3

To list Browser sessions using the AWS SDK for Python (Boto3), use the `list_browser_sessions` method:

Request Syntax

The following shows the request syntax:

```
response = dp_client.list_browser_sessions(
    browserIdentifier="aws.browser.v1"
)
```

You can also filter sessions by status:

```
# List only active sessions
filtered_response = dp_client.list_browser_sessions(
    browserIdentifier="aws.browser.v1",
    status="READY"
)

# Print filtered session information
for session in filtered_response['items']:
    print(f"Ready Session ID: {session['sessionId']}")
    print(f"Name: {session['name']}")
    print("---")
```

API

To list browser sessions using the API, use the following call:

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/<your-browser-id>/
sessions/list" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "maxResults": 10
  }'
```

You can also filter sessions by status:

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/aws.browser.v1/
sessions/list" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "maxResults": 10,
    "status": "READY"
  }'
```

Stopping a browser session

When you are finished using a Browser session, you should stop it to release resources and avoid unnecessary charges.

AWS CLI

To stop a Browser session using the AWS CLI, use the `stop-browser-session` command:

```
aws bedrock-agentcore stop-browser-session \
  --region <Region> \
  --browser-id "<your-browser-id>" \
  --session-id "<your-session-id>"
```

Boto3

To stop a Browser session using the AWS SDK for Python (Boto3), use the `stop_browser_session` method:

Request Syntax

The following shows the request syntax:

```

response = dp_client.stop_browser_session(
        browserIdentifier="aws.browser.v1",
        sessionId="<your-session-id>",
    )
```

API

To stop a browser session using the API, use the following call:

```

# Using awscurl
awscurl -X PUT \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/aws.browser.v1/
sessions/stop?sessionId=<your-session-id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region>
```

Updating browser streams

You can update browser streams to enable or disable automation. This is useful when you need to enter sensitive information like login credentials that you don't want the agent to see.

Boto3

```
response = dp_client.update_browser_stream(
    browserIdentifier="aws.browser.v1",
    sessionId="<your-session-id>",
    streamUpdate={
        "automationStreamUpdate": {
            "streamStatus": "DISABLED"  # or "ENABLED"
        }
    }
)
```

API

```
awscurl -X PUT \
  "https://bedrock-agentcore.<Region>.amazonaws.com/browsers/aws.browser.v1/
sessions/streams/update?sessionId=<your-session-id>" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "streamUpdate": {
            "automationStreamUpdate": {
              "streamStatus": "ENABLED"
            }
          }
  }'
```

CLI

```
aws bedrock-agentcore update-browser-stream \
  --region <Region> \
  --browser-id "<your-browser-id>" \
  --session-id "<your-session-id>" \
  --stream-update automationStreamUpdate={streamStatus=ENABLED}
```

Use cases

The AgentCore Browser can be used for a wide range of use cases, enabling AI agents to interact with web applications just as humans do. This section describes common use cases.

Common use cases

With the AgentCore Browser, you can:

- Test web applications in a secure environment
- Access online resources and services
- Perform web-based tasks and workflows
- Interact with web interfaces
- Capture screenshots and record browser sessions
- Build AI agents that can navigate the web
- Automate form submissions and data entry
- Extract information from websites
- Perform e-commerce transactions
- Monitor website changes and updates

Rendering live view using AWS DCV Web Client

Amazon Bedrock AgentCore's live view is powered by **AWS DCV**. Each browser session launches a dedicated DCV server that streams the browser interface and enables real-time user interaction.

To render the live view, you must use the **AWS DCV Web Client**, which supports interactive display within a browser. Authentication is handled via **IAM SigV4-signed query parameters**, which must be appended to the live view URL to authorize access.

The example SDK includes a lightweight **web server** that hosts the DCV Web Client and connects to the live view, enabling an end-to-end interactive experience out of the box.

If you want to directly integrate the live view experience into their own web applications, they can embed the **DCV Web Client** and generate the signed connection URL using the SDK's helper methods. This allows full customization of the UI while leveraging Amazon Bedrock AgentCore's Browser Tool capabilities.

Using Callbacks to Customize URL Parameters

The DCV Web SDK supports custom callbacks that you can use to modify the URLs used during authentication and session establishment. This feature enables advanced integration scenarios, including the ability to append custom query parameters and add AWS Signature Version 4 (SigV4) signed values to secure and authorize connections through external systems.

Customizing Authentication and connection URL:`httpExtraSearchParamsCallback`

The `authenticate` method supports a callback parameter, `httpExtraSearchParamsCallback`. Before initiating the request, you can use this callback to inject custom query parameters into the authentication URL.

When establishing a WebSocket connection to the DCV server, you can use the `httpExtraSearchParamsCallback` in the `connect` method to customize the URL used.

Example:

Example

The following shows a sample code:

```
async function startAndConnect() {
  const response = await fetch('/presigned-url');
  const { sessionId, presignedUrl: url } = await response.json();
  presignedUrl = url; // Set global variable

  dcv.setLogLevel(dcv.LogLevel.INFO);
  auth = dcv.authenticate(presignedUrl, {
    promptCredentials: onPromptCredentials,
    error: onError,
    success: (auth, result) => {
      const { sessionId, authToken } = result[0];
      connect(presignedUrl, sessionId, authToken);
    },
    httpExtraSearchParams: httpExtraSearchParamsCb
  });
}


function connect(serverUrl, sessionId, authToken) {
  dcv.connect({
    url: serverUrl,
    sessionId,
    authToken,
    divId: 'dcv-display',
    observers: {
      httpExtraSearchParams: httpExtraSearchParamsCb,
      displayLayout: displayLayoutCallbackCb,
    }
  })
    .then((conn) => {
      console.log('Connection established');
      connection = conn;
    })
    .catch((error) => {
      console.error('Connection failed:', error.message);
    });
}

function httpExtraSearchParamsCb(method, url, body) {
  const presignedUrl = getPresignedUrlForLiveViewEndpoint();
  const searchParams = new URL(presignedUrl).searchParams;

  return searchParams;
}
```

These callbacks offer fine-grained control over the URL and headers used by the SDK during key stages of session negotiation and connection, supporting advanced use cases and integration with existing security infrastructure.

Observability and session replay

The AgentCore Browser provides the following observability features:

Session replay

> You can replay browser sessions using the Amazon Bedrock AgentCore SDK to view session recordings stored in Amazon S3. This feature enables you to review past browser interactions for debugging, auditing, or training purposes. The recordings in S3 include DOM change events, browser network activity, and console logs for comprehensive session analysis.

Metrics

> You can view browser session metrics in Amazon CloudWatch, including session counts, durations, and error rates to monitor usage and performance.

Session replay

For session replay to work, you'll first need to create a Browser with recording enabled and provide the S3 bucket and prefix where you want the recording to be stored. Note that session replay is not available in the AWS managed Browser (aws.browser.v1).

Note

Session replay in Amazon Bedrock AgentCore captures DOM mutations within your browser session and replays those changes by reconstructing the DOM. During replay, the browser may make cross-origin HTTP requests to fetch external assets such as JavaScript files, CSS stylesheets, images, and other resources required to render the page accurately.

Boto3

```
## Request
response = cp_client.create_browser(
    name="my_custom_browser",
    description="Test browser for development",
    networkConfiguration={
        "networkMode": "PUBLIC"
    },
    executionRoleArn="arn:aws:iam::123456789012:role/Sessionreplay",
    clientToken=str(uuid.uuid4()),
    recording={
    "enabled": True,
    "s3Location": {
        "bucket": "session-record-123456789012",
        "prefix": "replay-data"
      }
    }
)
```

API

```
awscurl \
  --region <Region> \
  --service bedrock-agentcore \
  --request PUT \
  --header "Content-Type: application/json" \
  --data '{
    "name": "dsi_browser_2",
    "description": "Test sandbox for development",
    "networkConfiguration": {
      "networkMode": "PUBLIC"
    },
    "clientToken": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z"
  }' \
  https://bedrock-agentcore.<Region>.amazonaws.com/browsers
)
```

Permissions

The executionRoleArn will be used to write the recording data to your given s3 bucket. The IAM role should have the below permissions:

```
{
    "Sid": "BedrockAgentCoreBuiltInToolsS3Policy",
    "Effect": "Allow",
    "Action": [
        "s3:PutObject",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload"
    ],
    "Resource": "arn:aws:s3:::example-s3-bucket/example-prefix/*",
    "Condition": {
        "StringEquals": {
            "aws:ResourceAccount": "{{accountId}}"
        }
    }
}
```

Following is the trust policy for the role ARN.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [{
        "Sid": "BedrockAgentCoreBuiltInTools",
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock-agentcore.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "111122223333"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:111122223333:*"
            }
        }
    }]
}
```

Once you starts a browser session and interact with the browser either via automation end point or via live view, the session recording will start getting generated and pushed to your provided s3 in chunks.

Standalone Session Replay Viewer

A separate tool for viewing recorded browser sessions directly from S3 without creating a new browser.

- Connect directly to S3 to view recordings
- View any past recording by specifying its session ID
- Automatically finds the latest recording if no session ID is provided

```
# View the latest recording in a bucket
python view_recordings.py --bucket session-record-test-123456789012 --prefix replay-
data

# View a specific recording
python view_recordings.py --bucket session-record-test-123456789012 --prefix replay-
data --session 01JZVDG02M8MXZY2N7P3PKDQ74

# Use a specific AWS profile
python view_recordings.py --bucket session-record-test-123456789012 --prefix replay-
data --profile my-profile
```

For reference, see Standalone session replay viewer on GitHub.

Complete Browser Experience with Recording and Replay

A separate tool for viewing recorded browser sessions directly from S3 without creating a new browser.

- Create browser sessions with automatic recording to S3
- Live view with interactive control (take/release)
- Adjust display resolution on the fly
- Automatic session recording to S3
- Integrated session replay viewer for watching recordings

```
# View the latest recording in a bucket
python -m live_view_sessionreplay.browser_interactive_session
```

For reference, see Interactive browser session on GitHub.

CloudWatch Metrics

You can view the following metrics in Amazon CloudWatch:

- Session counts: The number of browser sessions that have been requested
- Session duration: The length of time browser sessions are active
- Error rates: The frequency of errors encountered during browser sessions
- Resource utilization: CPU, memory, and network usage by browser sessions

These metrics can be used to monitor the usage and performance of your browser sessions, set up alarms for abnormal behavior, and optimize your resource allocation.

Troubleshoot AgentCore built-in tools

This section provides solutions to common issues you might encounter when using Amazon Bedrock AgentCore built-in tools.

Browser tool issues

Agent cannot make progress due to CAPTCHA checks

Issue: Your agent gets blocked by CAPTCHA verification when using the Browser tool to interact with websites.

Cause: Anti-bot measures on popular websites detect automated browsing and require human verification.

Solution: Structure your agent to avoid search engines and implement the following architecture pattern:

- Use the Browser tool only for specific page actions, not general web searching
- Use non-browser MCP tools like Tavily search for general web search operations
- Consider adding a live view feature to your agent application that allows end users to take control and solve CAPTCHAs when needed

CORS errors when integrating with browser applications

Issue: Cross-Origin Resource Sharing (CORS) errors occur when building browser-based web applications that call a custom Amazon Bedrock AgentCore runtime server.

Cause: Browser security policies block cross-origin requests to your runtime server during local development or self-hosted deployment.

Solution: Add CORS middleware to your BedrockAgentCoreApp to handle cross-origin requests from your frontend:

```
from bedrock_agentcore.runtime import BedrockAgentCoreApp
from fastapi.middleware.cors import CORSMiddleware

app = BedrockAgentCoreApp()

# Add CORS middleware to allow browser requests
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],            # Customize in production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Handle browser preflight requests to /invocations
@app.options("/invocations")
async def options_handler():
    return {"message": "OK"}

@app.entrypoint
def my_agent(payload):
    return {"response": "Hello from agent"}
```

> ⚠ **Important**
>
> In production environments, replace `allow_origins=["*"]` with specific domain origins for better security.

Code Interpreter issues

For general Code Interpreter troubleshooting, see the specific documentation for the section called “AgentCore Code Interpreter: Execute code and analyze data”.

Common issues with Code Interpreter typically relate to:

- Code execution timeouts
- Memory limitations during data processing
- Package installation restrictions

For detailed troubleshooting steps, refer to the Code Interpreter tool documentation.

Amazon Bedrock AgentCore Gateway: Securely connect tools and other resources to your Gateway

Amazon Bedrock AgentCore Gateway provides an easy and secure way for developers to build, deploy, discover, and connect to tools at scale. AI agents need tools to perform real-world tasks—from querying databases to sending messages to analyzing documents. With Gateway, developers can convert APIs, Lambda functions, and existing services into Model Context Protocol (MCP)-compatible tools and make them available to agents through Gateway endpoints with just a few lines of code. Gateway supports OpenAPI, Smithy, and Lambda as input types, and is the only solution that provides both comprehensive ingress authentication and egress authentication in a fully-managed service. Gateway also provides 1-click integration with several popular tools such as Salesforce, Slack, Jira, Asana, and Zendesk. Gateway eliminates weeks of custom code development, infrastructure provisioning, and security implementation so developers can focus on building innovative agent applications.

Key benefits

Simplify tool development and integration

> Transform existing enterprise resources into agent-ready tools in just a few lines of code. Instead of spending months writing custom integration code and managing infrastructure, developers can focus on building differentiated agent capabilities while Gateway handles the undifferentiated heavy lifting of tool management and security at enterprise scale. Gateway also provides 1-click integration with several popular tools such as Salesforce, Slack, Jira, Asana, and Zendesk.

Accelerate agent development through unified access

> Enable your agents to discover and use tools through a single, secure endpoint. By combining multiple tool sources—from APIs to Lambda functions—into one unified interface, developers can build and scale agent workflows faster without managing multiple tool connections or reimplementing integrations.

Scale with confidence through intelligent tool discovery

> As your tool collection grows, help your agents find and use the right tools through contextual search. Built-in semantic search capabilities help agents effectively utilize available tools based

on their task context, improving agent performance and reducing development complexity at scale.

Comprehensive authentication

Manage both inbound authentication (verifying agent identity) and outbound authentication (connecting to tools) in a single service. Handle OAuth flows, token refresh, and secure credential storage for third-party services.

Framework compatibility

Work with popular open-source frameworks including CrewAI, LangGraph, LlamaIndex, and Strands Agents. Integrate with any model while maintaining enterprise-grade security and reliability.

Serverless infrastructure

Eliminate infrastructure management with a fully managed service that automatically scales based on demand. Built-in observability and auditing capabilities simplify monitoring and troubleshooting.

Key capabilities

Gateway provides the following key capabilities:

- **Security Guard** - Manages OAuth authorization to ensure only valid users and agents can access tools and resources.
- **Translation** - Converts agent requests using protocols like Model Context Protocol (MCP) into API requests and Lambda invocations, eliminating the need to manage protocol integration or version support.
- **Composition** - Combines multiple APIs, functions, and tools into a single MCP endpoint for streamlined agent access.
- **Secure Credential Exchange** - Handles credential injection for each tool, enabling agents to use tools with different authentication requirements seamlessly.
- **Semantic Tool Selection** - Enables agents to search across available tools to find the most appropriate ones for specific contexts, allowing agents to leverage thousands of tools while minimizing prompt size and reducing latency.
- **Infrastructure Manager** - Provides a serverless solution with built-in observability and auditing, eliminating infrastructure management overhead.

Quick Start with creating and using a Gateway

This section provides quick start examples for creating a gateway and using it with different frameworks.

Prerequisites

Before creating a gateway, ensure you have the following prerequisites:

- AWS account with permissions to create IAM roles, Lambda functions, and Cognito resources. You will also need permission to use Bedrock AgentCore APIs.
- AWS credentials configured on your development environment
- Python 3.6+ with boto3 installed

If you would rather create your roles, Lambda, and/or Cognito authorization server yourself, refer to the detailed setup instructions in Set up an Amazon Bedrock AgentCore gateway.

Install the key dependencies:

```
pip install boto3
pip install bedrock-agentcore-starter-toolkit
pip install bedrock-agentcore
pip install strands-agents
```

For detailed setup instructions, see Set up an Amazon Bedrock AgentCore gateway.

Creating a Gateway and attaching a Target

Now, let's dive in! Let's first create a Gateway and attach a Lambda Target:

```
from bedrock_agentcore_starter_toolkit.operations.gateway.client import GatewayClient
import logging

# setup the client
client = GatewayClient(region_name="us-east-1")
client.logger.setLevel(logging.DEBUG)
```

```
# create cognito authorizer
cognito_response = client.create_oauth_authorizer_with_cognito("TestGateway")

# create the gateway
gateway =
 client.create_mcp_gateway(authorizer_config=cognito_response["authorizer_config"])

# create a lambda target
lambda_target = client.create_mcp_gateway_target(gateway=gateway, target_type="lambda")
```

You can customize your Gateway and Targets using your own role, Lambda functions, etc.:

```
# create the gateway.
gateway = client.create_mcp_gateway(
    name=None, # the name of the Gateway - if you don't set one, one will be generated.
    role_arn=None, # the role arn that the Gateway will use - if you don't set one, one
 will be created.
    authorizer_config=cognito_response["authorizer_config"], # the OAuth authorizer
 details for authorizing callers to your Gateway (MCP only supports OAuth).
    enable_semantic_search=True, # enable semantic search.
)
```

```
# create a lambda target.
lambda_target = client.create_mcp_gateway_target(
    gateway=gateway,
    name=None, # the name of the Target - if you don't set one, one will be generated.
    target_type="lambda", # the type of the Target - you will see other target types
 later in the tutorial.
    target_payload=None, # the target details - set this to define your own lambda if
 you pre-created one. Otherwise leave this None and one will be created for you.
    credentials=None, # you will see later in the tutorial how to use this to connect
 to APIs using API keys and OAuth credentials.
)
```

Example of target_payload for Lambda. Note this Lambda will be created for you if you don't provide a target_payload:

```
{
    "lambdaArn": "<insert your lambda arn>",
    "toolSchema": {
        "inlinePayload": [
            {
                "name": "get_weather",
                "description": "Get weather for a location",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "the location e.g. seattle, wa"
                        }
                    },
                    "required": [
                        "location"
                    ]
                }
            },
            {
                "name": "get_time",
                "description": "Get time for a timezone",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "timezone": {
                            "type": "string"
                        }
                    },
                    "required": [
                        "timezone"
                    ]
                }
            }
        ]
    }
}
```

OpenAPI and Smithy Targets

You can also add targets based on API specifications like Smithy and OpenAPI, making it possible for an Agent to get access to your APIs via Gateway.

Smithy API Model Targets

Let's add a target using the Smithy API model for DynamoDB (that is the default if no target_payload is specified):

```
# create a smithy target
smithy_target = client.create_mcp_gateway_target(gateway=gateway,
                                                 target_type="smithyModel")
```

Note: you can also use your own Smithy API model like this:

```
# create a smithy target
smithy_target = client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel",
    target_payload={
        "s3": {
            "uri": "<smithy model uri>"
        }
    }
)
```

or you can even include an inline model:

```
# create a smithy target
smithy_target = client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel",
    target_payload={
        "inlinePayload": json.dumps(<smithy model>)
    }
)
```

You can find Smithy API models for hundreds of AWS services here.

Open API Model Targets

Let's add the OpenAPI model for Brave search. You will need to sign up and get an API key to use Brave.

```
# create an openapi target w/ api key
open_api_target = client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="openApiSchema",
    target_payload={
        "s3": {
            "uri": "s3://amazonbedrockagentcore-built-sampleschemas455e0815-
oj7jujcd8xiu/brave-search-open-api.json"
        }
    },
    credentials={
        "api_key": "<api key>",
        "credential_location": "HEADER",
        "credential_parameter_name": "X-Subscription-Token"
    }
)
```

Then let's add an OpenAPI w/ OAuth target:

```
open_api_with_oauth_target = client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="openApiSchema",
    target_payload={
        "s3": {
            "uri": "<to be updated>"
        }
    },
    credentials={"oauth2_provider_config": {
        "customOauth2ProviderConfig": {
      "oauthDiscovery" : {
        "authorizationServerMetadata" : {
          "issuer" : "<endpoint>",
          "authorizationEndpoint" : "<endpoint>",
          "tokenEndpoint" : "<endpoint>"
        }
      },
      "clientId" : "<client id>",
      "clientSecret" : "<client secret>"
    }}}
)
```

Using the Gateway in an Agent

Now that we have setup a Gateway and Target, let's test it out! The following code sets up an interactive Strands agent with Amazon Bedrock:

```
from strands import Agent
import logging
from strands.models import BedrockModel
from strands.tools.mcp.mcp_client import MCPClient
from mcp.client.streamable_http import streamablehttp_client
import os

def create_streamable_http_transport(mcp_url: str, access_token: str):
       return streamablehttp_client(mcp_url, headers={"Authorization": f"Bearer
 {access_token}"})

def get_full_tools_list(client):
    more_tools = True
    tools = []
    pagination_token = None
    while more_tools:
        tmp_tools = client.list_tools_sync(pagination_token=pagination_token)
        tools.extend(tmp_tools)
        if tmp_tools.pagination_token is None:
            more_tools = False
        else:
            more_tools = True
            pagination_token = tmp_tools.pagination_token
    return tools

def run_agent(mcp_url: str, access_token: str):
    bedrockmodel = BedrockModel(
        inference_profile_id="anthropic.claude-3-7-sonnet-20250219-v1:0",
        temperature=0.7,
        streaming=True,
    )

    mcp_client = MCPClient(lambda: create_streamable_http_transport(mcp_url,
 access_token))

    with mcp_client:
        tools = get_full_tools_list(mcp_client)
        print(f"Found the following tools: {[tool.tool_name for tool in tools]}")
        agent = Agent(model=bedrockmodel,tools=tools)
        while True:
            user_input = input("\nThis is an interactive Strands Agent. Ask me
 something. When you're finished, say exit or quit: ")
            if user_input.lower() in ["exit", "quit", "bye"]:
                print("Goodbye!")
                break
            print("\nThinking...\n")
            agent(user_input)

# get access token
access_token = client.get_access_token_for_cognito(cognito_response["client_info"])

# Run your agent!
run_agent(gateway["gatewayUrl"], access_token)
```

Let's run it! Why don't you try asking the agent for the weather? Note: in this example weather details and time details are hard-coded.

Core concepts for Amazon Bedrock AgentCore Gateway

Amazon Bedrock AgentCore Gateway provides a standardized way for AI agents to discover and interact with tools. Understanding the core concepts of Gateway will help you design and implement effective tool integration strategies for your AI agents.

Key concepts

Gateway

An Gateway acts like an MCP server, providing a single access point for an agent to interact with its tools. A Gateway can have multiple targets, each representing a different tool or set of tools.

Gateway Target

A target defines the APIs or Lambda function that a Gateway will provide as tools to an agent. Targets can be Lambda functions, OpenAPI specifications, Smithy models, or other tool definitions.

AgentCore Gateway Authorizer

Since MCP only supports OAuth, each Gateway must have an attached OAuth authorizer. If you don’t have an OAuth authorization server already, you will be able to create one in this guide using Cognito.

AgentCore Credential Provider

When Gateway makes calls to your APIs or Lambda function it must use some credentials to access those functionalities. When you create a Smithy or Lambda target, Gateway uses the attached execution role to make calls to those targets. When you create an OpenAPI target, you must attach an AgentCore credential provider which stores the API Key or OAuth credentials that Gateway will use to access the OpenAPI target.

Tool types

Gateway supports several types of tools and integration methods:

OpenAPI specifications

Transform existing REST APIs into MCP-compatible tools by providing an OpenAPI specification. The gateway automatically handles the translation between MCP and REST formats.

Lambda functions

Connect Lambda functions as tools, allowing you to implement custom business logic in your preferred programming language. The gateway invokes the Lambda function and translates the response into the MCP format.

Smithy models

Use Smithy models to define your API interfaces and generate MCP-compatible tools. Smithy is a language for defining services and SDKs that can be used with AWS services. The gateway can use Smithy models to generate tools that interact with AWS services or custom APIs.

Set up an Amazon Bedrock AgentCore gateway

Amazon Bedrock AgentCore Gateway provides a unified connectivity layer between agents and the tools and resources they need to interact with. Before setting up your Gateway, it's important to understand how to specify permissions so that you can secure your gateway properly.

Gateway workflow

The Gateway workflow involves the following steps to connect your agents to external tools:

1. **Create the tools for your Gateway** - Define your tools using schemas such as OpenAPI specifications for REST APIs or JSON schemas for Lambda functions. The OpenAPI specifications or tool schemas for your tools are then parsed by Amazon Bedrock AgentCore for creating the Gateway.
2. **Create a Gateway endpoint** - Use the AWS console or AWS SDK to create the gateway that will serve as the MCP entry point. Each API endpoint or function will become an MCP-compatible tool, and will be made available through your MCP server URL. To secure the gateway, you can use inbound authorization to control the ingress to the gateway.
3. **Add targets to your Gateway** - Configure targets that define how the gateway routes requests to specific tools. To securely connect to backend resources on behalf of authenticated users, use Outbound Authorization. Together, Inbound and Outbound Authorization create a secure bridge between users and their target resources, supporting both IAM credentials and OAuth-based authentication flows.
4. **Update your agent code** - Connect your agent to the Gateway endpoint to access all configured tools through the unified MCP interface.

Topics

Prerequisites to set up a gateway

Amazon Bedrock AgentCore Gateway can connect to both AWS resources and external services. This means that along with the standard AWS Identity and Access Management (IAM) for managing permissions in Amazon Bedrock AgentCore Gateway, the permissions model supports additional external authentication mechanisms.

When working with Gateways, there are three main categories of permissions to consider:

1. Gateway Management Permissions - Permissions needed to create and manage Gateways

2. Gateway Access Permissions or Inbound Auth Configuration - Who can invoke what via the MCP protocol
3. Gateway Execution Permissions or Outbound Authorization configuration - Permissions that a Gateway needs to perform actions on other resources and services

You'll configure Gateway Access Permissions when Creating gateways in the next section, and Gateway Execution Permissions when Adding targets.

Gateway Management Permissions

These permissions allow you to create and manage Gateways. You can create a gateway specific policy (example name `BedrockAgentCoreGatewayFullAccess`) which could look like:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "bedrock-agentcore:*Gateway*",
        "bedrock-agentcore:*WorkloadIdentity",
        "bedrock-agentcore:*CredentialProvider",
        "bedrock-agentcore:*Token*",
        "bedrock-agentcore:*Access*"
      ],
      "Resource": "arn:aws:bedrock-agentcore:*:*:*gateway*"
    }
  ]
}
```

You may also need additional permissions for related services:

- `s3:GetObject` and `s3:PutObject` for storing and retrieving schemas when you configure targets based on S3
- `kms:Encrypt`, `kms:Decrypt`, `kms:GenerateDataKey*` for encryption operations

- Other service-specific permissions based on your Gateway's functionality or configuration

For more comprehensive permissions across all AgentCore services, consider using the `BedrockAgentCoreFullAccess` managed policy, especially when working with multiple AgentCore products.

If you prefer to follow the principle of least privilege, you can create a custom policy that grants only specific permissions. Here's an example of a ReadOnly Gateway permission policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "bedrock-agentcore:ListGateways",
        "bedrock-agentcore:GetGateway",
        "bedrock-agentcore:ListGatewayTargets",
        "bedrock-agentcore:GetGatewayTarget"
      ],
      "Resource": "arn:aws:bedrock-agentcore:*:*:*gateway*"
    }
  ]
}
```

Gateway Access Permissions or Inbound Auth Configuration

Unlike other AWS services, which use standard AWS IAM mechanisms for access control, Amazon Bedrock AgentCore Gateway uses JWT token-based authentication as specified in the Model Context Protocol (MCP). These configurations have to be specified as a property of the gateway.

You'll configure these permissions when Creating gateways in the next section.

Gateway Execution Permissions or Outbound Authorization configuration

When creating a Gateway, you need to provide an execution role that will be used by the Gateway to access AWS resources or external services. This role defines the permissions that the Gateway has when making requests to other services. Based on the type of target, the role would either have permissions to access the AWS resources configured for the target, or for external resources, the role would have permissions to acquire the needed authorization to invoke the external resources. You will configure these after you have setup your gateway while Adding targets.

At the very least, whatever type of target is being configured, the execution role must have a trust policy that allows the Amazon Bedrock AgentCore service to assume the role:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "GatewayAssumeRolePolicy",
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock-agentcore.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:111122223333:gateway/gateway-name-*"
        }
      }
    }
  ]
}
```

For AWS resources as targets like Lambda functions, don't forget to give the Gateway permissions to access it in that resource's (ex. Lambda's) policy as well.

Best practices for Gateway permissions

Follow the principle of least privilege

- Grant only the permissions necessary for your Gateway to function
- Use specific resource ARNs rather than wildcards when possible
- Regularly review and audit permissions

Separate roles by function

- Use different roles for management and execution
- Create separate roles for different Gateways with different purposes

Secure credential storage

- Store API keys and OAuth credentials in AWS Secrets Manager
- Rotate credentials regularly

Monitor and audit

- Enable CloudTrail logging for Gateway operations
- Regularly review access patterns and permissions usage

Use conditions in policies

- Add conditions to limit when and how permissions can be used
- Consider using source IP restrictions for management operations

Creating gateways

This guide walks you through the process of creating and configuring an Amazon Bedrock AgentCore Gateway. The Gateway serves as a unified entry point for agents to access tools and resources through the Model Context Protocol (MCP) and creating it is the first step in building your tool integration platform. When you create a gateway, you create a managed service that handles authentication and invokes callable endpoints as tools.

To create a gateway, you set up inbound authorization and configure invocable targets. Targets establish the connection between your gateway and various tool types, including Lambda functions and REST API services. Each target contains configuration details that specify the tool location, authentication requirements, and any necessary request transformation rules.

Topics

- Setting up inbound Auth
- Creating your Gateway

Setting up inbound Auth

Before creating your Gateway, you need to set up inbound authorization to validate callers attempting to access targets through your Amazon Bedrock AgentCore Gateway.

> **Note**
>
> If you're using the AgentCore SDK, the Cognito EZ Auth can configure this automatically for you, so you can skip the manual inbound Auth setup.

Inbound authorization works with OAuth authorization, where the client application must authenticate with the OAuth authorizer before using the Gateway. Your client would receive an access token which is used at runtime.

You need to specify an OAuth discovery server and client IDs/audiences when you create the gateway. You can specify the following:

- `Discovery Url` — String that must match the pattern `^.+/\.well-known/openid-configuration$` for OpenID Connect discovery URLs
- At least one of the below options depending on the chosen identity provider.
  - `Allowed audiences` — List of allowed audiences for JWT tokens
  - `Allowed clients` — List of allowed client identifiers

Setting up identity providers for Inbound Auth

For general information about setting up different identity providers, see Identity provider setup and configuration.

> **Important**
>
> Using inbound authorization based on JWT tokens will result in logging of some claims of the JWT token in CloudTrail. The entry includes the Subject of the provided web identity token. We recommend that you avoid using any personally identifiable information (PII) in

this field. For example, you could instead use a GUID or a pairwise identifier, as suggested in the OIDC specification.

Choose your preferred identity provider from the options below:

Amazon Cognito EZ Auth with AgentCore SDK

You can also set up Cognito EZ Auth with the AgentCore Python SDK. This eliminates the complexity of OAuth setup. You only need to run the following command:

```
# Import SDK and set up client
from bedrock_agentcore_starter_toolkit.operations.gateway.client import
 GatewayClient
client = GatewayClient()

# Retrieve the authorization configuration from the create response. When you create
 the gateway, specify it in the authorizer_config field
cognito_result = client.create_oauth_authorizer_with_cognito("my-gateway")
authorizer_configuration = cognito_result["authorizer_config"]
```

Amazon Cognito

Amazon Cognito provides a fully managed user directory service that can be used to authenticate users for your Gateway.

To create a Cognito user pool for machine-to-machine authentication

1. Create a user pool:

```
aws cognito-idp create-user-pool \
  --region us-west-2 \
  --pool-name "gateway-user-pool"
```

2. Note the user pool ID from the response or retrieve it using:

```
aws cognito-idp list-user-pools \
  --region us-west-2 \
  --max-results 60
```

3. Create a resource server for the user pool:

```
aws cognito-idp create-resource-server \
  --region us-west-2 \
  --user-pool-id ${UserPoolId} \
  --identifier "gateway-resource-server" \
  --name "GatewayResourceServer" \
  --scopes '[{"ScopeName":"read","ScopeDescription":"Read access"},
 {"ScopeName":"write","ScopeDescription":"Write access"}]'
```

4. Create a client for the user pool:

```
aws cognito-idp create-user-pool-client \
  --region us-west-2 \
  --user-pool-id ${UserPoolId} \
  --client-name "gateway-client" \
  --generate-secret \
  --allowed-o-auth-flows client_credentials \
  --allowed-o-auth-scopes "gateway-resource-server/read" "gateway-resource-
server/write" \
  --allowed-o-auth-flows-user-pool-client \
  --supported-identity-providers "COGNITO"
```

   Note the client ID and client secret from the response.

5. Create a domain for your user pool:

```
aws cognito-idp create-user-pool-domain \
  --domain ${DomainPrefix} \
  --user-pool-id ${UserPoolId} \
  --region ${Region}
```

   The token endpoint from which you can retrieve the bearer token can be constructed as follows:

```
https://${DomainPrefix}.auth.${Region}.amazoncognito.com/oauth2/token
```

6. Construct the discovery URL for your Cognito user pool:

```
https://cognito-idp.us-west-2.amazonaws.com/${UserPoolId}/.well-known/openid-configuration
```

7. Configure the Gateway Inbound authorization with the following values:

   - **Discovery URL**: The URL constructed in the previous step
   - **Allowed clients**: The client ID obtained when creating the user pool client

```
authorizerConfiguration= {
  "customJWTAuthorizer": {
    "discoveryUrl": "https://cognito-idp.us-west-2.amazonaws.com/user-pool-id/.well-known/openid-configuration",
    "allowedClients": ["client-id"]
  }
}
```

Auth0

Auth0 can be used as an identity provider for Gateway Inbound Auth. Follow these steps to set up Auth0 and obtain the necessary configuration values:

To set up Auth0 for Gateway authentication

1. Create an API in Auth0:

   a. Log in to your Auth0 dashboard.

   b. Navigate to "APIs" and click "Create API".

   c. Provide a name and identifier for your API (e.g., "gateway-api").

   d. Select the signing algorithm (RS256 recommended).

   e. Click "Create".

2. Configure API scopes:

   a. In the API settings, go to the "Scopes" tab.

   b. Add scopes such as "invoke:gateway" and "read:gateway".

3. Create an application:

   a. Navigate to "Applications" and click "Create Application".

   b. Select "Machine to Machine Application".

   c. Select the API you created in step 1.

   d. Authorize the application for the scopes you created.

   e. Click "Create".

4. Note the client ID and client secret from the application settings.

5. Construct the discovery URL for your Auth0 tenant:

   ```
   https://<your-domain>/.well-known/openid-configuration
   ```

   Where `<your-domain>` is your Auth0 tenant domain (e.g., "dev-example.us.auth0.com").

6. Configure the Gateway Inbound authorization with the following values:

   - **Discovery URL**: The URL constructed in the previous step
   - **Allowed audiences**: The API identifier you created in step 1

     ```
     authorizerConfiguration= {
       "customJWTAuthorizer": {
         "discoveryUrl": "https://dev-example.us.auth0.com/.well-known/openid-configuration",
         "allowedAudience": ["gateway123"]
       }
     }
     ```

To obtain a bearer token for use with the Data Plane API:

```
curl --request POST \
  --url https://<your-domain>/oauth/token \
  --header 'content-type: application/json' \
  --data '{
    "client_id":"<ClientId>",
    "client_secret":"<ClientSecret>",
    "audience":"<ApiIdentifier>",
    "grant_type":"client_credentials",
    "scope": "invoke:gateway"
  }'
```

The response will include an access token that can be used as a bearer token when making requests to the Gateway.

This setup should provide the following values to help you retrieve a bearer token:

- Client ID – The ID of the client.
- Client secret – The secret access key to authenticate access to retrieve the bearer token.
- Token endpoint – The endpoint from which to retrieve the bearer token.

Creating your Gateway

Once you have set up your identity provider, you can create your Gateway with the AWS Management Console or with the CreateGateway API operation.

When you create a gateway, you can include the following capabilities:

- **Semantic search** – Allows using semantic search to deliver contextually relevant tools. For more information, see Create a gateway with semantic search.
- **Debug mode** – Allows the return of specific error messages related to the gateway target configuration to help you with debugging. For more information, see Turn on debugging messages.

For more details on a specific method, select a tab:

AgentCore SDK

You can create a gateway with the AgentCore SDK:

```
from bedrock_agentcore_starter_toolkit.operations.gateway.client import
 GatewayClient

# Initialize the Gateway client
client = GatewayClient(region_name="us-west-2")

# EZ Auth - automatically sets up Cognito OAuth
cognito_result = client.create_oauth_authorizer_with_cognito("my-gateway")

# create the gateway.
gateway = client.create_mcp_gateway(
  name=None, # the name of the Gateway - if you don't set one, one will be
 generated.
  role_arn=None, # the role arn that the Gateway will use - if you don't set one,
 one will be created.
  authorizer_config=authorization, # Variable from inbound authorization setup
 steps. Contains the OAuth authorizer details for authorizing callers to your
 Gateway (MCP only supports OAuth).
  enable_semantic_search=True, # enable semantic search.
  exception_level="DEBUG" # enable debugging
)

print(f"MCP Endpoint: {gateway.get_mcp_url()}")
print(f"OAuth Credentials:")
print(f"  Client ID: {cognito_result['client_info']['client_id']}")
print(f"  Scope: {cognito_result['client_info']['scope']}")
```

CLI

The AgentCore CLI provides a simple way to create and manage gateways:

```
# Create a Gateway with Lambda target
agentcore create_mcp_gateway \
--name my-gateway \
--target arn:aws:lambda:us-west-2:123456789012:function:MyFunction \
--execution-role BedrockAgentCoreGatewayRole
```

The CLI automatically:

- Detects target type from ARN patterns or file extensions

- Sets up Cognito OAuth (EZ Auth)
- Detects your AWS region and account
- Builds full role ARN from role name

Console

To create your Gateway endpoint

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Choose **Gateways**.
3. Choose **Create gateway**.
4. In the **Gateway details** section:

   a. Enter a **Gateway name**

   b. Expand the **Additional configurations** section and:

      i. Enter an optional **Description** for your gateway.

      ii. (Optional) For **Instructions**, enter any special instructions or context that should be passed to tools when they are invoked.

      iii. (Optional) Optionally enable **Semantic search** to enable the built-in tool that can be used to search the tools on the gateway.
5. In the **Inbound Identity** section, configure authentication for users accessing your gateway:

   a. For **Discovery URL**, enter the OpenID Connect discovery URL for your identity provider (for example, `https://auth.example.com/.well-known/openid-configuration`).

   b. For **Allowed audiences**, enter the audience values that your gateway will accept. Add multiple audiences by choosing **Add audience**.
6. In the **Permissions** section:

   a. For **Service role**, choose an existing IAM role or create a new one that allows Amazon Bedrock AgentCore to access your tools on your behalf.

   b. (Optional) For **KMS key**, choose a customer managed key for encrypting your gateway data, or leave blank to use the default Amazon Bedrock AgentCore managed key.
7. In the **Target configuration** section:

   a. Enter a **Target name**.

   b. (Optional) Provide an optional **Target description**.

   c. For **Target type**, choose either:

      - **Lambda ARN** - To connect to an Lambda function that implements your tools
      - **REST API** - To connect to a REST API service

   d. Configure the target based on your selection:

      - **For Lambda function targets:**
        - For **Lambda ARN**, enter the ARN of your Lambda function.
        - For **Tool schema**, choose to either provide the schema inline or reference an Amazon S3 location containing your tool schema.
      - **For REST API targets:**
        - For **OpenAPI schema**, choose to either provide the schema inline or reference an Amazon S3 location containing your OpenAPI specification.

   e. (Optional) In the **Outbound authentication** section, configure authentication for accessing external services:

      - For **Authentication type**, choose **OAuth client** or **API key**.
      - Select the appropriate authentication resource from your account.

8. To add more targets, choose **Add another target** and repeat the target configuration steps.

9. Choose **Create gateway**.

After creating your gateway, you can view its details, including the endpoint URL and associated targets, in the AgentCore console. The gateway endpoint URL follows the format: `https://{gatewayId}.gateway.{region}.amazonaws.com/mcp`.

Boto3

The following Python code shows how to create a gateway with boto3 (AWS SDK for Python)

```
import boto3
# create the agentcore client
agentcore_client = boto3.client('bedrock-agentcore-control')
# create a gateway
```

```
gateway = agentcore_client.create_gateway(
  name="<target-name e.g. ProductSearch>",
  roleArn="<existing role ARN e.g. arn:aws:iam::123456789012:role/MyRole>",
  protocolType="MCP",
  authorizerType="CUSTOM_JWT",
  authorizerConfiguration= {
      "customJWTAuthorizer": {
          "discoveryUrl": "<existing discovery URL e.g. https://cognito-idp.us-
west-2.amazonaws.com/some-user-pool/.well-known/openid-configuration>",
          "allowedClients": ["<clientId>"]
      }
  }
)
```

API

Use `CreateGateway` to create a gateway. The operation requires a gateway name and protocol type, while accepting optional parameters like role ARN for IAM permissions, authorizer configuration for JWT-based authentication, and custom transform configuration for request/ response processing.

Example request

The following example creates a Gateway with MCP protocol and JWT authorization:

```
POST /gateways/ HTTP/1.1
Content-Type: application/json

{
  "name": "my-ai-gateway",
  "description": "Gateway for AI model interactions",
  "clientToken": "12345678-1234-1234-1234-123456789012",
  "roleArn": "arn:aws:iam::123456789012:role/AgentCoreGatewayRole",
  "protocolType": "MCP",
  "protocolConfiguration": {
      "mcp": {
          "version": "1.0",
          "searchType": "SEMANTIC"
      }
  },
  "authorizerConfiguration": {
      "customJWTAuthorizer": {
          "discoveryUrl": "https://auth.example.com/.well-known/openid-
configuration",
          "allowedAudience": ["api.example.com"],
          "allowedClients": ["client-app-123"]
      }
  },
  "encryptionKeyArn": "arn:aws:kms:us-
east-1:123456789012:key/12345678-1234-1234-1234-123456789012"
}
```

After creating the gateway, you can call `CreateGatewayTarget` to add targets to the gateway. The operation accepts a gateway identifier in the URI path along with target specifications including the target name and configuration details.

Example request for OpenAPI target

This example creates a target using an OpenAPI schema for a product catalog service:

```
PUT /gateways/abc123def4/targets/ HTTP/1.1
Content-Type: application/json

{
"name": "ProductCatalogAPI",
"description": "Routes to product catalog and inventory service",
"targetConfiguration": {
  "mcp": {
    "openApiSchema": {
      "s3Uri": "s3://retail-schemas-bucket/catalog/product-api.json"
    }
  }
}
}
```

Create a gateway with semantic search

Semantic search enables intelligent tool discovery so that we are not limited by typical list tools limits (typically 100 or so). Our semantic search capability delivers contextually relevant tool subsets, significantly improving tool selection accuracy through focused, relevant results, inference performance with reduced token processing and overall orchestration efficiency and response times.

To enable it, add `"searchType": "SEMANTIC"` to the `CreateGateway` request in the MCP object within the `protocolConfiguration` field:

```
"protocolConfiguration": {
  "mcp": {
      "searchType": "SEMANTIC"
  }
}
```

> **Note**
>
> You can only enable it during create, you cannot update a gateway later to be able to support search.

For an identity to create a gateway with semantic search, ensure that it has permissions to use the `bedrock-agentcore:SynchronizeGatewayTargets"` IAM action.

Add targets to an existing gateway

After creating a gateway, you can add targets, which define the tools that your gateway will host. AgentCore Gateway supports multiple target type that are detailed in the following topics. Each target can have its own credential provider attached, enabling you to securely control access targets. By adding targets, your gateway becomes a single MCP URL that enables access to all of the relevant tools for an agent.

The following topics explain how target names are constructed for a gateway, how to set up outbound authorization for a target, and how to create different target types.

Topics

- Understand how AgentCore Gateway tools are named
- Setting up Outbound Auth
- Add Lambda targets to your gateway
- Add OpenAPI targets to your gateway
- Add Smithy targets to your gateway

Understand how AgentCore Gateway tools are named

The name of a tool in your gateway is dependent on the name of the target through which the tool can be accessed. Tool names are constructed in the following pattern:

```
${target_name}__${tool_name}
```

For example, if your target's name is `LambdaUsingSDK` and you have a tool named `get_order_tool` that is accessible at that target, the tool name that is visible through the Model Context Protocol (MCP) is `LambdaUsingSDK__get_order_tool`.

You should ensure that your application code accounts for the discrepancy between the tool name visible through the MCP and the tool name itself.

For an example of a Lambda handler function that strips the target name prefix from the tool name before passing it to a handler function, see Lambda function input format.

Setting up Outbound Auth

Outbound authorization lets Amazon Bedrock AgentCore gateways securely access gateway targets on behalf of users authenticated and authorized during Inbound Auth. For more information on authorization, see Prerequisites to set up a gateway.

Similar to AWS resources or Lambda functions, you authenticate by using IAM credentials. With other resources, you can use OAuth 2LO or API keys. OAuth 2LO is a type of OAuth 2.0 where a client application accesses resources on it's behalf, instead of on behalf of the user. For more information, see OAuth 2LO.

First, you register your client application with third-party providers and then create an outbound authorization with the client ID and secret. Then configure a gateway target with the outbound authorization that you created.

Topics

- Creating an Outbound Auth
- Setting up credential providers for Outbound Auth

Creating an Outbound Auth

When a user wants to access Gateway target, the gateway confirms that the access tokens (provided by Incoming Auth) are valid and if so, allows access to the target.

Console

To create an Outbound Auth

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the left navigation pane, choose **Identity**.
3. In **Outbound Auth** choose **Add OAuth client/API Key** then select the outbound authorization that you want to create.
4. If you chose OAuth client, do the following:
   a. Enter a name for the OAuth client
   b. If an included provider is the provider that you want to use, choose that provider. Then enter the client ID and client secret.
   c. Choose **Add OAuth Client**
5. If you chose **Add API Key**, enter name and the API key that you want to use, and then choose **Add**.

SDK

- Use the `CreateOauth2CredentialProvider` operation to add an OAuth outbound authorization.
- Use the `CreateApiKeyCredentialProvider` operation to add an API Key outbound authorization.

For more information, see Setting up credential providers for Outbound Auth.

Setting up credential providers for Outbound Auth

This section provides step-by-step instructions for setting up credential providers for Gateway Outbound Auth. These credential providers allow your gateway to authenticate with target services on behalf of users. For more information on setting up credential providers, see Manage credential providers with AgentCore Identity.

Choose your credential provider type from the tabs below:

IAM Role-based authentication (GATEWAY_IAM_ROLE)

When the tools registered with the gateway are AWS resources like Lambda functions, the Gateway's execution role needs appropriate permissions to access those resources.

For AWS services, use the `GATEWAY_IAM_ROLE` credential provider type in your target configuration while creating the gateway target:

```
credentialProviderConfigurations=[{
  "credentialProviderType": "GATEWAY_IAM_ROLE"
}]
```

This uses the Gateway's execution role to authenticate with AWS services.

The execution role must have permissions to access the respective resource. For example, to invoke a Lambda function, the execution role needs the `lambda:InvokeFunction` permission:

Additionally, your Lambda function needs a resource-based policy that allows the Gateway's execution role to invoke it:

You can add this policy using the AWS CLI:

```
aws lambda add-permission \
  --function-name "YourLambdaFunction" \
  --statement-id "GatewayInvoke" \
  --action "lambda:InvokeFunction" \
  --principal "arn:aws:iam::{{accountId}}:role/YourGatewayExecutionRole" \
  --region {{region}}
```

API Key authentication (API_KEY)

API Key credential providers allow your gateway to authenticate with services that use API keys for authentication. Follow these steps to set up an API Key credential provider:

To create an API Key credential provider

- Use the following AWS CLI command to create an API Key credential provider:

```
aws acps create-api-key-credential-provider \
  --region us-east-1 \
  --credential-provider-name api-key-credential-provider \
  --api-key <API_KEY_VALUE>
```

Note the provider ARN from the response. It will have a format similar to:

```
arn:aws:agent-credential-provider:us-east-1:123456789012:token-vault/default/
apikeycredentialprovider/abcdefghijk
```

When creating or updating a gateway target, you can use this credential provider in the credential provider configuration:

```
credentialProviderConfigurations=[{
  "credentialProviderType": "API_KEY",
  "credentialProvider": {
    "apiKeyCredentialProvider": {
      "providerArn": "{{credential-provider-arn}}",
      "credentialLocation": "<either HEADER OR BODY, in this case HEADER>",
      "credentialParameterName": "<name of the parameter, in this case: X-
Subscription-Token>"
    }
  }
}]
```

The `credentialLocation` can be either `HEADER` or `QUERY_PARAMETER`, depending on how the target service expects to receive the API key.

The execution role needs permission to access the API key:

```
{
  "Sid": "GetResourceApiKey",
  "Effect": "Allow",
  "Action": [
    "bedrock-agentcore:GetResourceApiKey"
  ],
  "Resource": [
    "{{credential-provider-arn}}"
  ]
}
```

For API Key authentication, if the credentials are stored in AWS Secrets Manager, the execution role also needs permission to access those secrets:

```
{
  "Sid": "GetSecretValue",
  "Effect": "Allow",
  "Action": [
    "secretsmanager:GetSecretValue"
  ],
  "Resource": [
    "{{secrets-manager-arn}}"
  ]
}

{
  "Sid": "GetAgentAccessToken",
  "Effect": "Allow",
  "Action": [
    "bedrock-agentcore:GetWorkloadAccessToken",
  ],
  "Resource": [
    "arn:aws:bedrock-agentcore:{{region}}:{{accountId}}:workload-identity-directory/
default",
    "arn:aws:bedrock-agentcore:{{region}}:{{accountId}}:workload-identity-directory/
default/workload-identity/{{gatewayName}}-*"
  ]
}
```

OAuth authentication (OAUTH)

OAuth credential providers allow your gateway to authenticate with services that use OAuth for authentication. Follow these steps to set up an OAuth credential provider:

To create an OAuth credential provider with discovery URL

- Use the following AWS CLI command to create an OAuth credential provider using a discovery URL:

```
aws acps create-oauth2-credential-provider \
  --region us-east-1 \
  --credential-provider-name oauth-credential-provider \
  --credential-provider-type CustomOAuth2 \
  --o-auth2-provider-config-input '{
    "customOAuth2ProviderConfig": {
      "oauthDiscovery": {
        "discoveryUrl": "<DiscoveryUrl>"
      },
      "clientId": "<ClientId>",
      "clientSecret": "<ClientSecret>"
    }
  }'
```

  Note the provider ARN from the response. It will have a format similar to:

```
arn:aws:agent-credential-provider:us-east-1:123456789012:token-vault/default/
oauth2credentialprovider/abcdefghijk
```

To create an OAuth credential provider with server metadata

- If you don't have a discovery URL, you can create an OAuth credential provider using server metadata:

```
aws acps create-oauth2-credential-provider \
  --region us-east-1 \
  --credential-provider-name oauth-metadata-provider \
  --credential-provider-type CustomOAuth2 \
  --o-auth2-provider-config-input '{
    "customOAuth2ProviderConfig": {
      "oauthDiscovery": {
        "authorizationServerMetadata": {
          "issuer": "https://example.auth0.com/",
          "authorizationEndpoint": "https://example.auth0.com/authorize",
          "tokenEndpoint": "https://example.auth0.com/oauth/token",
          "responseTypes": ["token"]
        }
      },
      "clientId": "<ClientId>",
      "clientSecret": "<ClientSecret>"
    }
  }'
```

When creating or updating a gateway target, you can use this credential provider in the credential provider configuration:

```
credentialProviderConfigurations=[{
  "credentialProviderType": "OAUTH",
  "credentialProvider": {
    "oauthCredentialProvider": {
      "providerArn": "{{credential-provider-arn}}",
      "scopes": ["scope1", "scope2"]
    }
  }
}]
```

The execution role needs permission to obtain OAuth tokens:

```
{
  "Sid": "GetResourceOauth2Token",
  "Effect": "Allow",
  "Action": [
    "bedrock-agentcore:GetResourceOauth2Token"
  ],
  "Resource": [
    "{{credential-provider-arn}}"
  ]
}
```

For OAuth authentication, if the credentials are stored in AWS Secrets Manager, the execution role also needs permission to access those secrets:

```
{
  "Sid": "GetSecretValue",
  "Effect": "Allow",
  "Action": [
    "secretsmanager:GetSecretValue"
  ],
  "Resource": [
    "{{secrets-manager-arn}}"
  ]
}

{
  "Sid": "GetAgentAccessToken",
  "Effect": "Allow",
  "Action": [
    "bedrock-agentcore:GetWorkloadAccessToken",
  ],
  "Resource": [
    "arn:aws:bedrock-agentcore:{{region}}:{{accountId}}:workload-identity-directory/
default",
    "arn:aws:bedrock-agentcore:{{region}}:{{accountId}}:workload-identity-directory/
default/workload-identity/{{gatewayName}}-*"
  ]
}
```

Add Lambda targets to your gateway

Lambda targets allow you to connect your gateway to AWS Lambda functions that implement your tools. This is useful when you want to execute custom code in response to tool invocations.

To add a Lambda target, you need:

- A Lambda function ARN
- A tool schema that defines the tools implemented by your Lambda function

- Credential Provider configuration for how the Gateway authenticates with the Lambda function

Prerequisites

Before adding a Lambda target, ensure you have:

- Created a Gateway: Follow the instructions in the "Set Up Gateway" guide to create your Gateway
- Created a Lambda Function: Create a Lambda function that implements the tools you want to expose
- Configured Permissions: Ensure your Gateway's execution role has permission to invoke the Lambda function

Configuring permissions

For Lambda function targets, your Gateway's execution role needs the `lambda:InvokeFunction` permission:

(Optional) Additionally, if your Lambda function was created in an account that's different from where the Gateway is being set up, it needs a resource-based policy that allows the Gateway's execution role to invoke it:

You can add this policy using the AWS CLI:

```
aws lambda add-permission \
  --function-name "YourLambdaFunction" \
  --statement-id "GatewayInvoke" \
  --action "lambda:InvokeFunction" \
  --principal "arn:aws:iam::{{accountId}}:role/YourGatewayExecutionRole" \
  --region {{region}}
```

Adding a Lambda target

You can add a Lambda target to your Gateway using one of the following methods:

CLI

The AgentCore CLI provides a simple way to add Lambda targets:

```
# Create a Gateway with Lambda target
agentcore create_mcp_gateway_target \
  --region us-east-1 \
  --gateway-arn arn:aws:bedrock-agentcore:us-east-1:123456789012:gateway/gateway-id
 \
  --gateway-url https://gateway-id.gateway.bedrock-agentcore.us-
west-2.amazonaws.com/mcp \
  --role-arn arn:aws:iam::123456789012:role/BedrockAgentCoreGatewayRole \
  --target-type lambda
```

Console

To add a target to an existing gateway

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Select the gateway to which you want to add a target.
3. Choose the **Targets** tab.
4. Choose **Add target**.
5. Enter a **Target name**.
6. (Optional) Provide an optional **Target description**.
7. For **Target type**, choose **Lambda function**.
8. For **Lambda function**, enter the ARN of your Lambda function.
9. For **Tool schema**, choose to either provide the schema inline or reference an Amazon S3 location containing your tool schema.
10. In the **Outbound authentication** section, configure authentication for accessing the Lambda function:

    - For **Authentication type**, choose **GATEWAY_IAM_ROLE**.

11. Choose **Add target**.

AgentCore SDK

You can add a Lambda target using the AgentCore SDK:

```
# create a lambda target.
lambda_target = client.create_mcp_gateway_target(
    gateway=gateway,
    name=None, # the name of the Target - if you don't set one, one will be
 generated.
    target_type="lambda", # the type of the Target - you will see other target types
 later in the tutorial.
    target_payload=None, # the target details - set this to define your own lambda
 if you pre-created one. Otherwise leave this None and one will be created for you.
    credentials=None, # you will see later in the tutorial how to use this to
 connect to APIs using API keys and OAuth credentials.
)
```

Example of target_payload for Lambda. Note this Lambda will be created for you if you don't provide a target_payload:

```
{
    "lambdaArn": "<insert your lambda arn>",
    "toolSchema": {
        "inlinePayload": [
            {
                "name": "get_weather",
                "description": "Get weather for a location",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "the location e.g. seattle, wa"
                        }
                    },
                    "required": [
                        "location"
                    ]
                }
            },
            {
                "name": "get_time",
                "description": "Get time for a timezone",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "timezone": {
                            "type": "string"
                        }
                    },
                    "required": [
                        "timezone"
                    ]
                }
            }
        ]
    }
}
```

Boto3

The following Python code shows how to add a Lambda target using boto3 (AWS SDK for Python):

```
import boto3

# Create the agentcore client
agentcore_client = boto3.client('bedrock-agentcore-control')

# Create a Lambda target
target = agentcore_client.create_gateway_target(
    gatewayIdentifier="your-gateway-id",
    name="LambdaTarget",
    targetConfiguration={
        "mcp": {
            "lambda": {
                "lambdaArn": "arn:aws:lambda:us-
west-2:123456789012:function:YourLambdaFunction",
                "toolSchema": {
                    "inlinePayload": [
                        {
                            "name": "get_weather",
                            "description": "Get weather for a location",
                            "inputSchema": {
                                "type": "object",
                                "properties": {"location": {"type": "string"}},
                                "required": ["location"],
                            },
                        },
                        {
                            "name": "get_time",
                            "description": "Get time for a timezone",
                            "inputSchema": {
                                "type": "object",
                                "properties": {"timezone": {"type": "string"}},
                                "required": ["timezone"],
                            },
                        },
                    ]
                }
            }
        }
    },
    credentialProviderConfigurations=[
        {
            "credentialProviderType": "GATEWAY_IAM_ROLE"
        }
    ]
)
```

API

Use the `CreateGatewayTarget` operation to add a Lambda target to your Gateway:

```
PUT /gateways/abc123def4/targets/ HTTP/1.1
Content-Type: application/json

{
  "gatewayIdentifier": "abc123def4",
  "name": "LambdaTarget",
  "targetConfiguration": {
    "mcp": {
      "lambda": {
        "lambdaArn": "arn:aws:lambda:us-
west-2:123456789012:function:YourLambdaFunction",
        "toolSchema": {
          "inlinePayload": [
            {
              "name": "get_weather",
              "description": "Get weather for a location",
              "inputSchema": {
                "type": "object",
                "properties": {"location": {"type": "string"}},
                "required": ["location"]
              }
            }
          ]
        }
      }
    }
  },
  "credentialProviderConfigurations": [
    {
      "credentialProviderType": "GATEWAY_IAM_ROLE"
    }
  ]
}
```

Lambda function schema

When creating a Lambda target without custom tools, the CLI auto-generates a default tool:

```
{
  "name": "invoke_function",
  "description": "Invoke the Lambda function",
  "inputSchema": {
    "type": "object",
    "properties": {},
    "required": []
  }
}
```

To specify custom tools, create a Lambda configuration file:

```
{
  "arn": "arn:aws:lambda:us-west-2:123456789012:function:MyFunction",
  "tools": [
    {
      "name": "process_data",
      "description": "Process input data",
      "inputSchema": {
        "type": "object",
        "properties": {
          "input": {"type": "string"}
        },
        "required": ["input"]
      }
    }
  ]
}
```

Each Tool in the toolSchema list should adhere to the following specification. You can optionally specify an outputSchema to provide more information to your agent about that tool.

```
{
  "name": "string",
  "description": "string", # optional
  "inputSchema": {
    "type": "string",
    "properties": {
      "string (property name)": {
        "type": "string | number | object | array | boolean | integer",
        "items": { # optional, only applicable if type is list
          "type": "string | number | object | boolean | integer"
        },
        "properties": {<same structure as this properties object>}, # optional, only
 applicable if type is object
        "required": []
      }
    },
    "required": []
  },
  "outputSchema": { # optional
    "type": "string",
    "properties": {
      "string (property name)": {
        "type": "string | number | object | array | boolean | integer",
        "items": { # optional, only applicable if type is list
```

```
          "type": "string | number | object | boolean | integer"
        },
        "properties": {<same structure as this properties object>}, # optional, only
 applicable if type is object
        "required": []
      }
    },
    "required": []
  }
}
```

Adding multiple tools to a Lambda target

When you have multiple related tools that should be handled by the same Lambda function, you can define them in a single target. This approach simplifies management and allows your Lambda function to handle different types of requests.

AWS SDK

The following example shows how to create a Lambda target with multiple tools using the AWS SDK for Python (Boto3):

```
import boto3

# Create the AgentCore client
agentcore_client = boto3.client('bedrock-agentcore-control')

# Create a Lambda target with multiple tools
agentcore_client.create_gateway_target(
    gatewayIdentifier="ProductGateway-AAAAA12345",
    name="ProductSearch",
    targetConfiguration={
        "mcp": {"lambda": {
            "lambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:product-
search",
            "toolSchema": {"inlinePayload": [
                {
                    "name": "searchProducts",
                    "description": "Searches for products based on keywords",
                    "inputSchema": {
                        "type": "object",
                        "properties": {
                            "keywords": {
                                "type": "string",
                                "description": "Search keywords"
                            },
                            "category": {
                                "type": "string",
                                "description": "Product category"
                            }
                        },
                        "required": ["keywords"]
                    }
                },
                {
                    "name": "getProductDetails",
                    "description": "Gets detailed information about a specific
 product",
                    "inputSchema": {
                        "type": "object",
                        "properties": {
                            "productId": {
                                "type": "string",
                                "description": "Unique product identifier"
                            }
                        },
                        "required": ["productId"]
                    }
                }
            ]}
        }}
    },
    credentialProviderConfigurations=[{"credentialProviderType":
 "GATEWAY_IAM_ROLE"}]
)
```

When your Lambda function is invoked, it can determine which tool is being called by examining the `bedrockAgentCoreToolName` property in the context object.

Using S3 for tool schemas

When you have a large number of tools or complex tool schemas, it might be more manageable to store your tool schemas in an Amazon S3 bucket. This approach allows you to maintain your tool definitions separately from your gateway configuration code.

AWS SDK

The following example shows how to create a Lambda target with a tool schema stored in S3:

```
import boto3

# Create the AgentCore client
agentcore_client = boto3.client('bedrock-agentcore-control')

# Create a Lambda target with a tool schema from S3
agentcore_client.create_gateway_target(
    gatewayIdentifier="ProductGateway-AAAAA12345",
    name="ProductSearch",
    targetConfiguration={
        "mcp": {"lambda": {
            "lambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:product-
search",
            "toolSchema": {"s3": {"uri": "s3://my-schemas/product-tools.json"}}
        }}
    },
    credentialProviderConfigurations=[{"credentialProviderType":
 "GATEWAY_IAM_ROLE"}]
)
```

The S3 object should contain a JSON array of tool definitions following the same format as the inline payload. For example:

```
[
  {
    "name": "searchProducts",
    "description": "Searches for products based on keywords",
    "inputSchema": {
      "type": "object",
      "properties": {
        "keywords": {
          "type": "string",
          "description": "Search keywords"
        },
        "category": {
          "type": "string",
          "description": "Product category"
        }
      },
      "required": ["keywords"]
    }
  },
  {
    "name": "getProductDetails",
    "description": "Gets detailed information about a specific product",
    "inputSchema": {
      "type": "object",
      "properties": {
        "productId": {
          "type": "string",
          "description": "Unique product identifier"
        }
      },
      "required": ["productId"]
    }
  }
]
```

To use S3 for tool schemas, ensure that the gateway's execution role has the necessary permissions to access the S3 bucket:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::my-schemas/*"
      ]
    }
  ]
}
```

Lambda function input format

When a gateway invokes a Lambda function, it passes an `event` object and a `context` object.

- **Event object** – A map of `properties` from the `inputSchema` to their values, as returned by the tool. For example, if your input schema contains the properties `keywords` and `category`, the event object could be the following:

```
{
  "keywords": "wireless headphones",
  "category": "electronics"
}
```

- **Context object** – Contains the following metadata:
  - bedrockAgentCoreMessageVersion – The version of the message.
  - bedrockAgentCoreGatewayId – The ID of the gateway that was invoked.
  - bedrockAgentCoreTargetId – The ID of the gateway target that was invoked.
  - bedrockAgentCoreToolName– The name of the tool that was called. The tool name is in the format *`${target_name}__${tool_name}`*.

  The format of the context object is as follows:

```
{
  "bedrockAgentCoreMessageVersion": "1.0",
  "bedrockAgentCoreGatewayId": "string",
  "bedrockAgentCoreTargetId": "string",
  "bedrockAgentCoreToolName": "string"
}
```

The Lambda function that you write can access the properties of the event and context object. You can use the following boilerplate code to get started:

```
# Access context properties in your Lambda function
def lambda_handler(event, context):
    # Since the visible tool name includes the target name as a prefix, we can use this
 delimiter to strip the prefix
    delimiter = "___"

    # Get the tool name from the context
    originalToolName = context.client_context.custom['bedrockAgentCoreToolName']
    toolName = originalToolName[originalToolName.index(delimiter) + len(delimiter):]

    # Get other context properties
    endpoint_id = context.client_context.custom['bedrockagentcoreEndpointId']
    target_id = context.client_context.custom['bedrockagentcoreTargetId']
    message_version = context.client_context.custom['bedrockagentcoreMessageVersion']
    session_id = context.client_context.custom['bedrockagentcoreSessionId']

    # Process the request based on the tool name
    if tool_name == 'searchProducts':
        # Handle searchProducts tool
        pass
    elif tool_name == 'getProductDetails':
        # Handle getProductDetails tool
        pass
    else:
        # Handle unknown tool
        pass
```

Limitations and considerations

When working with Lambda targets, be aware of the following limitations and considerations:

- Tool name prefixes will need to be manually stripped off from the toolname in your AWS Lambda function
- If you are using an existing AWS Lambda function and import it as a tool into the gateway, you will need to change the function code to account for a schema change for event and context objects
- The Lambda function must return a valid JSON response that can be parsed by the gateway

- Lambda function timeouts should be configured appropriately to handle the expected processing time of your tools
- Consider implementing error handling in your Lambda function to provide meaningful error messages to the client

Add OpenAPI targets to your gateway

OpenAPI (formerly known as Swagger) is a widely used standard for describing RESTful APIs. Gateway supports OpenAPI 3.0 specifications for defining API targets.

Understanding OpenAPI Targets

OpenAPI targets connect your Gateway to REST APIs defined using OpenAPI specifications. The Gateway translates incoming MCP requests into HTTP requests to these APIs and handles the response formatting.

Key components of OpenAPI targets include:

- **OpenAPI Schema**: The OpenAPI specification that describes the REST API
- **Credential Provider**: Configuration for how the Gateway authenticates with the API
- **Outbound Auth**: Configuration for authentication with external services

Required Permissions

For OpenAPI targets that use API Key or OAuth authentication, your Gateway's execution role needs permissions to access the credential provider:

For OAuth authentication:

```
{
  "Sid": "GetResourceOauth2Token",
  "Effect": "Allow",
  "Action": [
    "bedrock-agentcore:GetResourceOauth2Token"
  ],
  "Resource": [
    "arn:aws:agent-credential-provider:us-east-1:123456789012:token-vault/default/
oauth2credentialprovider/abcdefghijk"
  ]
}
```

If the credentials are stored in AWS Secrets Manager, the execution role also needs permission to access those secrets:

```
{
  "Sid": "GetSecretValue",
  "Effect": "Allow",
  "Action": [
    "secretsmanager:GetSecretValue"
  ],
  "Resource": [
    "arn:aws:secretsmanager:us-east-1:123456789012:secret:your-secret-name"
  ]
}
```

Key considerations and limitations

> ⚠ **Important**
>
> The OpenAPI specification must include `operationId` fields for all operations that you want to expose as tools. The operationId is used as the tool name in the MCP interface.

When using OpenAPI targets, keep in mind the following requirements and limitations:

- OpenAPI versions 3.0 and 3.1 are supported (Swagger 2.0 is not supported)
- The OpenAPI file must be free of semantic errors
- The server attribute needs to have a valid URL of the actual endpoint
- Only application/json content type is fully supported
- Complex schema features like oneOf, anyOf, and allOf are not supported
- Path parameter serializers and parameter serializers for query, header, and cookie parameters are not supported
- Each LLM will have ToolSpec constraints. If OpenAPI has APIs/properties/object names not compliant to ToolSpec of the respective downstream LLMs, the data plane will fail. Common

errors are property name exceeding the allowed length or the name containing unsupported character.

For best results with OpenAPI targets:

- Always include operationId in all operations
- Use simple parameter structures instead of complex serialization
- Implement authentication and authorization outside of the specification
- Stick to supported media types for maximum compatibility

OpenAPI Specification and Feature Support

The OpenAPI specification defines the REST API that your Gateway will expose. Here's an example of a simple OpenAPI specification:

Topics

- OpenAPI Feature Support
- OpenAPI Specification Format

OpenAPI Feature Support

The following table outlines the OpenAPI features that are supported and unsupported by Gateway:

OpenAPI Feature Support

Supported Features	Unsupported Features
Schema Definitions • Basic data types (string, number, integer, boolean, array, object) • Required field validation • Nested object structures • Array definitions with item specifications	**Schema Composition** • oneOf specifications • anyOf specifications • allOf specifications
HTTP Methods	**Security Schemes**

Supported Features	Unsupported Features
• Standard HTTP methods (GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS)	• Security schemes at the OpenAPI specification level (authentication must be configured using the Gateway's outbound authorization configuration)
Media Types • application/json • application/xml • multipart/form-data • application/x-www-form-urlencoded	**Media Types** • Custom media types beyond the supported list • Binary media types
Path Parameters • Simple path parameter definitions (Example: /users/{userId})	**Parameter Serialization** • Complex path parameter serializers (Example: `/users{;id\*}{?metadata}`) • Query parameter arrays with complex serialization • Header parameter serializers • Cookie parameter serializers
Query Parameters • Basic query parameter definitions • Simple string, number, and boolean types	**Callbacks and Webhooks** • Callback operations • Webhook definitions
Request/Response Bodies • JSON request and response bodies • XML request and response bodies • Standard HTTP status codes (200, 201, 400, 404, 500, etc.)	**Links** • Links between operations

OpenAPI Specification Format

When using OpenAPI specifications with Gateway, ensure that your API definitions adhere to the supported features and avoid using unsupported features to prevent errors during target creation and invocation.

Supported OpenAPI specification Example 1

Following shows an example of a supported OpenAPI specification

Example of a supported OpenAPI specification:

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Weather API",
    "version": "1.0.0",
    "description": "API for retrieving weather information"
  },
  "paths": {
    "/weather": {
      "get": {
        "summary": "Get current weather",
        "description": "Returns current weather information for a location",
        "operationId": "getCurrentWeather",
        "parameters": [
          {
            "name": "location",
            "in": "query",
            "description": "City name or coordinates",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "units",
            "in": "query",
            "description": "Units of measurement (metric or imperial)",
            "required": false,
            "schema": {
              "type": "string",
              "enum": ["metric", "imperial"],
              "default": "metric"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "Successful response",
            "content": {
              "application/json": {
                "schema": {
                  "type": "object",
                  "properties": {
                    "location": {
                      "type": "string"
                    },
                    "temperature": {
                      "type": "number"
                    },
                    "conditions": {
                      "type": "string"
                    },
                    "humidity": {
                      "type": "number"
                    }
                  }
                }
              }
            }
          },
          "400": {
            "description": "Invalid request"
          },
          "404": {
            "description": "Location not found"
          }
        }
      }
    }
  }
}
```

Supported OpenAPI Specification Example 2

Following shows another example of a supported OpenAPI specification.

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Search API",
    "version": "1.0.0",
    "description": "API for searching content"
  },
  "servers": [
    {
      "url": "https://api.example.com/v1"
    }
  ],
  "paths": {
    "/search": {
      "get": {
        "summary": "Search for content",
        "operationId": "searchContent",
        "parameters": [
          {
            "name": "query",
            "in": "query",
            "description": "Search query",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "limit",
            "in": "query",
            "description": "Maximum number of results",
            "required": false,
            "schema": {
              "type": "integer",
              "default": 10
            }
          }
        ],
        "responses": {
          "200": {
            "description": "Successful response",
            "content": {
              "application/json": {
                "schema": {
                  "type": "object",
                  "properties": {
                    "results": {
                      "type": "array",
                      "items": {
                        "type": "object",
                        "properties": {
                          "title": {
                            "type": "string"
                          },
                          "url": {
                            "type": "string"
                          },
                          "snippet": {
                            "type": "string"
                          }
                        }
                      }
                    },
                    "total": {
                      "type": "integer"
                    }
                  }
                }
              }
            }
          },
          "400": {
            "description": "Bad request"
          }
        }
      }
    }
  }
}
```

Unsupported OpenAPI schema

The following shows an example of an unsupported schema with oneOf:

```
{
  "oneOf": [
    {"$ref": "#/components/schemas/Pencil"},
    {"$ref": "#/components/schemas/Pen"}
  ]
}
```

Creating an OpenAPI target

You can add an OpenAPI target to your Gateway using one of the following methods:

Console

To add an OpenAPI target to an existing gateway

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Choose **Gateways**.
3. Select the gateway to which you want to add a target.
4. Choose the **Targets** tab.
5. Choose **Add target**.
6. Enter a **Target name**.
7. (Optional) Provide an optional **Target description**.
8. For **Target type**, choose **REST API** to connect to a REST API Service.
9. Choose **OpenAPI schema**
10. For OpenAPI schema, choose to either provide the schema inline or reference an Amazon S3 location containing your OpenAPI specification.
11. (Optional) In the **Outbound authentication** section, configure authentication for accessing external services:

    - For **Authentication type**, choose **OAuth client** or **API key**.
    - Select the appropriate authentication resource from your account.

12. Choose **Add target**.

AgentCore SDK

You can add an OpenAPI target with API Key authentication using the AgentCore SDK:

```
from bedrockagentcoresdk.gateway import GatewayClient

# Initialize the Gateway client
gateway_client = GatewayClient(region_name="us-west-2")

# Create an OpenAPI target with API Key authentication
open_api_target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="openApiSchema",
    target_payload={
        "s3": {
            "uri": "s3://your-bucket/path/to/open-api-spec.json"
        }
    },
    credentials={
        "api_key": "your-api-key",
        "credential_location": "HEADER",
        "credential_parameter_name": "X-API-Key"
    }
)
```

You can also add an OpenAPI target with OAuth authentication:

```
# Create an OpenAPI target with OAuth authentication
open_api_with_oauth_target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="openApiSchema",
    target_payload={
        "s3": {
            "uri": "s3://your-bucket/path/to/open-api-spec.json"
        }
    },
    credentials={"oauth2_provider_config": {"customOauth2ProviderConfig": {
      "oauthDiscovery": {
        "authorizationServerMetadata": {
          "issuer": "https://example.auth0.com",
          "authorizationEndpoint": "https://example.auth0.com/authorize",
          "tokenEndpoint": "https://example.auth0.com/oauth/token"
        }
      },
      "clientId": "your-client-id",
      "clientSecret": "your-client-secret"
    }}}
)
```

Boto3

The following Python code shows how to add an OpenAPI target using boto3 (AWS SDK for Python):

```
import boto3

# Create the agentcore client
agentcore_client = boto3.client('bedrock-agentcore-control')

# Create an OpenAPI target with API Key authentication
target = agentcore_client.create_gateway_target(
    gatewayIdentifier="your-gateway-id",
    name="SearchAPITarget",
    targetConfiguration={
        "mcp": {
            "openApiSchema": {
                "s3": {
                    "uri": "s3://your-bucket/path/to/open-api-spec.json",
                    "bucketOwnerAccountId": "123456789012"
                }
            }
        }
    },
    credentialProviderConfigurations=[
        {
            "credentialProviderType": "API_KEY",
            "credentialProvider": {
                "apiKeyCredentialProvider": {
                    "providerArn": "arn:aws:agent-credential-provider:us-
east-1:123456789012:token-vault/default/apikeycredentialprovider/abcdefghijk",
                    "credentialLocation": "HEADER",
                    "credentialParameterName": "X-API-Key"
                }
            }
        }
    ]
)
```

API

Use the `CreateGatewayTarget` operation to add an OpenAPI target to your Gateway:

```
PUT /gateways/abc123def4/targets/ HTTP/1.1
Content-Type: application/json

{
  "name": "SearchAPITarget",
  "description": "Target for search API",
  "targetConfiguration": {
    "mcp": {
      "openApiSchema": {
        "s3": {
          "uri": "s3://your-bucket/path/to/open-api-spec.json"
        }
      }
    }
  },
  "credentialProviderConfigurations": [
    {
      "credentialProviderType": "API_KEY",
      "credentialProvider": {
        "apiKeyCredentialProvider": {
          "providerArn": "arn:aws:agent-credential-provider:us-
east-1:123456789012:token-vault/default/apikeycredentialprovider/abcdefghijk",
          "credentialLocation": "HEADER",
          "credentialParameterName": "X-API-Key"
        }
      }
    }
  ]
}
```

Updating an OpenAPI target

You can update an existing OpenAPI target using the `UpdateGatewayTarget` API:

```
updated_target = agentcore_client.update_gateway_target(
    gatewayIdentifier="your-gateway-id",
    targetId="your-target-id",
    name="UpdatedSearchAPITarget",
    targetConfiguration={
        "mcp": {
            "openApiSchema": {
                "s3": {
                    "uri": "s3://your-bucket/path/to/updated-open-api-spec.json"
                }
            }
        }
    },
    credentialProviderConfigurations=[
        {
            "credentialProviderType": "API_KEY",
            "credentialProvider": {
                "apiKeyCredentialProvider": {
                    "providerArn": "arn:aws:agent-credential-provider:us-
east-1:123456789012:token-vault/default/apikeycredentialprovider/abcdefghijk",
                    "credentialLocation": "HEADER",
                    "credentialParameterName": "X-API-Key"
                }
            }
        }
    ]
)
```

Inline OpenAPI specifications

For small OpenAPI specifications, you can provide the specification inline when creating the target:

```
target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="openApiSchema",
    target_payload={
        "inlinePayload": """
        {
          "openapi": "3.0.0",
          "info": {
            "title": "Simple API",
            "version": "1.0.0"
          },
          "paths": {
            "/hello": {
              "get": {
                "operationId": "sayHello",
                "parameters": [
                  {
                    "name": "name",
                    "in": "query",
                    "schema": {
                      "type": "string"
                    }
                  }
                ],
                "responses": {
                  "200": {
                    "description": "OK",
                    "content": {
                      "application/json": {
                        "schema": {
                          "type": "object",
                          "properties": {
                            "message": {
                              "type": "string"
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
        """
    }
)
```

Example

Example OpenAPI target configuration with inline schema:

```
{
  "name": "OpenAPITarget",
  "targetConfiguration": {
    "mcp": {
      "openApiSchema": {
        "inlinePayload": "{\"openapi\":\"3.0.0\",\"info\":{\"title\":\"Listing
 Application API\",\"description\":\"A simple API for creating, reading, updating,
 and deleting listings\",\"version\":\"1.0.0\"},\"paths\":{\"/listings\":{\"get\":
{\"operationId\":\"listListings\",\"responses\":{\"200\":{\"description\":\"Successful
 operation\"}}}}}}}"
      }
    }
  },
  "credentialProviderConfigurations": [{
    "credentialProviderType": "GATEWAY_IAM_ROLE"
  }]
}
```

Example OpenAPI target configuration with S3 reference

```
{
  "name": "OpenAPITarget",
  "targetConfiguration": {
    "mcp": {
      "openApiSchema": {
        "s3": {
          "uri": "s3://my-bucket/api-specs/openapi.json",
          "bucketOwnerAccountId": "123456789012"
        }
      }
    }
  },
  "credentialProviderConfigurations": [{
    "credentialProviderType": "GATEWAY_IAM_ROLE"
  }]
}
```

Add Smithy targets to your gateway

Smithy is a language for defining services and SDKs that works well with Gateway. Smithy models provide a more structured approach to defining APIs compared to OpenAPI, and are particularly useful for connecting to AWS services.

Built-in Smithy Models

Gateway provides built-in Smithy models for popular AWS services. When you create a Smithy target without specifying a model, it defaults to the DynamoDB model:

```
# Create a Smithy target with default DynamoDB model
smithy_target = agentcore_client.create_gateway_target(
    gatewayIdentifier=gateway["gatewayId"],
    name="DynamoDBTarget",
    targetConfiguration={"mcp": {"smithyModel": {}}},  # Empty - uses default DynamoDB
 model
    credentialProviderConfigurations=[{"credentialProviderType": "GATEWAY_IAM_ROLE"}]
)
```

You can find Smithy API models for hundreds of AWS services in the AWS API Models repository. Popular services include:

- DynamoDB (default)
- S3
- Lambda
- EC2
- RDS

To add a Smithy target, you need:

- A Smithy model in JSON AST format
- Optional authentication configuration for accessing the service

You can provide the Smithy model in two ways:

- Inline as a JSON string
- As a reference to an S3 object

Understanding Smithy Model Targets

Smithy model targets connect your Gateway to services defined using Smithy models, such as AWS services like DynamoDB, S3, and more. The Gateway translates incoming MCP requests into API calls to these services and handles the response formatting.

Key components of Smithy model targets include:

- **Smithy Model**: The Smithy model definition that describes the service API
- **Credential Provider**: Configuration for how the Gateway authenticates with the service

Required Permissions

For Smithy model targets that access AWS services, your Gateway's execution role needs permissions to access those services. For example, for a DynamoDB target, your execution role needs permissions to perform DynamoDB operations:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*"
    }
  ]
}
```

Smithy Model Types

AgentCore Gateway supports built-in AWS service models only. Smithy models are restricted to AWS services and custom Smithy models for non-AWS services are not supported.

AgentCore Gateway provides built-in Smithy models for common AWS services via an AWS provided S3 bucket that hosts the Smithy files. You can pass the Smithy file URIs to the create target API.

When you create a Smithy model target without specifying a custom model in the AgentCoreSDK, the Gateway will use the built-in DynamoDB model.

Here's an example of creating a Smithy model target for DynamoDB:

```
# Create a Smithy model target for DynamoDB
dynamodb_target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel",
    target_name="DynamoDBTarget",
    target_description="Target for DynamoDB operations"
)
```

With this target, your Gateway will expose DynamoDB operations as tools that can be invoked through the MCP interface.

Smithy Feature Support

The following table outlines the Smithy features that are supported and unsupported by Gateway:

Smithy Feature Support

Supported Features	Unsupported Features
Service Definitions	**Endpoint Rules**

Supported Features	Unsupported Features
• Service structure definitions based on Smithy specifications • Operation definitions with input/output shapes • Resource definitions • Trait shapes	• Endpoint creation rule sets • Runtime endpoint determination based on conditions • Complex URL parameters beyond simple {region} substitution
Protocol Support • RestJson protocol • Standard HTTP request/response patterns	**Protocol Support** • RestXml protocol • JsonRpc protocol • AwsQuery protocol • Ec2Query protocol • Custom protocols
Data Types • Primitive types (string, integer, boolean, float, double) • Complex types (structures, lists, maps) • Timestamp handling • Blob data types	**Authentication** • Multiple egress authentication types for specific APIs • Complex authentication schemes requiring runtime decisions
HTTP Bindings • Basic HTTP method bindings • Simple path parameter bindings • Query parameter bindings • Header bindings for simple cases	**Operations** • Streaming operations • Operations requiring custom protocol implementations

When using Smithy models with Gateway, be aware of the following limitations:

- Maximum model size: 10MB
- Only JSON protocol bindings are fully supported

- Only RestJson protocol is supported
- Complex endpoint creation rule sets are not supported
- Only simple URL parameters like {region} are supported

Example of a supported Smithy model for a weather service:

```
namespace example.weather

use aws.protocols#restJson1
use smithy.framework#ValidationException

/// Weather service for retrieving weather information
@restJson1
service WeatherService {
    version: "1.0.0",
    operations: [GetCurrentWeather]
}

/// Get current weather for a location
@http(method: "GET", uri: "/weather")
operation GetCurrentWeather {
    input: GetCurrentWeatherInput,
    output: GetCurrentWeatherOutput,
    errors: [ValidationException]
}

structure GetCurrentWeatherInput {
    /// City name or coordinates
    @required
    @httpQuery("location")
    location: String,

    /// Units of measurement (metric or imperial)
    @httpQuery("units")
    units: Units = metric
}

structure GetCurrentWeatherOutput {
    /// Location name
    location: String,
    /// Current temperature
    temperature: Float,

    /// Weather conditions description
    conditions: String,

    /// Humidity percentage
    humidity: Float
}

enum Units {
    metric
    imperial
}
```

Example of an unsupported endpoint rules configuration:

```
@endpointRuleSet({
  "rules": [
    {
      "conditions": [{"fn": "booleanEquals", "argv": [{"ref": "UseFIPS"}, true]}],
      "endpoint": {"url": "https://weather-fips.{Region}.example.com"}
    },
    {
      "endpoint": {"url": "https://weather.{Region}.example.com"}
    }
  ]
})
```

Creating a Smithy target

You can add a Smithy target to your Gateway using one of the following methods:

Console

To add a Smithy target to an existing gateway

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Choose **Gateways**.

3. Select the gateway to which you want to add a target.
4. Choose the **Targets** tab.
5. Choose **Add target**.
6. Enter a **Target name**.
7. (Optional) Provide an optional **Target description**.
8. For **Target type**, choose **REST API**.
9. Choose **Smithy schema**.
10. For **Smithy schema**, choose to either use a built-in AWS service model or provide a custom Smithy model via S3 or inline.
11. In the **Outbound authentication** section, configure authentication for accessing the service:

    - For **Authentication type**, choose **GATEWAY_IAM_ROLE**.
12. Choose **Add target**.

AgentCore SDK

You can add a Smithy model target using the AgentCore SDK:

```
from bedrock_agentcore_starter_toolkit.operations.gateway.client import
 GatewayClient

# Initialize the Gateway client
gateway_client = GatewayClient(region_name="us-west-2")

# Create a Smithy model target for a built-in AWS service (e.g., DynamoDB)
smithy_target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel"
)

# Or create a Smithy model target with a custom model
custom_smithy_target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel",
    target_payload={
        "s3": {
            "uri": "s3://your-bucket/path/to/smithy-model.json"
        }
    }
)
```

Boto3

The following Python code shows how to add a Smithy model target using boto3 (AWS SDK for Python):

```
import boto3

# Create the agentcore client
agentcore_client = boto3.client('bedrock-agentcore-control')

# Create a Smithy model target
target = agentcore_client.create_gateway_target(
    gatewayIdentifier="your-gateway-id",
    name="DynamoDBTarget",
    targetConfiguration={
        "mcp": {
            "smithyModel": {
                "s3": {
                    "uri": "s3://your-bucket/path/to/smithy-model.json",
                    "bucketOwnerAccountId": "123456789012"
                }
            }
        }
    },
    credentialProviderConfigurations=[
        {
            "credentialProviderType": "GATEWAY_IAM_ROLE"
        }
    ]
)
```

API

Use the `CreateGatewayTarget` operation to add a Smithy model target to your Gateway:

```
PUT /gateways/abc123def4/targets/ HTTP/1.1
Content-Type: application/json

{
  "gatewayIdentifier": "abc123def4",
  "name": "DynamoDBTarget",
  "targetConfiguration": {
    "mcp": {
      "smithyModel": {
        "s3": {
          "uri": "s3://your-bucket/path/to/smithy-model.json",
          "bucketOwnerAccountId": "123456789012"
        }
      }
    }
  },
  "credentialProviderConfigurations": [
    {
      "credentialProviderType": "GATEWAY_IAM_ROLE"
    }
  ]
}
```

Updating a Smithy Model Target

You can update an existing Smithy model target using the `UpdateGatewayTarget` API:

```
updated_target = agentcore_client.update_gateway_target(
    gatewayIdentifier="your-gateway-id",
    targetId="your-target-id",
    name="UpdatedDynamoDBTarget",
    targetConfiguration={
        "mcp": {
            "smithyModel": {
                "s3": {
                    "uri": "s3://your-bucket/path/to/updated-smithy-model.json"
                }
            }
        }
    },
    credentialProviderConfigurations=[
        {
            "credentialProviderType": "GATEWAY_IAM_ROLE"
        }
    ]
)
```

Advanced Smithy Model Target Configurations

Inline Smithy Models

For small Smithy models, you can provide the model inline when creating the target:

```
target = gateway_client.create_mcp_gateway_target(
    gateway=gateway,
    target_type="smithyModel",
    target_payload={
        "inlinePayload": """
        {
            "smithy": "1.0",
            "shapes": {
                "com.example#MyService": {
                    "type": "service",
                    "version": "2023-01-01",
                    "operations": [
                        {
                            "target": "com.example#GetData"
                        }
                    ]
                },
                "com.example#GetData": {
                    "type": "operation",
                    "input": {
                        "target": "com.example#GetDataInput"
                    },
                    "output": {
                        "target": "com.example#GetDataOutput"
                    }
                },
                "com.example#GetDataInput": {
                    "type": "structure",
                    "members": {
                        "id": {
                            "target": "smithy.api#String",
                            "required": true
                        }
                    }
                },
                "com.example#GetDataOutput": {
                    "type": "structure",
                    "members": {
                        "data": {
                            "target": "smithy.api#String"
                        }
                    }
                }
            }
        }
        """
    }
)
```

Use an AgentCore gateway

After setting up your gateway with targets, you can configure your application or agent to use the gateway through the Model Context Protocol (MCP). The MCP provides a standardized way for agents to discover and invoke tools.

> **Note**
>
> AgentCore Gateway supports the following MCP versions:
>
> - 2025-03-26

You can use the following MCP operations with an AgentCore gateway:

Operation	Description
tools/call	Invokes a specific tool with the provided arguments

Operation	Description
tools/list	Lists all available tools provided by the gateway

The following topics explain how to obtain an authentication token to make requests to your gateway and describe the capabilities that AgentCore gateways offer.

Topics

Obtain an access token to make gateway requests

To make MCP requests to a gateway, you need authorize the requests with an access token from the identity provider configured for the gateway's inbound authentication. The process for obtaining the token depends on the identity provider:

- For Amazon Cognito, use the OAuth 2.0 token endpoint with client credentials flow.
- For Auth0, use the OAuth 2.0 token endpoint with client credentials flow.
- For other identity providers, refer to their documentation for obtaining access tokens.

After you receive the access token, specify it in the `Authorization` header of your request, as in the following format:

```
Authorization: Bearer ${AccessToken}
```

To see examples of how to retrieve your access token with Amazon Cognito, select one of the following methods:

curl

```
curl --http1.1 -X POST ${token_endpoint} \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d
 "grant_type=client_credentials&client_id=${client_id}&client_secret=${client_secret}"
```

Python requests package

```
import requests
import base64

def get_cognito_access_token(client_id, client_secret, token_endpoint):
    credentials = base64.b64encode(f"{client_id}:{client_secret}".encode()).decode()

    response = requests.post(
        token_endpoint,
        headers={
            "Authorization": f"Basic {credentials}",
            "Content-Type": "application/x-www-form-urlencoded"
        },
        data={"grant_type": "client_credentials"}
    )

    return response.json()["access_token"]
```

Replace the *`client_id`*, *`client_secret`*, and *`token_endpoint`* with the values that were created when you configured inbound authorization.

List available tools in an AgentCore gateway

To list all available tools that an AgentCore gateway provides, make a POST request to the gateway's MCP endpoint and specify `tools/list` as the method in the request body:

```
POST /mcp HTTP/1.1
Host: ${GatewayEndpoint}
Content-Type: application/json
Authorization: Bearer ${AccessToken}

${RequestBody}
```

Replace the following values:

- *${GatewayEndpoint}* – The URL of the gateway, as provided in the response of the CreateGateway API.
- *${AccessToken}* – The access token provided by the identity provider when you set up inbound authorization.
- *${RequestBody}* – The JSON payload of the request body, as specified in Listing tools in the Model Context Protocol (MCP). Include `tools/list` as the `method`.

> **Note**
>
> For a list of optionally supported parameters for `tools/list`, see the `params` object in the request body at Tools in the Model Context Protocol documentation. At the top of the page next to the search bar, you can select the MCP version whose documentation you want to view. Make sure that the version is one supported by Amazon Bedrock AgentCore.

The response returns a list of available tools with their names, descriptions, and parameter schemas.

Code samples for listing tools

To see examples of listing available tools in the gateway, select one of the following methods:

Python requests package

```
import requests
import json

def list_tools(gateway_url, access_token):
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}"
    }

    payload = {
        "jsonrpc": "2.0",
        "id": "list-tools-request",
        "method": "tools/list"
    }

    response = requests.post(gateway_url, headers=headers, json=payload)
    return response.json()

# Example usage
gateway_url = "https://${GatewayEndpoint}" # Replace with your actual gateway
 endpoint
access_token = "${AccessToken}" # Replace with your actual access token
tools = list_tools(gateway_url, access_token)
print(json.dumps(tools, indent=2))
```

MCP Client

```

from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def execute_mcp(
    url,
    headers=None
):
    headers = {**headers} if headers else {}
    async with streamablehttp_client(
       url=url,
       headers=headers,
    ) as (
        read_stream,
        write_stream,
        callA,
    ):
        async with ClientSession(read_stream, write_stream) as session:
            # 1. Perform initialization handshake
            print("Initializing MCP...")
            _init_response = await session.initialize()
            print(f"MCP Server Initialize successful! - {_init_response}")

            # 2. List available tools
            print("Listing tools...")
            cursor = True
            tools = []
            while cursor:
                next_cursor = cursor
                if type(cursor) == bool:
                    next_cursor = None
                list_tools_response = await session.list_tools(next_cursor)
                tools.extend(list_tools_response.tools)
                cursor = list_tools_response.nextCursor

            tool_names = []
            if tools:
                for tool in tools:
                    tool_names.append(tool.name)
            tool_names_string = "\n".join(tool_names)
            print(
                f"List MCP tools. # of tools - {len(tools)}"
               f"List of tools - \n{tool_names_string}\n"
            )
```

Strands MCP Client

```
from strands import Agent
import logging
from strands.models import BedrockModel
from strands.tools.mcp.mcp_client import MCPClient
from mcp.client.streamable_http import streamablehttp_client
import os

def create_streamable_http_transport(mcp_url: str, access_token: str):
       return streamablehttp_client(mcp_url, headers={"Authorization": f"Bearer
 {access_token}"})

def get_full_tools_list(client):
    """
    List tools w/ support for pagination
    """
    more_tools = True
    tools = []
    pagination_token = None
    while more_tools:
        tmp_tools = client.list_tools_sync(pagination_token=pagination_token)
        tools.extend(tmp_tools)
        if tmp_tools.pagination_token is None:
            more_tools = False
        else:
            more_tools = True
            pagination_token = tmp_tools.pagination_token
    return tools

def run_agent(mcp_url: str, access_token: str):
    mcp_client = MCPClient(lambda: create_streamable_http_transport(mcp_url,
 access_token))

    with mcp_client:
        tools = get_full_tools_list(mcp_client)
        print(f"Found the following tools: {[tool.tool_name for tool in tools]}")

run_agent(<MCP URL>, <Access token>)
```

LangGraph MCP Client

```
import asyncio

from langchain_mcp_adapters.client import MultiServerMCPClient


def list_tools(
    url,
    headers
):
    mcp_client = MultiServerMCPClient(
        {
            "agent": {
                "transport": "streamable_http",
                "url": url,
                "headers": headers,
            }
        }
    )
    tools = asyncio.run(mcp_client.get_tools())
    tool_details = []
    tool_names = []
    for tool in tools:
        tool_names.append(f"{tool.name}")
        tool_detail = f"{tool.name} - {tool.description} \n"
```

```
            tool_properties = tool.args_schema.get('properties', {})
            properties = []
            for property_name, tool_property in tool_properties.items():
                properties.append(f"{property_name} - {tool_property.get('description',
None)} \n")
            tool_details.append(f"{tool_detail}{"\n".join(properties)}")

        tool_details_string = "\n".join(tool_details)
        tool_names_string = "\n".join(tool_names)
        print(
            f"Langchain: List MCP tools. # of tools - {len(tools)}\n",
            f"Langchain: List of tool names - \n{tool_names_string}\n"
            f"Langchain: Details of tools - \n{tool_details_string}\n"
        )
```

> **Note**
>
> If search is enabled on the gateway, then the search tool, `x_amz_bedrock_agentcore_search` will be listed first in the response.

Call a tool in a AgentCore gateway

To call a specific tool, make a POST request to the gateway's MCP endpoint and specify `tools/call` as the method in the request body, name of the tool, and the arguments:

```
POST /mcp HTTP/1.1
Host: ${GatewayEndpoint}
Content-Type: application/json
Authorization: Bearer ${AccessToken}

${RequestBody}
```

Replace the following values:

- *${GatewayEndpoint}* – The URL of the gateway, as provided in the response of the CreateGateway API.
- *${AccessToken}* – The access token provided by the identity provider when you set up inbound authorization.

- *`${RequestBody}`* – The JSON payload of the request body, as specified in Calling tools in the Model Context Protocol (MCP). Include `tools/call` as the `method` and include the `name` of the tool and its `arguments`.

The response returns the content returned by the tool and associated metadata.

Code samples for calling tools

To see examples of listing available tools in the gateway, select one of the following methods:

curl

The following curl request shows an example request to call a tool called `searchProducts` through a gateway with the ID `mygateway-abcdefghij`.

```
curl -X POST \
  https://mygateway-abcdefghij.gateway.bedrock-agentcore.us-west-2.amazonaws.com/mcp
 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
  -d '{
    "jsonrpc": "2.0",
    "id": "invoke-tool-request",
    "method": "tools/call",
    "params": {
      "name": "searchProducts",
      "arguments": {
        "query": "wireless headphones",
        "category": "Electronics",
        "maxResults": 2,
        "priceRange": {
          "min": 50.00,
          "max": 200.00
        }
      }
    }
}'
```

Python requests package

```
import requests
import json
```

```
def call_tool(gateway_url, access_token, tool_name, arguments):
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}"
    }

    payload = {
        "jsonrpc": "2.0",
        "id": "call-tool-request",
        "method": "tools/call",
        "params": {
            "name": tool_name,
            "arguments": arguments
        }
    }

    response = requests.post(gateway_url, headers=headers, json=payload)
    return response.json()

# Example usage
gateway_url = "https://${GatewayEndpoint}" # Replace with your actual gateway
 endpoint
access_token = "${AccessToken}" # Replace with your actual access token
result = call_tool(
    gateway_url,
    access_token,
    "getOrderStatus",
    {"orderId": "ORD-12345-67890", "customerId": "CUST-98765"}
)
print(json.dumps(result, indent=2))
```

MCP Client

```
import json
from datetime import timedelta

from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def execute_mcp(
    url,
    headers=None
):
    headers = {**headers} if headers else {}
    async with streamablehttp_client(
       url=url,
       headers=headers,
    ) as (
        read_stream,
        write_stream,
        callA,
    ):
        async with ClientSession(read_stream, write_stream) as session:
            # 1. Perform initialization handshake
            print("Initializing MCP...")
            _init_response = await session.initialize()
            print(f"MCP Server Initialize successful! - {_init_response}")

            # 2. Invoke a tool
            list_tools_response = await session.list_tools()
            tools = list_tools_response.tools
            print("Invoking Tools...")
            for tool in tools:
                tool_name = tool.name
                args = {
                    "param1": "paramValue1"
                }
                invoke_tool_response = await session.call_tool(
                    tool_name,
                    arguments=args,
                    read_timeout_seconds=timedelta(seconds=60)
                )
                contents = invoke_tool_response.content
                for content in contents:
                    text = content.text
                    try:
                        content = json.dumps(json.loads(text), indent=4)
                    except:
                        content = text
                    print(
                        f"Invoke tool response: Name - {content}"
                    )
```

Strands MCP Client

> **Note**
>
> This is for invoking agent

```
import asyncio
from strands import Agent
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main():
    # Create strands agent
    agent = Agent()

    # Connect to MCP server via streamable HTTP
    async with streamablehttp_client("{gatewayURL}") as (read, write,
 get_session_id):
        async with ClientSession(read, write) as session:
            # Initialize MCP session
            await session.initialize()

            result = await
 session.call_tool('x_amz_bedrock_agentcore_searchaws___recommend',
 arguments={"query": "How do I process images?"})
            print(f"Result: {result.content}")

def create_streamable_http_transport(mcp_url: str, access_token: str):
       return streamablehttp_client(mcp_url, headers={"Authorization": f"Bearer
 {access_token}"})


if __name__ == "__main__":
    asyncio.run(main())
```

LangGraph MCP Client

> **Note**
>
> This is for invoking agent

```
import asyncio

from langgraph.prebuilt import create_react_agent

def execute_agent(
    user_prompt,
    model_id,
    tools
):
    agent = create_react_agent(model_id, tools)
    _response = asyncio.run(agent.ainvoke({
        "messages": user_prompt
    }))

    _response = _response.get('messages', {})[1].content
    print(
        f"Invoke Langchain Agents Response"
        f"Response - \n{_response}\n"
    )
    return _response
```

Errors

The `tools/call` operation can return the following types of errors:

- Errors returned as part of the HTTP status code:

  AuthenticationError

  The request failed due to invalid authentication credentials.

  HTTP Status Code: 401

  AuthorizationError

  The caller does not have permission to invoke the tool.

  HTTP Status Code: 403

  ResourceNotFoundError

  The specified tool does not exist.

  HTTP Status Code: 404

ValidationError

The provided arguments do not conform to the tool's input schema.

HTTP Status Code: 400

ToolExecutionError

An error occurred while executing the tool.

HTTP Status Code: 500

InternalServerError

An internal server error occurred.

HTTP Status Code: 500

- MCP errors. For more information about these types of errors, Error Handling in the Model Context Protocol (MCP) documentation.

Search for tools in your AgentCore gateway with a natural language query

If you enabled semantic search for your gateway when you created it, you can call the `x_amz_bedrock_agentcore_search` tool to search for tools in your gateway with a natural language query. Semantic search is particularly useful when you have many tools and need to find the most appropriate ones for your use case. To learn how to enable semantic search during gateway creation, see Creating your Gateway.

To search for a tool using this AgentCore tool, make the following POST request with the `tools/call` method to the gateway's MCP endpoint:

```
POST /mcp HTTP/1.1
Host: ${GatewayEndpoint}
Content-Type: application/json
Authorization: Bearer ${AccessToken}

{
  "jsonrpc": "2.0",
  "id": "${RequestName}",
  "method": "tools/call",
  "params": {
    "name": "x_amz_bedrock_agentcore_search",
    "arguments": {
      "query": ${Query}
    }
  }
}
```

Replace the following values:

- `${GatewayEndpoint}` – The URL of the gateway, as provided in the response of the CreateGateway API.
- `${AccessToken}` – The access token provided by the identity provider when you set up inbound authorization.
- `${RequestName}` – A name for the request.
- `${Query}` – A natural language query to search for tools.

The response returns a list of tools that are relevant to the query.

Code samples for tool searching

To see examples of using natural language queries to find tools in the gateway, select one of the following methods:

Python requests package

```
import requests
import json

def search_tools(gateway_url, access_token, query):
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}"
    }

    payload = {
        "jsonrpc": "2.0",
        "id": "search-tools-request",
        "method": "tools/call",
        "params": {
            "name": "x_amz_bedrock_agentcore_search",
            "arguments": {
                "query": query
            }
        }
    }

    response = requests.post(gateway_url, headers=headers, json=payload)
    return response.json()

# Example usage
gateway_url = "https://${GatewayEndpoint}" # Replace with your actual gateway
 endpoint
access_token = "${AccessToken}" # Replace with your actual access token
results = search_tools(gateway_url, access_token, "find order information")
print(json.dumps(results, indent=2))
```

MCP Client

```
import json
from datetime import timedelta

from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client


async def execute_mcp(
    url,
    headers=None
):
    headers = {**headers} if headers else {}
    async with streamablehttp_client(
       url=url,
       headers=headers,
    ) as (
        read_stream,
        write_stream,
        callA,
    ):
        async with ClientSession(read_stream, write_stream) as session:
            # 1. Perform initialization handshake
            print("Initializing MCP...")
            _init_response = await session.initialize()
            print(f"MCP Server Initialize successful! - {_init_response}")

            # 2. Invoke search tool
            print("Invoking search tool...")
            tool_name = "x_amz_bedrock_agentcore_search"
            args = {
                "query": "How do I process images?"
            }
            invoke_tool_response = await session.call_tool(
                tool_name,
                arguments=args,
                read_timeout_seconds=timedelta(seconds=60)
            )
            contents = invoke_tool_response.content
            for content in contents:
                text = content.text
                try:
                    content = json.dumps(json.loads(text), indent=4)
                except:
                    content = text
                print(
                    f"Invoke tool response: Name - {content}"
                )
```

Strands MCP Client

```
import asyncio
from strands import Agent
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main():
    # Create strands agent
    agent = Agent()

    # Connect to MCP server via streamable HTTP
    async with streamablehttp_client("{gatewayURL}") as (read, write,
 get_session_id):
        async with ClientSession(read, write) as session:
            # Initialize MCP session
            await session.initialize()
            result = await
 session.call_tool('x_amz_bedrock_agentcore_searchaws___recommend',
 arguments={"query": "How do I process images?"})
            print(f"Result: {result.content}")

def create_streamable_http_transport(mcp_url: str, access_token: str):
       return streamablehttp_client(mcp_url, headers={"Authorization": f"Bearer
 {access_token}"})

if __name__ == "__main__":
    asyncio.run(main())
```

LangGraph MCP Client

```
import asyncio

from langchain_mcp_adapters.client import MultiServerMCPClient
from langgraph.prebuilt import create_react_agent

url = ""
headers = {}

def filter_search_tool(
):
    mcp_client = MultiServerMCPClient(
        {
            "agent": {
                "transport": "streamable_http",
                "url": url,
                "headers": headers,
            }
        }
    )
    tools = asyncio.run(mcp_client.get_tools())
    builtin_search_tool = []
    for tool in tools:
        if tool.name == "x_amz_bedrock_agentcore_search":
            builtin_search_tool.append(tool)
    return builtin_search_tool

def execute_agent(
    user_prompt,
    model_id
):
    agent = create_react_agent(model_id, filter_search_tool())
    _response = asyncio.run(agent.ainvoke({
        "messages": user_prompt
    }))

    _response = _response.get('messages', {})[1].content
    print(
        f"Invoke Langchain Agents Response"
        f"Response - \n{_response}\n"
    )
    return _response
```

Create an agent that uses your AgentCore gateway

After creating and testing your gateway, you can create and connect AI agents to your gateway. An agent connected to your gateway is able to call the tools in the gateway and use a Amazon Bedrock model to respond to queries.

To learn how to create an agent, connect it to a gateway, and invoke it to answer queries, select one of the following methods:

Strands

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

# Replace with actual values and the Amazon Bedrock model of your choice
gateway_url = "${GatewayUrl}"
access_token = "${AccessToken}"
model_id = "us.anthropic.claude-3-7-sonnet-20250219-v1:0"

# Create an MCP Client that points to the gateway
gateway_client = MCPClient(lambda: streamablehttp_client(gateway_url,
 headers={"Authorization": f"Bearer {access_token}"}))

# Create an Agent that uses a Amazon Bedrock model and the gateway to answer
def product_agent(question: str):
    with gateway_client:
        tools = gateway_client.list_tools_sync()
        agent = Agent(
          model=model_id,
          tools=tools,
          system_prompt="try to answer the user's question using the tools
 available",
          callback_handler=None
        )
        return agent(question)

# Send a query to the agent. The agent has access to the tools in the gateway to
 help answer the query
print(product_agent("what types of shoes does Amazon sell?").message["content"])
```

LangGraph

```
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

from langgraph.prebuilt import create_react_agent
from langchain_mcp_adapters.tools import load_mcp_tools

# Replace with actual values and the Amazon Bedrock model of your choice
gateway_url = "${GatewayUrl}"
access_token = "${AccessToken}"

model_id = "openai:gpt-4.1"

async with streamablehttp_client(gateway_url, headers={"Authorization": f"Bearer
 {access_token}"}) as (read, write, _):
    async with ClientSession(read, write) as session:
        # Initialize the connection
        await session.initialize()

        # Get tools
        tools = await load_mcp_tools(session)
        agent = create_react_agent(model_id, tools)
        math_response = await agent.ainvoke({"messages": "what's (3 + 5) x 12?"})
```

Claude Code

```
#!/bin/bash
# Script to add MCP server to Claude

# Server configuration
```

```
SERVER_NAME=${ServerName} # Write your server name
GATEWAY_MCP_SERVER_URL=${GatewayUrl} # The gateway MCP URL
AUTH_TOKEN=${AuthToken} # Claude authentication token

echo "Adding MCP server to Claude..."
echo "Server Name: $SERVER_NAME"
echo "Server URL: $SERVER_URL"
echo ""

# Add the MCP server
claude mcp add "$SERVER_NAME" "$GATEWAY_MCP_SERVER_URL" \
  --transport http \
  --header "Authorization: Bearer $AUTH_TOKEN"

# Check if the command was successful
if [ $? -eq 0 ]; then
    echo "MCP server added successfully!"
    echo ""
    echo "You can now use the server with: claude --mcp-server $SERVER_NAME"
else
    echo "Failed to add MCP server"
    exit 1
fi
```

Q Dev CLI

Integration with Q Dev CLI is available through MCP protocol support.

```
# Configure Q Dev CLI to use your gateway
q config set mcp-server-url https://${GatewayId}.gateway.bedrock-
agentcore.${Region}.amazonaws.com/mcp
q config set mcp-auth-token ${AccessToken}
```

Debug your gateway

You can use different tools to help debug your gateway before putting it into a production environment, including built-in AWS and AgentCore Gateway tools, as well as external tools such as the MCP inspector.

The following topics provide more details about different methods that you can use to debug your gateway:

Topics

- Turn on debugging messages
- Use the MCP Inspector
- Monitoring gateway usage

Turn on debugging messages

While your gateway is in development, you can turn on debugging messages to return details on target configuration issues, including lambda function errors, egress authorizer errors, target specification parameter validation errors.

To turn on debugging messages, set the `exceptionLevel` value as `DEBUG` when you make a CreateGateway or UpdateGateway request. After turning on debugging messages, if an issue occurs when you submit a request through your gateway, the response will return messages to help you debug.

When you're done debugging your gateway, you can update the gateway and remove the `exceptionLevel` field, such that the message to the end user only shows an unspecified internal error.

Example

As an example, suppose that the user doesn't have permissions to invoke a Lambda function that is targeted by the gateway. A request that calls this function would return a different message depending on whether debugging is on or off:

- **Debugging on** – A detailed error message would be returned in the content's `text` field and also in the `_meta` field in the response, as in the following example:

```
{
  "jsonrpc": "2.0",
  "id": 24,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Access denied while invoking Lambda function arn:aws:lambda:us-
west-2:123456789012:function:TestGatewayLambda. Check the permissions on the Lambda
 function and Gateway execution role, and retry the request."
      }
    ],
    "_meta": {
      "debug": {
        "type": "text",
        "text": "Access denied while invoking Lambda function arn:aws:lambda:us-
west-2:123456789012:function:TestGatewayLambda. Check the permissions on the Lambda
 function and Gateway execution role, and retry the request."
      }
    },
    "isError": true
  }
}
```

- **Debugging off** – A generic error message would be returned in the content's `text` field in the response, as in the following example:

```
{
  "jsonrpc": "2.0",
  "id": 24,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "An internal error occurred. Please retry later."
      }
    ],
    "isError": true
  }
}
```

Use the MCP Inspector

The MCP Inspector, available through the Model Context Protocol (MCP), is a developer tool that helps you test and debug MCP servers by through an interactive interface. You can connect your AgentCore gateway to the MCP inspector to help you debug your gateway targets.

For more information about the MCP Inspector, see the MCP Inspector documentation.

To connect your gateway to the inspector

1. Open a terminal and run `npx @modelcontextprotocol/inspector` to do the following:

   a. Install the inspector

   b. Start the inspector on localhost.

   c. Generate a session token for authentication.

   d. Open your browser to the inspector interfacee.

2. In the inspector interface, configure the following fields:

   - **Transport Type** – Select **Streamable HTTP**
   - **URL** – Enter the gateway endpoint URL returned when you created your gateway.
   - Expand **Authentication**. The **Custom Headers** section should be pre-populated with one key-value pair.
     - The key's name should be `Authorization`.
     - The value's name should be `Bearer` when you reveal it. Change it to `Bearer: ${AccessToken}`. For more information, see Obtain an access token to make gateway requests.

3. Choose **Connect**. You will be connected to your gateway. You can use the MCP Inspector as a tool to examine and test your gateway before integrating it with your agent.

For more information about how you can inspect the MCP server that your gateway is connected to, see **Feature overview** in the Feature overview in the MCP Inspector documentation.

Important

The MCP Inspector is a development tool and should not be used in production environments. Always secure your access tokens and gateway credentials.

Troubleshooting

If you encounter issues when using the MCP Inspector with your gateway, check the following:

- **Connection issues**: Ensure that your gateway URL is correct and accessible from your network
- **Authentication issues**: Verify that your access token is valid and has not expired

- **Tool invocation errors**: Check the error messages in the response and ensure that your input parameters match the tool's schema
- **Proxy errors**: If you see errors related to the proxy connection, try restarting the MCP Inspector

Monitoring gateway usage

Gateway emits metrics to CloudWatch, allowing you to monitor its usage and performance. These metrics include:

- **Total invocations**: The number of tool invocations
- **Invocations by target type**: The number of invocations by target type (Lambda, OpenAPI, Smithy)
- **Invocations by authentication type**: The number of invocations by authentication type
- **Error count**: The number of errors, broken down by user errors and system errors
- **Throttle errors**: The number of throttle errors
- **Latency**: The latency of tool invocations, broken down by overall request latency and target execution latency

To view these metrics, go to the CloudWatch console in the account that owns the gateway resources. Navigate to Metrics → All Metrics and look for the Amazon Bedrock AgentCore namespace.

For more information, see Assess Gateway performance using monitoring and observability.

Assess Gateway performance using monitoring and observability

Amazon Bedrock AgentCore Gateway provides comprehensive observability capabilities to help you monitor and troubleshoot your tool integrations. You can track key metrics and analyze performance patterns to ensure optimal operation.

Available observability features include:

Request metrics

Monitor the number of requests processed, success rates, and error patterns across all your gateway targets.

Latency tracking

Track response times for tool invocations to identify performance bottlenecks and optimize your integrations.

Authentication events

Monitor authentication successes and failures, token refresh events, and credential-related issues.

Tool usage analytics

Analyze which tools are being used most frequently and by which agents, helping you understand usage patterns.

These metrics are available through the Amazon Bedrock AgentCore console and can be integrated with CloudWatch for custom dashboards and alerting.

Topics

- Setting up CloudWatch metrics and alarms
- Logging Gateway API calls with CloudTrail

Setting up CloudWatch metrics and alarms

Gateway publishes the following metrics to CloudWatch:

Topics

- Invocation metrics
- Usage metrics
- Setting up CloudWatch alarms

Invocation metrics

These metrics provide information about API invocations, performance, and errors.

For these metrics, the following dimensions are used:

- **Operation**: Represents the MCP operation being invoked (e.g., tools/list, tools/call)
- **Resource**: Represents the identifier of the resource (ARN)
- **Name**: Represents the version of the resource

Invocation metrics

Metric	Description	Statistics	Units
Invocations	The total number of requests made to each Data Plane API. Each API call counts as one invocation regardless of the response status.	Sum	Count
Throttles	The number of requests throttled (status code 429) by the service.	Sum	Count
SystemErrors	The number of requests which failed with 5xx status code.	Sum	Count
UserErrors	The number of requests which failed with 4xx status code except 429.	Sum	Count
Latency	The time elapsed between when the service receives the request and when it begins sending the first response token. In other words, initial response time.	Average, Minimum, Maximum, p50, p90, p99	Milliseconds
Duration	The total time elapsed between receiving the request and sending the final response token. Represents complete end-to-end processing time of the request.	Average, Minimum, Maximum, p50, p90, p99	Milliseconds
TargetExecutionTime	The total time taken to execute the target over Lambda / OpenAPI / etc. This	Average, Minimum,	Milliseconds

Metric	Description	Statistics	Units
	helps determine the contribution of the target to the total Latency.	Maximum, p50, p90, p99	

Usage metrics

These metrics provide information about how your gateway is being used.

Usage metrics

Metric	Description	Statistics	Units
TargetType	The total number of requests served by each type of target (MCP, Lambda, OpenAPI).	Sum	Count

To view these metrics in the CloudWatch console:

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. Choose the **BedrockAgentCore** namespace.
4. Choose a dimension to view the metrics (e.g., **By Endpoint**).
5. Select the metrics you want to view and choose **Add to graph**.

Setting up CloudWatch alarms

You can set up CloudWatch alarms to alert you when certain metrics exceed thresholds. For example, you might want to be notified when the error rate exceeds 5% or when the latency exceeds 1 second.

Here's an example of how to create an alarm for high error rates using the AWS CLI:

```
aws cloudwatch put-metric-alarm \
  --alarm-name "HighErrorRate" \
  --alarm-description "Alarm when error rate exceeds 5%" \
  --metric-name "SystemErrors" \
  --namespace "BedrockAgentCore" \
  --statistic "Sum" \
  --dimensions "Name=Resource,Value=my-gateway-arn" \
  --period 300 \
  --evaluation-periods 1 \
  --threshold 5 \
  --comparison-operator "GreaterThanThreshold" \
  --alarm-actions "arn:aws:sns:us-west-2:123456789012:my-topic"
```

This alarm will trigger when the number of system errors exceeds 5 in a 5-minute period. When the alarm triggers, it will send a notification to the specified SNS topic.

Logging Gateway API calls with CloudTrail

Gateway is integrated with CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Gateway. CloudTrail captures all API calls for Gateway as events, including calls from the Gateway console and code calls to the Gateway APIs. Using the information collected by CloudTrail, you can determine the request that was made to Gateway, who made the request, when it was made, and additional details. There are two types of events: Management events and Data events:

Gateway Event Types

This section provides information about the types of events that Gateway logs to CloudTrail.

Gateway Management Events in CloudTrail

Every management event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail *Event history*. The CloudTrail *Event history* provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

Gateway logs management events for the following operations:

- `CreateGateway` - Creates a new gateway
- `UpdateGateway` - Updates an existing gateway
- `DeleteGateway` - Deletes a gateway
- `GetGateway` - Gets information about a gateway
- `ListGateways` - Lists all gateways
- `CreateGatewayTarget` - Creates a new target for a gateway
- `UpdateGatewayTarget` - Updates an existing gateway target
- `DeleteGatewayTarget` - Deletes a gateway target
- `GetGatewayTarget` - Gets information about a gateway target
- `ListGatewayTargets` - Lists all targets for a gateway

Gateway Data Events in CloudTrail

Data events provide information about the resource operations performed on or in a resource. These are also known as data plane operations. Data events are often high-volume activities. You must explicitly enable data events as they are not logged by default. The CloudTrail *Event history* doesn't record data events.

Additional charges apply for logging data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can enable logging data events for the Gateway resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations.

The following table lists the Gateway resource types for which you can enable data events:

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
Bedrock-AgentCore gateway	`AWS::BedrockAgentCore::Gateway`	InvokeMcp

Identity Information in Data Events

Gateway data events differ from standard AWS data events in how identity information is stored. Since the Data API follows the MCP protocol and uses JWT token-based authentication rather than SigV4, Gateway data events don't have standard AWS identity information. Instead, identity is captured by logging specific JWT claims including the "sub" claim.

> **Note**
>
> We recommend that you avoid using any personally identifiable information (PII) in this field. For example, you could use a GUID or a pairwise identifier, as suggested in the OIDC specification instead of PII data like email.

Error Information in Data Events

Gateway provides error information as part of the `responseElements` field rather than as top-level `errorCode` and `errorMessage` fields. If you're looking for specific error types such as AccessDenied events, parse through the `responseElements` field in the CloudTrail event.

Data Event Routing

Since Gateway uses JWT tokens for authentication rather than SigV4 credentials, data events are only routed to the resource owner account.

Enabling CloudTrail Data Event Logging for Gateway

You can use CloudTrail data events to get information about Gateway requests. To enable CloudTrail data events for Gateway, you must create a trail manually in CloudTrail backed by an Amazon S3 bucket.

> **Note**
>
> - Data event logging incurs additional charges. You must explicitly enable data events as they are not captured by default. Check to ensure that you have data events enabled for your account.
> - With a Gateway that is generating a high workload, you could quickly generate thousands of logs in a short amount of time. Be mindful of how long you choose to enable CloudTrail data events for a busy Gateway.

CloudTrail stores Gateway data event logs in an Amazon S3 bucket of your choosing. Consider using a bucket in a separate AWS account to better organize events from multiple resources into a central place for easier querying and analysis.

When you log data events for a trail in CloudTrail, you must use advanced event selectors to log data events for Gateway operations.

AWS CLI

To enable CloudTrail data events for Gateway using the AWS CLI, you can use the following command:

```
aws cloudtrail put-event-selectors \
  --trail-name brac-gateway-canary-trail-prod-us-east-1 \
  --region us-east-1 \
  --advanced-event-selectors '[
    {
      "Name": "GatewayDataEvents",
      "FieldSelectors": [
        {
          "Field": "eventCategory",
          "Equals": ["Data"]
        },
        {
          "Field": "resources.type",
          "Equals": ["AWS::BedrockAgentCore::Gateway"]
        }
      ]
    }
  ]'
```

AWS CDK

Here's an example of how to create a CloudTrail trail with Gateway data events using AWS CDK:

```
import { Construct } from 'constructs';
import { Trail, CfnTrail } from 'aws-cdk-lib/aws-cloudtrail';
import { Bucket } from 'aws-cdk-lib/aws-s3';
import { Effect, PolicyStatement, ServicePrincipal } from 'aws-cdk-lib/aws-iam';
import { RemovalPolicy } from 'aws-cdk-lib';
```

```
export interface DataEventTrailProps {
  /**
   * Whether to enable multi-region trail
   */
  isMultiRegionTrail?: boolean;

  /**
   * Whether to include global service events
   */
  includeGlobalServiceEvents?: boolean;

  /**
   * AWS region
   */
  region: string;

  /**
   * Environment account ID
   */
  account: string;
}

/**
 * Creates a CloudTrail trail configured to capture data events for Bedrock Agent
 Core Gateway
 */
export class BedrockAgentCoreDataEventTrail extends Construct {
  /**
   * The CloudTrail trail
   */
  public readonly trail: Trail;

  /**
   * The S3 bucket for CloudTrail logs
   */
  public readonly logsBucket: Bucket;

  constructor(scope: Construct, id: string, props: DataEventTrailProps) {
    super(scope, id);

    // Create S3 bucket for CloudTrail logs
    const bucketName = `brac-gateway-cloudtrail-logs-${props.account}-
${props.region}`;
```

```
    this.logsBucket = new Bucket(this, 'CloudTrailLogsBucket', {
      bucketName,
      removalPolicy: RemovalPolicy.RETAIN,
    });

    // Create trail name (suffixing region since regional trail)
    const trailName = `brac-gateway-trail-${props.region}`;

    // Add CloudTrail bucket policy
    this.logsBucket.addToResourcePolicy(
      new PolicyStatement({
        sid: 'AWSCloudTrailAclCheck',
        effect: Effect.ALLOW,
        principals: [new ServicePrincipal('cloudtrail.amazonaws.com')],
        actions: ['s3:GetBucketAcl'],
        resources: [this.logsBucket.bucketArn],
        conditions: {
          StringEquals: {
            'aws:SourceArn': `arn:aws:cloudtrail:${props.region}:
${props.account}:trail/${trailName}`,
          },
        },
      }),
    );

    this.logsBucket.addToResourcePolicy(
      new PolicyStatement({
        sid: 'AWSCloudTrailWrite',
        effect: Effect.ALLOW,
        principals: [new ServicePrincipal('cloudtrail.amazonaws.com')],
        actions: ['s3:PutObject'],
        resources: [this.logsBucket.arnForObjects(`AWSLogs/${props.account}/*`)],
        conditions: {
          StringEquals: {
            's3:x-amz-acl': 'bucket-owner-full-control',
            'aws:SourceArn': `arn:aws:cloudtrail:${props.region}:
${props.account}:trail/${trailName}`,
          },
        },
      }),
    );

    // Create CloudTrail trail
    this.trail = new Trail(this, 'GatewayDataEventTrail', {
      trailName,
      bucket: this.logsBucket,
      isMultiRegionTrail: props.isMultiRegionTrail ?? false,
      includeGlobalServiceEvents: props.includeGlobalServiceEvents ?? true,
      enableFileValidation: true,
    });

    // Add advanced event selectors for Bedrock Agent Core Gateway data events
    const cfnTrail = this.trail.node.defaultChild as CfnTrail;

    // Define the advanced event selectors
    const advancedEventSelectors = [
      {
        // Log Bedrock Agent Core Gateway Data Events only
        fieldSelectors: [
          {
            field: 'eventCategory',
            equalTo: ['Data'],
          },
          {
            field: 'resources.type',
            equalTo: ['AWS::BedrockAgentCore::Gateway'],
          },
        ],
      },
    ];

    // Clear any existing event selectors and set advanced event selectors
    cfnTrail.eventSelectors = undefined;
    cfnTrail.advancedEventSelectors = advancedEventSelectors;
  }
}
```

Understanding Gateway CloudTrail Events

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on.

Note

The contents of the requests and responses for data events are REDACTED, and the JWT claims have HTML entities sanitized for security purposes.

InvokeMcp Data Event With Authentication Error

The following example shows a CloudTrail log entry that demonstrates the `InvokeMcp` action with an authentication error:

```
{
      "eventVersion": "1.11",
      "userIdentity": {
        "type": "AWSAccount",
        "principalId": "",
        "accountId": "anonymous"
      },
      "eventTime": "2025-07-14T02:14:42Z",
      "eventSource": "bedrock-agentcore.amazonaws.com",
      "eventName": "InvokeMcp",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "34.XXX.XXX.206",
      "userAgent": "python-httpx/0.28.1",
      "requestParameters": {
        "body": {
          "id": 0,
          "method": "initialize",
          "params": {
            "clientInfo": {
              "name": "mcp",
              "version": "0.1.0"
            },
            "protocolVersion": "2025-06-18",
            "capabilities": {}
          },
          "jsonrpc": "2.0"
        }
      },
      "responseElements": {
        "body": {
          "jsonrpc": "2.0",
          "id": 0,
          "error": {
            "code": -32001,
            "message": "Invalid Bearer token"
          }
        },
        "contentType": "application/json",
        "statusCode": 401
      },
      "requestID": "1234abcd-12ab-34cd-56ef-1234567890ab",
      "eventID": "12345678-1234-5678-9abc-123456789012",
      "readOnly": false,
      "resources": [
        {
          "accountId": "XXXXXXXXXX",
          "type": "AWS::BedrockAgentCore::Gateway",
          "ARN": "arn:aws:bedrock-agentcore:us-west-2:XXXXXXXXXX:gateway/test-
openapi-gateway-b24f8c26-u9p3rjw8qw"
        }
      ],
      "eventType": "AwsApiCall",
      "managementEvent": false,
      "recipientAccountId": "XXXXXXXXXX",
      "sharedEventID": "12345678-xxxx-xxxx-xxxx-123456789012",
      "eventCategory": "Data",
      "tlsDetails": {
        "tlsVersion": "TLSv1.2",
        "cipherSuite": "ECDHE-RSA-AES128-GCM-SHA256",
        "clientProvidedHostHeader": "test-openapi-gateway-xxxxxxx-
u9p3rjw8qw.gateway.bedrock-agentcore.us-west-2.amazonaws.com"
      }
    }
```

Successful InvokeMcp Data Event

The following example shows a CloudTrail log entry for a successful `InvokeMcp` action:

```
    {
      "eventVersion": "1.11",
      "userIdentity": {
        "type": "AWSAccount",
        "principalId": "",
        "accountId": "anonymous"
      },
      "eventTime": "2025-07-14T02:14:42Z",
      "eventSource": "bedrock-agentcore.amazonaws.com",
      "eventName": "InvokeMcp",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "35.88.103.184",
      "userAgent": "python-httpx/0.28.1",
      "requestParameters": {
        "body": {
          "id": 1,
          "method": "tools/call",
          "params": {
            "name": "SmithyTarget___ListTables",
            "arguments": "REDACTED"
          },
          "jsonrpc": "2.0"
        }
      },
      "responseElements": {
        "body": {
          "jsonrpc": "2.0",
          "id": 1,
          "result": {
            "isError": false,
            "content": "REDACTED"
          }
        },
        "contentType": "application/json",
        "statusCode": 200
      },
      "additionalEventData": {
        "targetId": "0JTXXX4YMA",
        "jwt": {
          "headers": {
            "kid": "hGrcJwz5MX6hNeuL6jdXE4hjK7sT6oj+yN7kN+arRv4=",
            "alg": "RS256"
          },
          "claims": {
            "sub": "4ammgxxxxxxxxxxxm3b8c",
            "token_use": "access",
            "scope": "python-cognito-resource-server-id/write python-cognito-
resource-server-id/read",
      "auth_time": 1752459276,
      "iss": "https://cognito-idp.us-west-2.amazonaws.com/us-
west-2_Fxxxxhtq",
      "exp": 1752462876,
      "iat": 1752459276,
      "version": 2,
      "jti": "1234abcd-12ab-34cd-56ef-1234567890ab"
     },
     "type": "JWS"
    },
    "downstreamRequestIds": [
     "H3RDH6T03DG10996U0M2P1V1IFVV4KQNSO5AEMVJF66Q9ASUAAJG"
    ]
   },
   "requestID": "1234abcd-12ab-34cd-56ef-1234567890ab",
   "eventID": "12345678-1234-5678-9abc-123456789012",
   "readOnly": false,
   "resources": [
    {
     "accountId": "XXXXXXXXXX",
     "type": "AWS::BedrockAgentCore::Gateway",
     "ARN": "arn:aws:bedrock-agentcore:us-west-2:XXXXXXXXXX:gateway/test-
gateway-65129e91-mtzoadyihf"
    }
   ],
   "eventType": "AwsApiCall",
   "managementEvent": false,
   "recipientAccountId": "XXXXXXXXXX",
   "sharedEventID": "1234abcd-12ab-34cd-56ef-1234567890ab",
   "eventCategory": "Data",
   "tlsDetails": {
    "tlsVersion": "TLSv1.2",
    "cipherSuite": "ECDHE-RSA-AES128-GCM-SHA256",
    "clientProvidedHostHeader": "test-gateway-65129e91-xxxxxxxx.gateway.bedrock-
agentcore.us-west-2.amazonaws.com"
   }
  }
```

Management Event

The following example shows a CloudTrail log entry for a management event:

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROXXXXXXXXXXXXNRD7D:xxxxx",
    "arn": "arn:aws:sts::XXXXXXXXXXXX:assumed-role/HydraInvocationRole-xxxxxxxxx/
xxxx",
    "accountId": "XXXXXXXXXXXX",
    "accessKeyId": "xxxxxxxxx",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "xxxxxxxx",
        "arn": "arn:aws:iam::XXXXXXXXXXXX:role/HydraInvocationRole-xxx",
        "accountId": "XXXXXXXXXXXX",
        "userName": "HydraInvocationRole-xxxxx"
      },
      "attributes": {
        "creationDate": "2025-07-14T02:42:43Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "bedrock-agentcore.amazonaws.com"
  },
  "eventTime": "2025-07-14T02:47:38Z",
  "eventSource": "bedrock-agentcore.amazonaws.com",
  "eventName": "CreateGateway",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "bedrock-agentcore.amazonaws.com",
  "userAgent": "bedrock-agentcore.amazonaws.com",
  "requestParameters": {
    "roleArn": "arn:aws:iam::XXXXXXXXXXXX:role/PythonGenesisTestGatewayRole",
    "name": "***",
    "authorizerConfiguration": {
      "customJWTAuthorizer": {
        "allowedClients": [
          "xxxxxxxxx"
        ],
        "discoveryUrl": "https://cognito-idp.us-west-2.amazonaws.com/us-
west-2_xxxxx/.well-known/openid-configuration"
      }
    },
    "description": "***",
    "protocolType": "MCP",
  "authorizerType": "CUSTOM_JWT"
 },
 "responseElements": {
  "authorizerConfiguration": {
   "customJWTAuthorizer": {
    "allowedClients": [
     "xxxxxxxxxxxxxxx"
    ],
    "discoveryUrl": "https://cognito-idp.us-west-2.amazonaws.com/us-
west-2_xxxxxx/.well-known/openid-configuration"
   }
  },
  "description": "***",
  "protocolType": "MCP",
  "gatewayArn": "arn:aws:bedrock-agentcore:us-west-2:XXXXXXXXXXXX:gateway/test-
openapi-gateway-xxxxxxx-xxxxxx",
  "workloadIdentityDetails": {
   "workloadIdentityArn": "arn:aws:bedrock-agentcore:us-
west-2:XXXXXXXXXXXX:workload-identity-directory/default/workload-identity/test-
openapi-gateway-xxxxxx-xxxxx"
  },
  "createdAt": "2025-07-14T02:47:38.302834063Z",
  "gatewayUrl": "https://test-openapi-gateway-xxxxxxx-8fb4mo6pqx.gateway.bedrock-
agentcore.us-west-2.amazonaws.com/mcp",
  "roleArn": "arn:aws:iam::XXXXXXXXXXXX:role/PythonGenesisTestGatewayRole",
  "name": "***",
  "authorizerType": "CUSTOM_JWT",
  "gatewayId": "test-openapi-gateway-9c8f7109-8fb4mo6pqx",
  "status": "CREATING",
  "updatedAt": "2025-07-14T02:47:38.302845797Z"
 },
 "requestID": "0fb99b0b-a4d1-xxxx-8aee-c703adaa6bd9",
 "eventID": "b12bf859-xxxx-48d7-952a-d5c6ec00fb68",
 "readOnly": false,
 "resources": [
  {
   "accountId": "XXXXXXXXXXXX",
   "type": "AWS::BedrockAgentCore::Gateway",
   "ARN": "arn:aws:bedrock-agentcore:us-west-2:XXXXXXXXXXXX:gateway/test-openapi-
gateway-xxxxxxx-8fb4mo6pqx"
  }
 ],
 "eventType": "AwsApiCall",
 "managementEvent": true,
    "recipientAccountId": "XXXXXXXXXXXX",
    "eventCategory": "Management"
  }
```

Additional Resources

For more information about using CloudTrail with Gateway, see the following resources:

- AWS CloudTrail User Guide
- Creating a Trail for Your AWS Account
- AWS CloudTrail API Reference
- AWS CloudTrail CLI Reference

Advanced features and topics for Amazon Bedrock AgentCore Gateway

This chapter covers some advanced topics and additional information that can help supplement your knowledge of gateways and how you can use them effectively in your applications.

Topics

- (Optional) Encryption configuration
- Setting up custom domain names for Gateway endpoints
- Performance optimization

(Optional) Encryption configuration

When creating a gateway, you can optionally provide a KMS key for the service to encrypt data at-rest by using the `kmsKeyArn` request parameter. If this parameter is not provided, a default service-managed key will be used to encrypt the data.

Here is an example request using the AWS CLI:

```
aws bedrock-agentcore-control create-gateway \
  --name "MyGateway" \
  --protocol-type "MCP" \
  --role-arn "arn:aws:iam::123456789012:role/GatewayRole" \
  --kms-key-arn "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab" \
  --description "My Gateway with custom encryption" \
  --authorizer-type "CUSTOM_JWT" \
  --authorizer-configuration '{
    "customJWTAuthorizer": {
      "allowedAudience": ["myAudience"],
      "discoveryUrl": "https://example.com/.well-known/openid-configuration"
    }
  }'
```

Using a customer-managed KMS key gives you more control over the encryption and allows you to implement your own key rotation policies and access controls.

Data encryption with KMS (Optional)

By default, Gateway encrypts your data at rest using a service-managed AWS Key Management Service (AWS KMS) key. However, you can optionally provide your own customer managed key (CMK) for encrypting data at rest when creating a gateway.

Using a customer managed key gives you more control over the encryption process, including the ability to:

- Rotate the key on your own schedule
- Control access to the key through IAM policies
- Disable or delete the key when it's no longer needed
- Audit key usage through CloudWatch logs and AWS CloudTrail

> **Note**
>
> If you choose to use a customer managed key, you are responsible for managing the key and its permissions. If the key is disabled or deleted, or if Gateway loses permission to use the key, you will lose access to the encrypted data.

Console

To specify a customer managed key when creating a gateway in the console:

1. Open the Amazon Bedrock AgentCore console at Amazon Bedrock AgentCore and choose **Gateways**.
2. Choose **Create gateway**.
3. Fill in the required fields in the **Gateway details** section.
4. In the **Permissions** section:
   a. For **Service role**, choose an existing IAM role or create a new one.

      > **Note**
      >
      > The IAM role you use must have permissions to use the selected KMS key.

   b. For **KMS key**, choose one of the following options:
      - **Use AWS owned key** - The default option. Amazon Bedrock AgentCore manages the key for you.
      - **Choose from your AWS KMS keys** - Select an existing customer managed key from the dropdown list.
      - **Enter AWS KMS key ARN** - Enter the ARN of a customer managed key.
5. Complete the remaining steps to create your gateway.

CLI

To specify a customer managed key when creating a gateway using the AWS CLI:

```
aws bedrockagentcore create-gateway \
  --name "MyGateway" \
  --protocol-type "MCP" \
  --role-arn "arn:aws:iam::123456789012:role/GatewayExecutionRole" \
  --kms-key-arn "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab" \
  --description "Gateway with customer managed encryption key" \
  --authorizer-type "CUSTOM_JWT" \
  --authorizer-configuration '{"customJWTAuthorizer":{"allowedAudience":
["api.example.com"],"discoveryUrl":"https://auth.example.com/.well-known/openid-
configuration"}}'
```

The key policy for the customer managed key must include permissions for Gateway to use the key. Here's an example key policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "ManageKMSKey",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:root"
      },
      "Action": "kms:*",
      "Resource": "*"
    },
    {
      "Sid": "AllowAgentCoreDataplaneServiceToUseKMSKey",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/GatewayServiceRole"
      },
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": "*",
      "Condition": {
          "StringEquals": {
              "kms:ViaService": "bedrock-agentcore.us-west-2.amazonaws.com"
          }
      }
    }
  ]
}
```

Setting up custom domain names for Gateway endpoints

By default, Gateway endpoints are provided with an AWS-managed domain name in the format `<gateway-id>.gateway.bedrock-agentcore.<region>.amazonaws.com`. For production environments or to create a more user-friendly experience, you may want to use a custom domain name for your gateway endpoint. This section guides you through setting up a custom domain name using Amazon CloudFront as a reverse proxy.

Prerequisites

Before you begin, ensure you have:

- A working Gateway endpoint
- DNS delegation (if your Route 53 domain needs to be publicly reachable)
- AWS CDK installed and configured (if following the CDK approach)
- Appropriate IAM permissions to create and manage CloudFront distributions, Route 53 hosted zones, and ACM certificates

Solution overview

The solution involves the following components:

- **Route 53 Hosted Zone**: Manages DNS records for your custom domain
- **ACM Certificate**: Provides SSL/TLS encryption for your custom domain
- **CloudFront Distribution**: Acts as a reverse proxy, forwarding requests from your custom domain to the Gateway endpoint
- **Route 53 A Record**: Maps your custom domain to the CloudFront distribution

The following steps will guide you through setting up these components using AWS CDK.

Implementation steps

Step 1: Create a Route 53 hosted zone

First, create a Route 53 hosted zone for your custom domain:

```
import { RemovalPolicy } from 'aws-cdk-lib';
import { PublicHostedZone } from 'aws-cdk-lib/aws-route53';

const domainName = 'my.example.com';

const hostedZone = new PublicHostedZone(this, 'HostedZone', {
    zoneName: domainName,
});
this.hostedZone.applyRemovalPolicy(RemovalPolicy.RETAIN);
```

Note

We apply a removal policy of `RETAIN` to prevent accidental deletion of the hosted zone during stack updates or deletion.

Step 2: Create a DNS-validated certificate

Next, create an SSL/TLS certificate for your custom domain using AWS Certificate Manager (ACM) with DNS validation:

```
import { RemovalPolicy } from 'aws-cdk-lib';
import { Certificate, CertificateValidation } from 'aws-cdk-lib/aws-
certificatemanager';

const certificate = new Certificate(this, 'SSLCertificate', {
    domainName: domainName, // route53 hosted zone domain name from step 1
    validation: CertificateValidation.fromDns(hostedZone), // route53 hosted zone from
 step 1
});
this.certificate.applyRemovalPolicy(RemovalPolicy.RETAIN);
```

DNS validation automatically creates the necessary validation records in your Route 53 hosted zone.

Step 3: Create a CloudFront distribution

Create a CloudFront distribution to act as a reverse proxy for your Gateway endpoint:

```
import {
  AllowedMethods,
  CachePolicy,
  Distribution,
  OriginProtocolPolicy,
  ViewerProtocolPolicy
} from 'aws-cdk-lib/aws-cloudfront';
import { HttpOrigin } from 'aws-cdk-lib/aws-cloudfront-origins';

const bedrockAgentCoreGatewayHostName = '<mymcpserver>.gateway.bedrock-
agentcore.<region>.amazonaws.com'
const bedrockAgentCoreGatewayPath = '/mcp' // can also be left undefined, depending on
 your requirement

const distribution = new Distribution(this, 'Distribution', {
    defaultBehavior: {
        origin: new HttpOrigin(bedrockAgentCoreGatewayHostName, {
            protocolPolicy: OriginProtocolPolicy.HTTPS_ONLY,
            originPath: bedrockAgentCoreGatewayPath,
        }),
        viewerProtocolPolicy: ViewerProtocolPolicy.HTTPS_ONLY,
        cachePolicy: CachePolicy.CACHING_DISABLED, // important since caching is
 enabled by default and hence is not suitable for a reverse proxy
        allowedMethods: AllowedMethods.ALLOW_ALL,
    },
    domainNames: [domainName], // route53 hosted zone domain name from step 1
    certificate: certificate, // ssl certificate for the route53 domain from step 2
});
```

> ⚠ **Important**
>
> Set `cachePolicy: CachePolicy.CACHING_DISABLED` to ensure that CloudFront doesn't cache responses from your Gateway endpoint, which is important for dynamic API interactions.

Replace `<mymcpserver>` with your gateway ID and `<region>` with your AWS Region (e.g., `us-east-1`).

Step 4: Create a Route 53 A record

Create a Route 53 A record that points your custom domain to the CloudFront distribution:

```
import { ARecord, RecordTarget } from 'aws-cdk-lib/aws-route53';
import { CloudFrontTarget } from 'aws-cdk-lib/aws-route53-targets';

const aRecord = new ARecord(this, 'AliasRecord', {
    zone: hostedZone, // route53 hosted zone from step 1
    recordName: domainName, // route53 hosted zone domain name from step 1
    target: RecordTarget.fromAlias(new CloudFrontTarget(distribution)), // cloufront
 distribution from from step 3
});
```

This creates an alias record that maps your custom domain to the CloudFront distribution.

Step 5: Deploy your infrastructure

Deploy your CDK stack to create the resources:

```
cdk deploy
```

The deployment process may take some time, especially for the certificate validation and CloudFront distribution creation.

Testing your custom domain

After deploying your infrastructure, verify that your custom domain is properly configured:

Verify DNS resolution

Use the `dig` command to verify that your custom domain resolves to the CloudFront distribution:

```
dig my.example.com
```

The output should show that your domain resolves to CloudFront's IP addresses.

Verify SSL certificate

Use `curl` to verify that the SSL certificate is properly configured:

```
curl -v https://my.example.com
```

The output should show a successful SSL handshake with no certificate errors.

Configuring MCP clients

Once your custom domain is set up and verified, you can configure your MCP clients to use it:

Cursor configuration

For Cursor, update your configuration file:

```
{
  "mcpServers": {
    "my-mcp-server": {
      "url": "https://my.example.com"
    }
  }
}
```

Other MCP clients

For MCP clients that don't natively support streamable HTTP:

```
{
  "mcpServers": {
    "my-mcp-server": {
      "command": "/path/to/uvx",
        "args": [
            "mcp-proxy",
            "--transport",
            "streamablehttp",
            "https://my.example.com"
        ]
    }
  }
}
```

Additional considerations

Cost implications

Using CloudFront as a reverse proxy incurs additional costs for data transfer and request handling. Review the CloudFront pricing model to understand the cost implications for your specific use case.

Security considerations

Consider implementing additional security measures such as:

- WAF rules to protect your endpoint from common web exploits
- Geo-restrictions to limit access to specific geographic regions
- Custom headers or request signing to add an extra layer of authentication

Monitoring and logging

Enable CloudFront access logs and configure CloudWatch alarms to monitor the health and performance of your custom domain setup.

Certificate renewal

ACM certificates issued through DNS validation are automatically renewed as long as the DNS records remain in place. Ensure that you don't delete the validation records.

Troubleshooting

DNS resolution issues

If your custom domain doesn't resolve correctly:

- Verify that the A record is correctly configured in your Route 53 hosted zone
- Check that your domain's name servers are correctly set at your domain registrar
- Allow time for DNS propagation (up to 48 hours in some cases)

SSL certificate issues

If you encounter SSL certificate errors:

- Verify that the certificate is issued and active in the ACM console
- Check that the certificate is correctly associated with your CloudFront distribution
- Ensure that the certificate covers the exact domain name you're using

Gateway connectivity issues

If your custom domain doesn't connect to your gateway:

- Verify that the origin domain and path in your CloudFront distribution are correct
- Check that your gateway endpoint is accessible directly
- Review CloudFront distribution logs for any errors

Conclusion

Setting up a custom domain name for your Gateway endpoint enhances the professional appearance of your application and provides flexibility in managing your API endpoints. By following the steps outlined in this guide, you can create a secure and reliable custom domain configuration using CloudFront as a reverse proxy.

For more information about Gateway features and capabilities, see *AgentCore Gateway : Securely connect to tools and resources*.

Performance optimization

To optimize the performance of your Gateway implementations, consider the following best practices:

Minimize tool latency

The overall latency of your gateway is largely determined by the latency of the underlying tools. To minimize latency:

- Use Lambda functions in the same region as your gateway
- Optimize your Lambda functions for fast cold starts
- Use provisioned concurrency for Lambda functions that require low latency
- Ensure that REST APIs have low latency and high availability

Use efficient tool schemas

Well-designed tool schemas can improve the performance of your gateway:

- Keep schemas as simple as possible
- Use appropriate data types for parameters
- Include clear descriptions for parameters to help agents use the tools correctly
- Use required fields to ensure that agents provide necessary parameters

Enable semantic search

Semantic search helps agents find the right tools for their tasks, improving the overall performance of your agent-gateway interactions. Enable semantic search when creating your gateway:

```
from bedrockagentcoresdk.gateway import GatewayClient

# Initialize the Gateway client
gateway_client = GatewayClient(region_name="us-west-2")

# Create a gateway with semantic search enabled
gateway = gateway_client.create_gateway(
    name="semantic-search-gateway",
    description="A gateway with semantic search enabled",
    protocol_configuration={
        "mcp": {
            "search_type": "SEMANTIC"
        }
    }
)
```

Monitor and optimize

Use the observability features described in the previous section to monitor the performance of your gateway and identify opportunities for optimization:

- Set up CloudWatch alarms for key metrics
- Analyze logs to identify patterns and issues
- Regularly review performance metrics and make adjustments as needed

Observe your agent applications on Amazon Bedrock AgentCore Observability

With AgentCore, you can trace, debug, and monitor AI agents' performance in production environments.

AgentCore Observability helps you trace, debug, and monitor agent performance in production environments. It offers detailed visualizations of each step in the agent workflow, enabling you to inspect an agent's execution path, audit intermediate outputs, and debug performance bottlenecks and failures.

AgentCore Observability gives you real-time visibility into agent operational performance through access to dashboards powered by Amazon CloudWatch and telemetry for key metrics such as session count, latency, duration, token usage, and error rates. Rich metadata tagging and filtering simplify issue investigation and quality maintenance at scale. AgentCore emits telemetry data in standardized OpenTelemetry (OTEL)-compatible format, enabling you to easily integrate it with your existing monitoring and observability stack.

By default, AgentCore outputs a set of key built-in metrics for agents, gateway resources, and memory resources. For memory resources, AgentCore also outputs spans and log data if you enable it. You can also instrument your agent code to provide additional span and trace data and custom metrics and logs. See the section called “Add observability to your agents” to learn more.

All of the metrics, spans, and logs output by AgentCore are stored in Amazon CloudWatch, and can be viewed in the CloudWatch console or downloaded from CloudWatch using the AWS CLI or one of the AWS SDKs.

In addition to the raw data stored in CloudWatch Logs, for agent runtime data only, the CloudWatch console provides an observability dashboard containing trace visualizations, graphs for custom span metrics, error breakdowns, and more. To learn more about viewing your agents' observability data, see the section called “View metrics for your agents”

Topics

Add observability to your Amazon Bedrock AgentCore resources

Amazon Bedrock AgentCore provides a number of built-in metrics to monitor the performance of resources for the AgentCore runtime, memory, gateway, and built-in tool resource types. This default data is available in Amazon CloudWatch. To view the full range of observability data in the CloudWatch console, or to output custom runtime metrics for agents, you need to instrument your code using the AWS Distro for Open Telemetry (ADOT) SDK.

To view the observability dashboard in CloudWatch, open the Amazon CloudWatch GenAi Observability page.

See the following sections to learn more about configuring your resources to view observability metrics in the CloudWatch console generative AI observability page and in CloudWatch Logs.

> **Tip**
>
> Use of the ADOT SDK to output custom metrics is also supported for agents running outside the AgentCore runtime. To learn how to enable observability for these agents, see the section called “Configure Observability for agents hosted outside of the AgentCore runtime”.

Enabling AgentCore runtime observability

To view metrics, spans, and traces generated by the AgentCore service, you first need to complete a one-time setup to turn on Amazon CloudWatch Transaction Search. To view service-provided spans for memory resources, you also need to enable tracing when you create a memory. See the section called “Enable observability for AgentCore memory, gateway, and built-in tool resources” to learn more.

The following sections describe how to perform these setup actions and to enable observability in your agent code.

Enabling CloudWatch Transaction Search

You can enable CloudWatch Transaction Search either by using the CloudWatch console, or by using an API through the AWS Command Line Interface (AWS CLI) or one of the AWS SDKs.

Use one of the following procedures to enable Transaction Search.

CloudWatch console

To enable CloudWatch Transaction Search in the CloudWatch console

1. Open the CloudWatch console.
2. In the navigation pane, expand **Application Signals (APM)** and choose **Transaction search**.
3. Choose **Enable Transaction Search**.
4. Select the checkbox to ingest spans as structured logs.
5. (Optional) Change the percentage of spans you want to be indexed as trace summaries by entering a value under **X-Ray trace indexing**. By default, 1% of spans are indexed as trace summaries for free, but you can alter the percentage to generate more trace summaries for end-to-end transaction analysis.
6. Choose **Save**.

API

To enable CloudWatch Transaction Search using an API

1. When using the AWS CLI or an AWS SDK to enable Transaction Search, first configure the necessary permissions to ingest spans in CloudWatch Logs by adding a resource-based policy with PutResourcePolicy.

   The following AWS CLI command adds a resource policy that gives AWS X-Ray permissions to send traces to CloudWatch Logs.

```
aws logs put-resource-policy --policy-name MyResourcePolicy --policy-
document '{ "Version": "2012-10-17",        "Statement": [ { "Sid":
 "TransactionSearchXRayAccess", "Effect": "Allow", "Principal": { "Service":
 "xray.amazonaws.com" }, "Action": "logs:PutLogEvents", "Resource":
 [ "arn:partition:logs:region:account-id:log-group:aws/spans:*",
 "arn:partition:logs:region:account-id:log-group:/aws/application-
signals/data:*" ], "Condition": { "ArnLike": { "aws:SourceArn":
 "arn:partition:logs:region:account-id:*" }, "StringEquals":
 { "aws:SourceAccount": "account-id" } } } ]}'
```

For clarity, the inline JSON policy in this command is shown expanded in the following example:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "TransactionSearchXRayAccess",
            "Effect": "Allow",
            "Principal": {
                "Service": "xray.amazonaws.com"
            },
            "Action": "logs:PutLogEvents",
            "Resource": [
                "arn:aws:logs:us-east-1:123456789012:log-group:aws/spans:*",
                "arn:aws:logs:us-east-1:123456789012:log-group:/aws/
application-signals/data:*"
            ],
            "Condition": {
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:xray:us-east-1:123456789012:*"
                },
                "StringEquals": {
                    "aws:SourceAccount": "123456789012"
                }
            }
        }
    ]
}
```

2. Configure the destination of your trace segments using UpdateTraceSegmentDestination.

   To use the AWS CLI, run the following command.

   ```
   aws xray update-trace-segment-destination --destination CloudWatchLogs
   ```

3. (Optional) Configure your desired sampling percentage using UpdateIndexingRule.

   To use the AWS CLI, run the following command.

```
aws xray update-indexing-rule --name "Default" --rule '{"Probabilistic":
 {"DesiredSamplingPercentage": number}}'
```

Enabling observability in agent code for AgentCore-hosted agents

In addition to the service-generated metrics, with AgentCore you can also gather span and trace data as well as custom metrics emitted from your agent code.

When you use agent frameworks like Strands, LangChain, or CrewAI with supported third-party instrumentation libraries, the framework itself comes with built in support for OTEL and GenAI semantic conventions, and it can also be instrumented with an auto-instrumentation package such as `opentelemetry-instrument-langchain`. It is also possible to send Generative AI semantic conventions telemetry and spans by defining a custom tracer. AgentCore supports use of the following instrumentation libraries in your agent framework:

- OpenInference
- Openllmetry
- OpenLit
- Traceloop

To view this data in the CloudWatch console generative AI observability page and in Amazon CloudWatch, you need to add the AWS Distro for Open Telemetry (ADOT) SDK to your agent code.

> **Note**
>
> With AgentCore, you can also view metrics for agents that aren't running in the AgentCore runtime. Additional setup steps are required to configure telemetry outputs for non-AgentCore agents. See the instructions in the section called “Configure Observability for agents hosted outside of the AgentCore runtime” to learn more.

To add ADOT support and enable AgentCore observability, follow the steps in the following procedure.

Add observability to your AgentCore agent

1. Ensure that your framework is configured to emit traces. For example, in the Strands framework, the tracer object must be configured to instruct Strands to emit Open Telemetry (OTEL) logs.
2. Add the ADOT SDK and boto3 to your agent's dependencies. For Python, add the following to your `requirements.txt` file:

   ```
   aws-opentelemetry-distro>=0.10.0
   boto3
   ```

   Alternatively, you can install the dependencies directly:

   ```
   pip install aws-opentelemetry-distro>=0.10.0 boto3
   ```

3. Execute your agent code using the OpenTelemetry auto-instrumentation command:

   ```
   opentelemetry-instrument python my_agent.py
   ```

   This auto-instrumentation approach automatically adds the SDK to the Python path. You may already be using this approach as part of your standard OpenTelemetry implementation.

   For containerized environment (such as docker) add the following command:

   ```
   CMD ["opentelemetry-instrument", "python", "main.py"]
   ```

   When using ADOT, in order to propagate session id correctly, define the `X-Amzn-Bedrock-AgentCore-Runtime-Session-Id` in the request header. ADOT then sets the session_id correctly in the downstream headers.

   To propoagate a trace ID, invoke the AgentCore runtime with the parameter `traceId=<traceId>` set.

   You can also invoke your agent with additional headers for additional observability options. See the section called “Enhanced AgentCore observability with custom headers” to learn more.

Configure Observability for agents hosted outside of the AgentCore runtime

To enable observability for agents hosted outside of the AgentCore runtime, first follow the steps in the previous sections to enable CloudWatch Transaction Search and add the ADOT SDK to your code.

For agents running outside of the AgentCore runtime, you also need to create an agent log-group which you include in your environment variables.

Configure your AWS environment variables, and then set your Open Telemetry environment variables as shown in the following.

AWS environment variables

```
AWS_ACCOUNT_ID=<account id>
AWS_DEFAULT_REGION=<default region>
AWS_REGION=<region>
AWS_ACCESS_KEY_ID=<access key id>
AWS_SECRET_ACCESS_KEY=<secret key>
```

OTEL environment variables

```
AGENT_OBSERVABILITY_ENABLED=true
OTEL_PYTHON_DISTRO=aws_distro
OTEL_PYTHON_CONFIGURATOR=aws_configurator # required for ADOT Python only
OTEL_RESOURCE_ATTRIBUTES=service.name=<agent-name>,aws.log.group.names=/aws/bedrock-
agentcore/runtimes/<agent-id>,cloud.resource_id=<AgentEndpointArn:AgentEndpointName> #
 endpoint is optional
OTEL_EXPORTER_OTLP_LOGS_HEADERS=x-aws-log-group=/aws/bedrock-agentcore/runtimes/<agent-
id>,x-aws-log-stream=runtime-logs,x-aws-metric-namespace=bedrock-agentcore
OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf
OTEL_TRACES_EXPORTER=otlp
```

Replace `<agent-name>` with your agent's name and `<agent-id>` with a unique identifier for your agent.

Session ID support

To propagate session ID, you need to invoke using session identifier in the OTEL baggage:

```
from opentelemetry import baggage

ctx = baggage.set_baggage("session.id", session_id) # Set the session.id in baggage
attach(ctx) # Attach the context to make it active token
```

Enable observability for AgentCore memory, gateway, and built-in tool resources

When you create an AgentCore runtime resource (agent), by default, AgentCore runtime creates a CloudWatch log group for the service-provided logs. However, for memory, gateway, and built-in tool resources, AgentCore doesn't configure log destinations for you automatically.

For memory and gateway resources, you can configure log destinations either in the console or by using an AWS SDK. If you use the console to configure a CloudWatch Logs destination, the default log group name for memory and gateway resources has the form `/aws/vendedlogs/bedrock-agentcore/{resource-type}/APPLICATION_LOGS/{resource-id}`, where `{resource-type}` is either `memory` or `gateway`.

For memory and gateway logs, you can also configure log destinations in Amazon S3 logs or Firehose stream logs using the AgentCore console. To learn more about storing logs in Amazon S3 or Firehose, see Uploading, downloading, and working with objects in Amazon S3 and Creating an Amazon Data Firehose delivery stream.

To learn more about the log data output by AgentCore for memory and gateway resources see Provided log data (memory) or Provided log data (gateway).

For built-in tool resources, the AgentCore service doesn't provide logs by default, but you can output your own logs from your code. If you supply your own log outputs, you need to manually configure log destinations to store this data.

To see what observability data AgentCore provides by default for each resource type, see the section called “AgentCore provided metrics”.

Configure log destinations using the console

To configure log destinations for memory or gateway logs in the AgentCore console, use the following procedures.

Memory

To configure log delivery for memory resources (console)

1. Open the Memory page in the AgentCore console.
2. In the **Memory** pane, select the memory you want to configure a log destination for.
3. Scroll down to the **Log delivery** pane and choose **Add**.
4. From the dropdown list, select the type of log destination you want to add (CloudWatch Logs group, Amazon S3 bucket, or Amazon Data Firehose).
5. For **Log type**, select **APPLICATION_LOGS**.
6. For Amazon S3 and Firehose destinations, enter a **Delivery destination ARN**. For CloudWatch Logs, the **Destination log group** is already populated with a default value.
7. (Optional) For CloudWatch Logs destinations, to change the default log group, enter a new log group name or select an existing log group under **Destination log group**.
8. (Optional) To change the fields that are captured in each log record or the logs' output format, expand **Additional settings -** ***optional***, and modify the **Field selection**, **Output format**, and **Field delimiter** to your desired configuration.
9. Choose **Add**.

Gateway

To configure log delivery for gateway resources (console)

1. Open the Gateways page in the AgentCore console.
2. In the **Gateways** pane, select the gateway you want to configure a log destination for.
3. Scroll down to the **Log delivery** pane and choose **Add**.
4. From the dropdown list, select the type of log destination you want to add (CloudWatch Logs group, Amazon S3 bucket, or Amazon Data Firehose).
5. For Amazon S3 and Firehose destinations, enter a **Delivery destination ARN**. For CloudWatch Logs, the **Destination log group** is already populated with a default value.
6. (Optional) For CloudWatch Logs destinations, to change the default log group, enter a new log group name or select an existing log group under **Destination log group**.
7. (Optional) To change the fields that are captured in each log record or the logs' output format, expand **Additional settings -** ***optional***, and modify the **Field selection**, **Output format**, and **Field delimiter** to your desired configuration.

8. Choose **Add**.

Configure CloudWatch resources using an AWS SDK

To configure a delivery source for logs and traces (SDK)

- Run the following Python code to configure CloudWatch for your memory, gateway, and built-in tool resources. Note that delivery sources and destinations for tracing are only applicable for memory resources.

```
import boto3

def enable_observability_for_resource(resource_arn, resource_id, account_id,
 region='us-east-1'):
    """
    Enable observability for a Bedrock AgentCore resource (e.g., Memory Store)
    """
    logs_client = boto3.client('logs', region_name=region)

    # Step 0: Create new log group for vended log delivery
    log_group_name = f'/aws/vendedlogs/bedrock-agentcore/{resource_id}'
    logs_client.create_log_group(logGroupName=log_group_name)
    log_group_arn = f'arn:aws:logs:{region}:{account_id}:log-group:{log_group_name}'

    # Step 1: Create delivery source for logs
    logs_source_response = logs_client.put_delivery_source(
        name=f"{resource_id}-logs-source",
        logType="APPLICATION_LOGS",
        resourceArn=resource_arn
    )

    # Step 2: Create delivery source for traces
    traces_source_response = logs_client.put_delivery_source(
        name=f"{resource_id}-traces-source",
        logType="TRACES",
        resourceArn=resource_arn
    )

    # Step 3: Create delivery destinations
    logs_destination_response = logs_client.put_delivery_destination(
        name=f"{resource_id}-logs-destination",
        deliveryDestinationType='CWL',
        deliveryDestinationConfiguration={
            'destinationResourceArn': log_group_arn,
        }
    )

    # Traces required for memory only
    traces_destination_response = logs_client.put_delivery_destination(
        name=f"{resource_id}-traces-destination",
        deliveryDestinationType='XRAY'
    )

    # Step 4: Create deliveries (connect sources to destinations)
    logs_delivery = logs_client.create_delivery(
        deliverySourceName=logs_source_response['deliverySource']['name'],
        deliveryDestinationArn=logs_destination_response['deliveryDestination']['arn']
    )

    # Traces required for memory only
    traces_delivery = logs_client.create_delivery(
        deliverySourceName=traces_source_response['deliverySource']['name'],
        deliveryDestinationArn=traces_destination_response['deliveryDestination']
['arn']
    )

    print(f"Observability enabled for {resource_id}")
    return {
        'logs_delivery_id': logs_delivery['id'],
        'traces_delivery_id': traces_delivery['id']
    }

# Usage example
resource_arn = "arn:aws:bedrock-agentcore:us-east-1:123456789012:memory/my-memory-id"
resource_id = "my-memory-id"
account_id = "123456789012"

delivery_ids = enable_observability_for_resource(resource_arn, resource_id, account_id)
```

Enhanced AgentCore observability with custom headers

You can invoke your agent with additional HTTP headers to provide enhanced observability options. The following example shows invocations including optional additional header requests for agents hosted in the AgentCore runtime.

Example Boto3 invocation

```
def invoke_agent(agent_id, payload, session_id=None):
    client = boto3.client("bedrock-agentcore", region="us-west-2")
    response = client.invoke_agent_runtime(
        agentRuntimeArn="arn:aws:bedrock-agentcore:us-west-2:111122223333:runtime/
test_agent_boto2-nIg2xk3VSR",
        runtimeSessionId="12345678-1234-5678-9abc-123456789012",
        payload='{"query": "Plan a weekend in Seattle"}',
    )
```

You can include the following optional headers when invoking your agent to enhance observability and tracing capabilities:

Optional request headers for observability

Header	Description	Sample Value	Technical Explanation
X-Amzn-Trace-Id	Trace ID for request tracking (X-Ray Format)	Root=1-5759e988-bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1	Used for distributed tracing across AWS services. Contains root ID (request origin), parent ID (previous service), and sampling decision for tracing. Sampling=1 means 100% sampling. Parent is X-Ray Trace format as well. OTEL will auto-generate trace IDs if not supplied.
traceparent	W3C standard tracing header	00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01	W3C format that includes version, trace ID, parent ID, and flags. Required for cross-service trace correlation when using modern tracing systems.
X-Amzn-Bedrock-AgentCore-Ru	AgentCore session identifier	a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa	Identifies a user session within the AgentCore system. Helps with session-based analytics and troubleshooting.

Header	Description	Sample Value	Technical Explanation
ntime-Session-Id			
mcp-session-id	MCP session identifier	mcp-a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa	Identifies a session in the Managed Cloud Platform. Enables tracing of operations across the MCP ecosystem.
tracestate	Additional tracing state information	congo=t61rcWkgMzE,rojo=00f067aa0ba902b7	Vendor-specific tracing information. Conveys additional context for tracing systems beyond what's in traceparent.
baggage	Context propagation for distributed tracing	userId=alice,serverRegion=us-east-1	Key-value pairs that propagate user-defined properties across service boundaries for contextual logging and analysis.

Observability best practices

Consider the following best practices when implementing observability for agents in AgentCore:

- Use consistent session IDs - When possible, reuse the same session ID for related requests to maintain context across interactions.
- Implement distributed tracing - Use the provided headers to enable end-to-end tracing across your application components.
- Add custom attributes - Enhance your traces and metrics with custom attributes that provide additional context for troubleshooting and analysis.
- Monitor resource usage - Pay attention to memory usage metrics to optimize your agent's performance.
- Set up alerts - Configure CloudWatch alarms to help notify you of potential issues before they impact your users.

Using other observability platforms

To integrate agents hosted in the AgentCore runtime with other observability platforms to capture and view telemetry outputs, set the following environment variable:

```
DISABLE_ADOT_OBSERVABILITY=true
```

Setting this variable to `true` unsets the AgentCore runtime's default ADOT environment variables, ensuring that none of the default ADOT configurations are set.

Understand observability for agentic resources in AgentCore

This section defines the concepts of sessions, traces and spans as they relate to monitoring and observability of agents.

Topics

- Sessions
- Traces
- Spans
- Relationship Between Sessions, Traces, and Spans

Sessions

A session represents a complete interaction context between a user and an agent. Sessions encapsulate the entire conversation or interaction flow, maintaining state and context across multiple exchanges. Each session has a unique identifier and captures the full lifecycle of user engagement with the agent, from initialization to termination.

Sessions provide the following capabilities for agents:

- Context persistence across multiple interactions within the same conversation
- State management for maintaining user-specific information
- Conversation history tracking for contextual understanding
- Resource allocation and management for the duration of the interaction
- Isolation between different user interactions with the same agent

From an observability perspective, sessions provide a high-level view of user engagement patterns, allowing you to monitor agent performance across metrics, traces, and spans and to understand how users interact with your agents over time and across different use cases.

By default, AgentCore provides a set of observability metrics at the session level for agents that are running in the AgentCore runtime. You can view the runtime metrics in the Amazon CloudWatch console on the generative AI observability page. This page offers a variety of graphs and visualizations to help you interpret your agents' data. AgentCore also outputs a default set of metrics for memory resources, gateway resources, and built-in tools. All of these metrics can be viewed in CloudWatch. In addition to the provided metrics, logs and spans are provided by default for memory resources, and by instrumenting your agent code, you can capture custom metrics, logs, and spans for your agent which can also be viewed on the CloudWatch generative AI observability page. See the following sections and the section called “View metrics for your agents” to learn more.

Traces

A trace represents a detailed record of a single request-response cycle beginning from with an agent invocation and may include additional calls to other agents. Traces capture the complete execution path of a request, including all internal processing steps, external service calls, decision points, and resource utilization. Each trace is associated with a specific session and provides granular visibility into the agent's behavior for a particular interaction.

Traces include the following components for agents:

- Request details including timestamps, input parameters, and context
- Processing steps showing the sequence of operations performed
- Tool invocations with input/output parameters and execution times
- Resource utilization metrics such as processing time
- Error information including exception details and recovery attempts
- Response generation details and final output

From an observability perspective, traces provide deep insights into the internal workings of your agents, allowing you to troubleshoot issues, optimize performance, and understand behavior patterns. By analyzing trace data, you can identify bottlenecks, detect anomalies, and verify that your agent is functioning as expected across different scenarios and inputs.

To gather trace data, you need to instrument your agent code using the AWS Distro for Open Telemetry (ADOT). See the section called “Enabling observability in agent code for AgentCore-hosted agents” and the section called “Configure Observability for agents hosted outside of the AgentCore runtime” to learn more.

Spans

A span represents a discrete, measurable unit of work within an agent's execution flow. Spans capture fine-grained operations that occur during request processing, providing detailed visibility into the internal components and steps that make up a complete trace. Each span has a defined start and end time, creating a precise timeline of agent activities and their durations.

Spans include the following essential attributes for agent observability:

- Operation name identifying the specific function or process being executed
- Timestamps marking the exact start and end times of the operation
- Parent-child relationships showing how operations nest within larger processes
- Tags and attributes providing contextual metadata about the operation
- Events marking significant occurrences within the span's lifetime
- Status information indicating success, failure, or other completion states
- Resource utilization metrics specific to the operation

Spans form a hierarchical structure within traces, with parent spans encompassing child spans that represent more granular operations. For example, a high-level "process user query" span might contain child spans for "parse input," "retrieve context," "generate response," and "format output." This hierarchical organization creates a detailed execution tree that reveals the complete flow of operations within the agent.

By default, AgentCore outputs a set of span data for memory resources only. This data can be viewed in CloudWatch Logs and CloudWatch Application signals. To record span data for your agents or gateway resources, you need to instrument your agent. See the section called “Enabling observability in agent code for AgentCore-hosted agents” and the section called “Configure Observability for agents hosted outside of the AgentCore runtime” to learn more.

Relationship Between Sessions, Traces, and Spans

Sessions, traces, and spans form a three-tiered hierarchical relationship in the observability framework for agents. A session contains multiple traces, with each trace representing a discrete interaction within the broader context of the session. Each trace, in turn, contains multiple spans that capture the fine-grained operations and steps within that interaction. This hierarchical structure allows you to analyze agent behavior at different levels of granularity, from high-level session patterns to mid-level interaction flows to detailed execution paths for specific operations.

The relationship between these three observability components can be visualized as:

- Sessions (highest level) - Represent complete user conversations or interaction contexts
- Traces (middle level) - Represent individual request-response cycles within a session
- Spans (lowest level) - Represent specific operations or steps within a trace

This multi-tiered relationship enables several important observability capabilities:

- Contextual analysis of individual interactions within their broader conversation flow
- Correlation of related requests across a user's interaction journey
- Progressive troubleshooting from session-level anomalies to trace-level patterns to span-level root causes
- Comprehensive performance profiling across different temporal and functional dimensions
- Holistic understanding of agent behavior patterns and evolution throughout a conversation
- Precise identification of performance bottlenecks at the operation level through span analysis

While traces provide visibility into complete request-response cycles, spans offer deeper insights into the internal workings of those cycles. Spans reveal exactly which operations consume the most time, where errors originate, and how different components interact within a single trace. This granularity is particularly valuable when troubleshooting complex issues or optimizing performance in sophisticated agent implementations.

By leveraging session, trace, and span data in your observability strategy, you can gain comprehensive insights into your agent's behavior, performance, and effectiveness at multiple levels of detail. This multi-layered approach to observability supports continuous improvement, robust troubleshooting, and informed optimization of your agent implementations, from high-level conversation patterns down to individual operation performance.

Amazon Bedrock AgentCore provided observability metrics

For agents running in the AgentCore runtime, AgentCore automatically generates a set of session metrics which you can view in the Amazon CloudWatch Logs generative AI observability page. You can also use AgentCore observability to monitor the performance of memory, gateway, and built-in tool resources, even if you're not using the AgentCore runtime to host your agents. For memory, gateway, and built-in tool resources, AgentCore outputs a default set of data to CloudWatch.

The following table summarizes the default data provided for each resource type, and where the data is available.

Resource type	Service-provided data	Available in CloudWatch gen AI observability	Available in CloudWatch (Logs or metrics)
Agent	Metrics	Yes	Yes
Memory	Metrics, Spans*, Logs*	No	Yes
Gateway	Metrics	No	Yes
Tools	Metrics	No	Yes

* memory spans and logs require enablement. See the section called “Add observability to your agents” to learn more.

Note

To view metrics, spans, and traces for AgentCore, you need to perform a one-time setup process to enable CloudWatch Transaction Search. To learn more see the section called “Enabling AgentCore runtime observability”.

The AgentCore service only provides logs for memory resources. You can supply your own logs for other resource types, and AgentCore will save them in CloudWatch Logs. Note that to store logs for memory, gateway, and built-in tool resources, you need to configure the necessary log groups in CloudWatch. See the section called “Enable observability for AgentCore memory, gateway, and built-in tool resources” to learn more.

Refer to the following topics to learn about the default service-provided observability metrics for AgentCore runtime, memory, and gateway resources.

Topics

- AgentCore provided runtime metrics
- AgentCore provided memory metrics and spans
- AgentCore provided gateway metrics and logs
- AgentCore provided built-in tools metrics

By instrumenting your agent code, you can also gather more detailed trace and span data as well as custom metrics. See the section called “Enabling observability in agent code for AgentCore-hosted agents” to learn more.

AgentCore provided runtime metrics

The runtime metrics provided by AgentCore give you visibility into your agent execution activity levels, processing latency, resource utilization, and error rates. AgentCore also provides aggregated metrics for total invocations and sessions.

The following list describes the runtime metrics provided by AgentCore. Runtime metrics are batched at one minute intervals. To learn more about viewing runtime metrics, see the section called “View metrics for your agents”.

Invocations

> Shows the total number of requests made to the Data Plane API. Each API call counts as one invocation, regardless of the request payload size or response status.

Invocations (aggregated)

> Shows the total number of invocations across all resources

Throttles

> Displays the number of requests throttled by the service due to exceeding allowed TPS (Transactions Per Second) or quota limits. These requests return ThrottlingException with HTTP status code 429. Monitor this metric to determine if you need to review your service quotas or optimize request patterns.

System Errors

Shows the number of server-side errors encountered by AgentCore during request processing. High levels of server-side errors can indicate potential infrastructure or service issues that require investigation. See the section called "Error types" for a list of possible error codes.

User Errors

Represents the number of client-side errors resulting from invalid requests. These require user action to resolve. High levels of client-side errors can indicate issues with request formatting or permissions that need to be addressed. See the section called "Error types" for a list of possible error codes.

Latency

The total time elapsed between receiving the request and sending the final response token. Represents complete end-to-end processing time of the request.

Total Errors

The total number of system and user errors. In the Amazon Bedrock AgentCore console, this metric displays the number of errors as a percentage of the total number of invocations.

Session Count

Shows the total number of agent sessions. Useful for monitoring overall platform usage, capacity planning, and understanding user engagement patterns.

Sessions (aggregated)

Shows the total number of sessions across all resources.

Error types

The following list defines the possible error types for user, system, and throttling errors.

User error codes

- `InvocationError.Validation` - Client provided invalid input (400)
- `InvocationError.ResourceNotFound` - Requested resource doesn't exist (404)
- `InvocationError.AccessDenied` - Client lacks permissions (403)
- `InvocationError.Conflict` - Resource conflict (409)

System error codes

- `InvocationError.Internal` - Internal server error (500)

Throttling error codes

- `InvocationError.Throttling` - Rate limiting (429)
- `InvocationError.ServiceQuota` - Service-side quota/limit reached (402)

AgentCore provided memory metrics and spans

For the AgentCore memory resource type, AgentCore outputs metrics to Amazon CloudWatch by default. AgentCore also outputs a default set of spans and logs, if you enable these. See the section called "Enable observability for AgentCore memory, gateway, and built-in tool resources" to learn more about enabling spans and logs.

Refer to the following sections to learn more about the provided observability data for your agent memory stores.

Provided memory metrics

The AgentCore memory resource type provides the following metrics by default.

Latency

The total time elapsed between receiving the request and sending the final response token. Represents complete end-to-end processing of the request. For a create event, this represents the end to end time taken from last createEvent that met strategy criteria to the memory stored completed.

Invocations

The total number of API requests made to the data plane and control plane. This metric also tracks the number of memory ingestion events.

System Errors

Number of invocations that result in AWS server-side errors.

User Errors

Number of invocations that result in client-side errors.

Errors

Total number of errors that occur while processing API requests in the data plane and control plane. This metric also tracks the total errors that occur during memory ingestion.

Throttles

Number of invocations that the system throttled. Throttled requests don't count as invocations or errors.

Creation Count

Counts the number of created memory events and memory records.

Provided span data

To enhance observability, AgentCore provides structured spans that trace the relationship between events and the memories they generate or access. To enable this span data, you need to instrument your agent code. See the section called "Add observability to your agents" to learn more.

This span data is available in full in CloudWatch Logs and CloudWatch Application Signals. To learn more about viewing observability data, see the section called "View metrics for your agents".

The following table defines the operations for which spans are created and the attributes for each captured span.

Operation name	Span attributes	Description
`CreateEvent`	`memory.id` , `session.id` , `event.id`, `actor.id`, `throttled` , `error`, `fault`	Creates a new event within a memory session
`GetEvent`	`memory.id` , `session.id` , `event.id`, `actor.id`, `throttled` , `error`, `fault`	Retrieves an existing memory event
`ListEvents`	`memory.id` , `session.id` , `event.id`, `actor.id`, `throttled` , `error`, `fault`	Lists events within a session

Operation name	Span attributes	Description
DeleteEvent	`memory.id` , `session.id` , `event.id`, `actor.id`, `throttled` , `error`, `fault`	Deletes an event from memory
RetrieveMemoryRecords	`memory.id` , `namespace` , `throttled` , `error`, `fault`	Retrieves memory records for a given namespace
ListMemoryRecords	`memory.id` , `namespace` , `throttled` , `error`, `fault`	Lists available memory records

Provided log data

AgentCore provides structured logs that help you monitor and troubleshoot key AgentCore Memory resource processes. To enable this log data, you need to instrument your agent code. See the section called "Add observability to your agents" to learn more.

AgentCore can output logs to CloudWatch Logs, Amazon S3, or Firehose stream. If you use a CloudWatch Logs destination, these logs are stored under the default log group `/aws/vendedlogs/bedrock-agentcore/memory/APPLICATION_LOGS/{memory_id}` or under a custom log group starting with `/aws/vendedlogs/`. See the section called "Enable observability for AgentCore memory, gateway, and built-in tool resources" to learn more.

When the `DeleteMemory` operation is called, logs are generated for the start and completion of the deletion process. Any corresponding deletion error logs will be provided with insights into why the call failed.

We also provide logs for various stages in the long-term memory creation process, namely extraction and consolidation. When new short term memory events are provided, AgentCore extracts key concepts from responses to begin the formation of new long-term memory records. Once these have been created, they are integrated with existing memory records to create a unified store of distinct memories.

See the following breakdown to learn how each workflow helps you monitor the formation of new memories:

Extraction logs

- Start and completion of extraction processing
- Number of memories successfully extracted
- Any errors in deserializing or processing input events

Consolidation logs:

- Start and completion of consolidation processing
- Number of memories requiring consolidation
- Success/failure of memory additions and updates
- Related memory retrieval status

The following table provides a more detailed breakdown of how different memory resource workflows use log fields alongside the log body itself to provide request-specific information.

Workflow name	Log fields	Description
Extraction	resource_arn, event_timestamp, memory_strategy_id, namespace, actor_id, session_id, event_id, requestId, isError	Analyzes incoming conversations to generate new memories
Consolidation	resource_arn, event_timestamp, memory_strategy_id, namespace, session_id, requestId, isError	Combines extracted memories with existing memories

AgentCore provided gateway metrics and logs

The following sections describe the gateway metrics and logs output by AgentCore to Amazon CloudWatch. These metrics aren't available on the CloudWatch generative AI observability page. Gateway metrics are batched at one minute intervals. To learn more about viewing gateway metrics, see the section called “View metrics for your agents”.

Note

To enable service-provided logs for AgentCore gateways, you need to configure the necessary CloudWatch resources. See the section called "Enable observability for AgentCore memory, gateway, and built-in tool resources" to learn more.

Provided metrics

Invocations

The total number of requests made to each Data Plane API. Each API call counts as one invocation regardless of the response status.

Throttles [429]

The number of requests throttled (status code 429) by the service.

SystemErrors [5xx]

The number of requests which failed with 5xx status code.

UserErrors [4xx]

The number of requests which failed with 4xx status codes other than 429.

Latency

The time elapsed between when the service receives the request and when it begins sending the first response token.

Duration

The total time elapsed between receiving the request and sending the final response token. Represents complete end-to-end processing time of the request.

TargetExecutionTime

The total time take to execute the target over Lambda, OpenAPI, etc. This metric helps you to determine the contribution of the target to the total Latency.

TargetType

The total number of requests served by each type of target (MCP, Lambda, OpenAPI).

Provided log data

AgentCore provides logs that help you monitor and troubleshoot key AgentCore gateway resource processes. To enable this log data, you need to create a log destination.

AgentCore can output logs to CloudWatch Logs, Amazon S3, or Firehose stream. If you use a CloudWatch Logs destination, these logs are stored under the default log group `/aws/vendedlogs/bedrock-agentcore/gateway/APPLICATION_LOGS/{gateway_id}` or under a custom log group starting with `/aws/vendedlogs/`. See the section called "Enable observability for AgentCore memory, gateway, and built-in tool resources" to learn more.

AgentCore logs the following information for gateway resources:

- Start and completion of gateway requests processing
- Error messages for Target configurations
- MCP Requests with missing or incorrect authorization headers
- MCP Requests with incorrect request parameters (tools, method)

AgentCore provided built-in tools metrics

AgentCore provides the following built-in metrics for the code interpreter and browser tools. Built-in tool metrics are batched at one minute intervals. To learn more about AgentCore tools, see *AgentCore Built-in Tools: Interact with your applications using built-in tools*.

Invoke tool:

Invocations

> The total number of requests made to the Data Plane API. Each API call counts as one invocation, regardless of the request payload size or response status.

Throttles

> The number of requests throttled by the service due to exceeding allowed TPS (Transactions Per Second) or quota limits. These requests return ThrottlingException with HTTP status code 429.

SystemErrors

> The number of server-side errors encountered during request processing.

UserErrors

The number of client-side errors resulting from invalid requests. This require user action in order to resolve.

Latency

The time elapsed between when the service receives the request and when it begins sending the first response token. Important for measuring initial response time.

Create tool session:

Invocations

The total number of requests made to the Data Plane API. Each API call counts as one invocation, regardless of the request payload size or response status.

Throttles

The number of requests throttled by the service due to exceeding allowed TPS (Transactions Per Second) or quota limits. These requests return ThrottlingException with HTTP status code 429.

SystemErrors

The number of server-side errors encountered during request processing.

UserErrors

The number of client-side errors resulting from invalid requests. This require user action in order to resolve.

Latency

The time elapsed between when the service receives the request and when it begins sending the first response token. Important for measuring initial response time.

Duration

The duration of tool session (Operation becomes CodeInterpreterSession/BrowserSession).

Browser user takeover:

TakerOverCount

The total number of user taking over

TakerOverReleaseCount

The total number of user releasing control

TakerOverDuration

The duration of user taking over

View observability data for your Amazon Bedrock AgentCore agents

After implementing observability in your agent, you can view the collected metrics and traces in both the CloudWatch console generative AI observability page and in CloudWatch Logs. Refer to the following sections to learn how to view metrics for your agents.

View data using generative AI observability in Amazon CloudWatch

The CloudWatch generative AI observability page displays all of the service-provided metrics output by the AgentCore agent runtime, as well as span- and trace-derived data if you have enabled instrumentation in your agent code. To view the observability dashboard in CloudWatch, open the Amazon CloudWatch GenAi Observability page.

With generative AI observability in CloudWatch, you can view tailored dashboards with graphs and other visualizations of your data, as well as error breakdowns, trace visualizations and more. To learn more about using generative AI observability in CloudWatch, including how to look at your agents' individual session and trace data, see Amazon Bedrock AgentCore agents in the *Amazon CloudWatch user guide*.

View other data in CloudWatch

All of the service-provided metrics and spans can also be viewed in CloudWatch, along with any metrics that your instrumented agent code outputs.

To view this data, refer to the following sections.

Logs

1. Open the CloudWatch console.
2. In the left hand navigation pane, expand **Logs** and select **Log groups**

3. Use the search field to find the log group for your agent, memory, or gateway resource.

   AgentCore agent log groups have the following format:

   - **Standard logs** - stdout/stderr output
     - **Location**: /aws/bedrock-agentcore/runtimes/<agent_id>-<endpoint_name>/[runtime-logs] <UUID>
     - **Contains**: Runtime errors, application logs, debugging statements
     - **Example Usage**:
       - print("Processing request...") # Appears in standard logs
       - logging.info("Request processed successfully") # Appears in standard logs
   - **OTEL structured logs** - Detailed operation information
     - **Location**: /aws/bedrock-agentcore/runtimes/<agent_id>-<endpoint_name>/otel-rt-logs
     - **Contains**: Execution details, error tracking, performance data
     - **Automatic collection**: No additional code required - generated by ADOT instrumentation
     - **Benefits**: Can include correlation IDs linking logs to relevant traces

Traces and Spans

Traces provide visibility into request execution paths through your agent:

- Location: `/aws/spans/default`
- Access via: CloudWatch Transaction Search console
- Requirements: CloudWatch Transaction Search must be enabled

Traces automatically capture:

- Agent invocation sequences
- Integration with framework components (LangChain, etc.)
- LLM calls and responses
- Tool invocations and results
- Error paths and exceptions

For distributed tracing across services, you can use standard HTTP headers:

- AWS X-Ray format: `X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1`
- W3C format: `traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01`

To view traces:

- Navigate to CloudWatch console
- Select **Transaction Search** from the left navigation
- Filter by service name or other criteria
- Select a trace to view the detailed execution graph

Metrics

If you have enabled observability by instrumenting your agent code as described in the section called "Enabling AgentCore runtime observability", your agent automatically generates OTEL metrics, which are sent to CloudWatch using Enhanced Metric Format (EMF):

- Namespace: `bedrock-agentcore`
- Access via: CloudWatch Metrics console
- Contents: Custom metrics generated by your agent code and frameworks
- Automatic collection: No additional code required - generated by ADOT instrumentation

In addition to agent-emitted metrics, the AgentCore service publishes standard service metrics to CloudWatch. Refer to the section called "Provided runtime metrics" for a list of these metrics.

Create agent and tool identities with AgentCore Identity

Amazon Bedrock AgentCore Identity is an identity and credential management service designed specifically for AI agents and automated workloads. It provides secure authentication, authorization, and credential management capabilities that enable agents and tools to access AWS resources and third-party services on behalf of users while helping to maintain strict security controls and audit trails. Agent identities are implemented as workload identities with specialized attributes that enable agent-specific capabilities while helping to maintain compatibility with industry-standard workload identity patterns. The service integrates natively with Amazon Bedrock AgentCore to provide identity and credential management for agent applications, including Host agent or tools with Amazon Bedrock AgentCore Runtime and Amazon Bedrock AgentCore Gateway: Securely connect tools and other resources to your Gateway.

Topics

Overview of Amazon Bedrock AgentCore Identity

In the rapidly evolving landscape of AI agents, organizations need robust identity management solutions that can handle the unique challenges associated with non-human identities. Amazon Bedrock AgentCore Identity addresses these challenges by providing a centralized capability for managing agent identities, securing credentials, and enabling seamless integration with AWS and third-party services through Sigv4, standardized OAuth 2.0 flows, and API keys.

The service implements authentication and authorization controls that verify each request independently, requiring explicit verification for all access attempts regardless of source. It integrates seamlessly with AWS services while also enabling agents to securely access external tools and services. Whether you're building simple automation scripts or complex multi-agent systems, AgentCore Identity provides the identity foundation to help your applications operate securely and efficiently.

Topics

- Features of AgentCore Identity
- AgentCore Identity terminology
- Example use cases

Features of AgentCore Identity

AgentCore Identity offers a set of features designed to address the unique challenges of workload identity management and credential security:

Topics

- Centralized agent identity management
- Secure credential storage
- OAuth 2.0 flow support
- Agent identity and access controls
- AgentCore SDK Integration
- Request verification security

Centralized agent identity management

Create, manage, and organize agent and workload identities through a unified directory service that acts as the single source of truth for all agent identities within your organization. Each agent receives a unique identity with associated metadata (such as name, ARN, OAuth return URLs, created time, last updated time) that can be managed centrally across your organization. The agent identity directory functions similarly to Cognito User Pools, providing a unit of governance that allows administrators to configure policies across a common set of agent identities. Agent identities are managed as specialized workload identities with agent-specific attributes and capabilities. For detailed procedures on creating and managing agent identities, see Manage workload identities with AgentCore Identity.

The centralized approach eliminates the complexity of managing agent identities across different environments and systems. Whether your agents run on AgentCore Runtime, self-hosted environments, or hybrid deployments, the service provides consistent identity management regardless of where your agents are deployed. Each agent identity receives a unique ARN (such as `arn:aws:bedrock-agentcore:region:account:workload-identity/directory/default/workload-identity/agent-name`) that enables precise access control and resource management. This centralization also enables hierarchical organization and group-based access controls, making it easier to implement enterprise-wide governance policies and maintain compliance across all agent operations. The hierarchical structure in the ARN path (with directory/default/workload-identity/agent-name components) allows administrators to organize agents logically and apply policies at different levels of the hierarchy—for example, targeting all agents within a specific directory or with similar attributes—without having to manage each agent identity individually.

Secure credential storage

The token vault provides security for storing OAuth 2.0 tokens, OAuth client credentials, and API keys with comprehensive encryption at rest and in transit. All credentials are encrypted using either customer-managed or service-managed AWS KMS keys and access-controlled to prevent unauthorized retrieval. The vault implements strict access controls, ensuring that credentials can only be accessed by authorized agents for specific purposes and only when they present verifiable proof of workload identity.

Building on OAuth 2.0's scope-based security model, the token vault implements additional security measures where every access request is validated independently, even from callers within the same trust domain. This extra security mechanism is necessary to protect end-user data from malicious or misbehaving agent code. The vault securely stores OAuth 2.0 tokens, reducing security risks while improving your overall security posture.

OAuth 2.0 flow support

Native support for both OAuth 2.0 client credentials grant (machine-to-machine) and OAuth 2.0 authorization code grant (user-delegated access) flows enables comprehensive authentication patterns for different use cases. The service handles the complexity of OAuth 2.0 implementations while providing simple APIs for agents to access AWS resources and third-party services. For 2LO flows, agents can authenticate themselves directly with resource servers without user interaction, while 3LO flows enable explicit user consent and authorization for accessing user-specific data from external services.

The service also provides built-in OAuth 2.0 credential providers for popular services such as Google, GitHub, Slack, Salesforce, and Atlassian (Jira), with authorization server endpoints and provider-specific parameters pre-filled to reduce development effort. For custom integrations, the service supports configurable OAuth 2.0 credential providers that can be tailored to work with any OAuth 2.0-compatible resource server. This comprehensive OAuth 2.0 support eliminates the heavy-lifting of agent developers implementing complex authorization flows and reduces the risk of security vulnerabilities in custom implementations. For comprehensive information about configuring these providers, see Configure credential provider.

Agent identity and access controls

AgentCore Identity supports impersonation flow where agents can access resources using credentials provided to them. This approach enables agents to perform actions on behalf of users while maintaining audit trails and access controls. The impersonation process allows agents to use provided credentials to access resources, with authorization decisions based on those credentials.

AgentCore SDK Integration

Seamless integration with the AgentCore SDK through declarative annotations like @requires_access_token and @requires_api_key automatically handles credential retrieval and injection, reducing boilerplate code and potential security vulnerabilities. These annotations eliminate the need for developers to implement complex OAuth flows manually, instead providing a simple declarative interface that abstracts away the underlying complexity of token management and credential handling.

The SDK integration also provides automatic error handling for common scenarios such as token expiration and user consent requirements. When tokens expire or user consent is needed, the SDK automatically generates appropriate authorization URLs and handles the OAuth flow orchestration, presenting developers with simple success or failure responses. This integration significantly reduces development time and the likelihood of security vulnerabilities while ensuring that all credential operations follow security best practices.

Request verification security

The service implements validation of all requests, including token signature verification, expiration checks, and scope validation.

By treating every request as requiring verification and requiring explicit proof of authorization, the service implements security validation for each request. All operations are logged with detailed context for security monitoring and compliance reporting, providing visibility into agent activities.

These features combine to provide significant benefits for organizations deploying AI agents:

- **Reduced Security Risk**: Centralized credential management eliminates the need to embed secrets in agent code or configuration files.
- **Simplified Development**: Declarative APIs and SDK integration reduce the complexity of implementing secure authentication in agent applications.
- **Enhanced Compliance**: Comprehensive audit trails and access controls support regulatory compliance requirements.
- **Operational Efficiency**: Automated credential refresh reduces operational overhead while improving security posture.

AgentCore Identity terminology

AgentCore Identity uses specific terminology to describe the components, processes, and relationships involved in workload identity management and credential handling. Understanding these terms will help you better comprehend how the service orchestrates secure authentication and authorization across multiple parties in agent workflows.

AgentCore Identity terminology definitions

Term	Definition
Identity and Authentication	
Agent	An AI-powered application or automated workload that performs tasks on behalf of users by accessing AWS resources and third-party services. Agents act with pre-authorized user consent, to accomplish user goals, such as retrieving data from APIs, processing information, or integrating with third-party systems. Unlike traditional applications that run with static credentials, agents require dynamic identity managemen t to securely access resources across multiple trust domains while maintaining proper authentication and authorization boundaries.
Agent identity	A unique identifier and associated metadata for an AI agent or automated workload. Agent identities are implemented as workload identities with specific attributes that identify them as agents, enabling specialized agent capabilities while

Term	Definition
	maintaining compatibility with broader workload identity standards. Agent identities enable agents to authenticate as themselves rather than impersonating users, supporting delegation-based access patterns.
Agent identity directory	A centralized registry that manages agent identities and their associated metadata and access policies. Similar to Cognito User Pools, it acts as a unit of governance for organizing agent identities within an account or region.
Workload identity	The underlying technical implementation for agent identities, representing a logical application or workload that is independent of specific hardware or infrastructure. Workload identities can operate across different environments while maintaining consistent authentication. Agent identities are a specialized type of workload identity with additional agent-specific attributes and capabilities.
Integration and Protocols	
Cross-service agents	AI agents that perform actions across multiple services, which may include accessing system resources (using machine-to-machine authentication) or user-specific data (using user-delegated access). Examples include agents that integrate with multiple backend systems for data processing or agents that access a user's calendar, email, and document storage. These agents require sophisticated identity management to operate securely across different trust domains.
MCP client	A client component that allows agents to communicate with MCP servers to access external tools and resources. MCP clients present authentication tokens to access MCP tools securely.

Term	Definition
MCP server	An intermediate server that hosts tools and resources for MCP clients. MCP servers act as OAuth 2.0 resource servers when accessed by agents and as OAuth 2.0 clients when accessing downstream resources.
Model context protocol (MCP)	MCP is an open protocol that standardizes how applications provide context to language models. AgentCore Identity is MCP-compliant, supporting standard protocols for agent-to-tool communication and enabling secure integration with MCP servers and tools.
OAuth and Token Management	
OAuth 2.0	An industry-standard authorization framework (defined in RFC 6749) that enables applications to obtain limited access to user accounts on external services without exposing user credentials. OAuth 2.0 provides secure delegation by allowing users to grant third-party applications access to their resources through access tokens rather than sharing passwords. For agent applications, OAuth 2.0 enables secure access to user data across multiple services while maintaining proper authentication boundaries and user consent mechanisms.
OAuth 2.0 authorizer	An SDK component that authenticates and authorizes incoming OAuth 2.0 API requests to agent endpoints. It validates tokens before allowing access to agent services.
OAuth 2.0 client credentials grant (2LO)	OAuth client credentials grant used for machine-to-machine authentication where no user interaction is required. Agents use 2LO to authenticate themselves directly with resource servers.
OAuth 2.0 authorization code grant (3LO)	OAuth authorization code grant that involves user consent and interaction. Agents use 3LO when they need explicit user permission to access user-specific data from external services like Google Calendar or Salesforce.

Term	Definition
Agent access token	An AWS-signed token that contains both workload identity and user identity information, enabling downstream services to make authorization decisions based on both identities. These tokens are created through the token exchange process.
Security and Trust	
Identity propagation	The process of maintaining and passing identity context through a chain of service calls. This enables downstream services to make authorization decisions based on both the calling service identity and the original user identity.
Trust domain	A security boundary within which entities share common authentication and authorization mechanisms. Agent workflows often span multiple trust domains, requiring careful identity propagation and token exchange.
Request verification security	A security model where every request is authenticated and authorized regardless of source or previous trust relationships. AgentCore Identity implements request verification to ensure validation of all access requests.
Service Components	
Resource credential provider	A component that manages connections to external identity providers and resource servers, handling OAuth 2.0 authorization flows and credential retrieval. It orchestrates the complex process of obtaining and refreshing credentials from third-party services. For detailed configuration information, see Configure credential provider.
Token vault	A secure storage system for OAuth 2.0 tokens, API keys, and other credentials that operates with strict access controls. The token vault ensures credentials can only be accessed by the specific agent and user combination that originally obtained them.

Example use cases

Amazon Bedrock AgentCore Identity supports a wide range of use cases across different industries and application types. This section provides detailed examples of how the service can be applied in specific scenarios, demonstrating both user-delegated access (OAuth 2.0 authorization code grant) and machine-to-machine authentication (OAuth 2.0 client credentials grant) patterns.

Topics

Personal assistant agents

AI agents that help users manage their personal productivity by accessing services like Google Drive, Microsoft Office 365, or Slack represent one of the most common and valuable applications of AgentCore Identity. These agents use OAuth 2.0 authorization code grant to obtain explicit user consent (3Lo) and access user data securely across multiple systems. For example, a research agent might search the web using AgentCore Browser, generate a comprehensive report, and save it to the user's Google Drive, all while maintaining proper authentication and authorization throughout the entire workflow.

The complexity of managing credentials across multiple third-party services makes AgentCore Identity particularly valuable for personal assistant scenarios. Consider a meeting agent that needs to access a user's Google Calendar to check availability, join a Zoom meeting to take notes, schedule follow-up meetings, and draft emails for approval. Each of these services requires different authentication mechanisms and user consent, but AgentCore Identity orchestrates the entire process seamlessly while the agent maintains its own identity and the user retains control over what data is accessed.

Personal assistant agents also benefit from AgentCore Identity's token storage and secure credential management, which eliminate the need for users to repeatedly authorize access to their accounts. Once a user has granted permission for an agent to access their Google Drive, for instance, the agent can continue to access that service for subsequent tasks without requiring re-authorization, as long as the stored tokens remain valid. This creates a smooth user experience while maintaining security through proper token management.

Enterprise automation agents

Agents that automate business processes by integrating with enterprise systems like Salesforce, SharePoint, or internal APIs represent a critical use case for organizations seeking to improve operational efficiency. These agents typically use OAuth 2.0 client credentials grant for machine-to-machine authentication (2Lo) when accessing systems that don't require user interaction, and may require access to multiple systems with different authentication requirements. For example, an HR automation agent might need to access employee data from an HRIS system, update records in Salesforce, and generate reports in SharePoint, each requiring different credentials and authorization scopes.

Enterprise automation scenarios often involve complex workflows that span multiple trust domains and require careful identity propagation to maintain security and compliance. AgentCore Identity addresses this challenge by providing a centralized approach to credential management that works across different enterprise systems. The service supports both AWS-hosted resources with IAM-based authentication and external enterprise systems with OAuth 2.0 or API key authentication, enabling agents to operate seamlessly across hybrid environments while helping to maintain consistent security standards.

The audit and compliance capabilities of AgentCore Identity are particularly important for enterprise automation use cases, where organizations need to maintain detailed records of automated actions for regulatory compliance and security monitoring. Every action performed by an enterprise automation agent is logged with both the agent identity and any associated user context, providing complete traceability of automated business processes. This level of visibility helps with compliance requirements and enables organizations to quickly identify and respond to any unauthorized or unexpected agent behavior.

Customer service agents

AI agents that assist customer service representatives by accessing customer data from CRM systems, knowledge bases, and support ticketing systems must authenticate securely while providing real-time assistance during customer interactions. These agents need to access sensitive customer information from multiple sources while maintaining strict security controls and audit trails. For example, a customer service agent might need to access a customer's order history from an e-commerce platform, check their support ticket status in a ticketing system, and retrieve relevant troubleshooting information from a knowledge base, all while the customer is on the phone.

The real-time nature of customer service interactions makes credential management particularly challenging, as agents cannot afford delays caused by authentication failures or expired tokens. AgentCore Identity addresses this challenge through its comprehensive error handling, ensuring that customer service agents can access the information they need without interruption. The service also supports fine-grained access controls that can be configured to have agents only access customer data that is relevant to the specific interaction, supporting privacy requirements and regulatory compliance.

Data processing and analytics agents

Agents that collect, process, and analyze data from multiple sources, including cloud storage services, databases, and APIs, often require long-running access to data sources. These agents typically operate on scheduled or triggered workflows that may run for hours or days, accessing large datasets from various sources to perform complex analytics operations. For example, a financial analytics agent might collect transaction data from multiple payment processors, combine it with customer data from CRM systems, and generate comprehensive reports that are stored in data warehouses and shared with business stakeholders.

The long-running nature of data processing workflows makes credential management particularly complex, as tokens may expire during processing and agents need to handle authentication failures gracefully without losing progress on lengthy operations. AgentCore Identity addresses these challenges through its robust error handling, helping data processing agents maintain access to required resources throughout their entire execution lifecycle. The service also supports batch processing scenarios where agents need to access multiple data sources simultaneously, providing efficient credential management that scales with the complexity of the data processing workflow.

Data processing and analytics use cases also benefit from AgentCore Identity's support for different authentication mechanisms across various data sources. A single analytics workflow might need to access data from AWS services using IAM credentials, third-party APIs using OAuth 2.0 tokens, and on-premise databases using API keys or other authentication methods. AgentCore Identity provides a unified interface for managing all these different credential types, enabling data processing agents to focus on their core analytics functions rather than the complexity of credential management across diverse systems.

Development and DevOps agents

Agents that automate software development workflows by integrating with version control systems, CI/CD pipelines, and deployment systems require secure access to development tools and infrastructure while maintaining comprehensive audit trails for compliance purposes. These agents might automatically create pull requests, trigger builds, deploy applications, and update documentation across multiple development tools and systems. For example, a DevOps agent might monitor application performance, detect issues, automatically create bug reports in JIRA, generate fixes through code analysis, and deploy patches through CI/CD pipelines, all while maintaining proper authentication and authorization throughout the entire workflow.

Development and DevOps scenarios present unique security challenges because agents often need elevated privileges to perform deployment and infrastructure management tasks, while also needing to maintain strict controls to prevent unauthorized changes to production systems. AgentCore Identity addresses these challenges through its fine-grained access control capabilities and comprehensive audit logging, ensuring that DevOps agents can perform necessary automation tasks while supporting security and compliance. The service supports role-based access controls that can be configured to limit agent access to specific environments, repositories, or deployment targets based on the agent's identity and the context of the operation.

The audit and compliance capabilities of AgentCore Identity are particularly valuable for development and DevOps use cases, where organizations need to maintain detailed records of all changes to code, infrastructure, and deployment configurations. Every action performed by a DevOps agent is logged with complete context, including the agent identity, the specific resources accessed, and the changes made, providing the level of traceability that supports regulatory compliance and security auditing. This comprehensive logging also enables organizations to quickly identify the root cause of issues and roll back changes when necessary, supporting the reliability and stability of development and deployment processes.

Getting started with Amazon Bedrock AgentCore Identity

This guide walks you through the essential steps to start using Amazon Bedrock AgentCore Identity for your AI agents. You'll learn how to set up your development environment, install the necessary SDKs, create your first agent identity, and allow your agent to access external resources securely.

By the end of this section, you'll have a working agent that can retrieve access tokens from Google with AgentCore Identity OAuth2 Credential Provider, and read files from Google Drive using access tokens. For detailed information about OAuth2 flows, see Manage credential providers with AgentCore Identity.

Topics

Prerequisites

Before you start, you need:

- An AWS account with appropriate permissions (for example, `BedrockAgentCoreFullAccess`)
- Basic understanding of Python programming

Install the SDK

To get started, install the `bedrock-agentcore` package:

```
pip install bedrock-agentcore
```

Obtain Google Client ID and Client Secret

To allow your agent to access Google Drive, you need to obtain a Google client ID and client secret for your agent. Go to the Google Developer Console and follow these steps:

1. Enable Google Drive API
2. Create OAuth consent screen
3. Create a new web application for the agent, for example, "My Agent 1"
4. Add the following OAuth 2.0 scope to your agent application: `https://www.googleapis.com/auth/drive.metadata.readonly`

5. Create OAuth 2.0 Credentials for the new web application, and note the generated Google client ID and client secret

> **Note**
>
> You must add the following URI to your application's redirect URI list: `https://bedrock-agentcore.us-east-1.amazonaws.com/identities/oauth2/callback`

Step 1: Import Identity and Auth modules

Add this import statement to your Python file:

```
from bedrock_agentcore.services.identity import IdentityClient
from bedrock_agentcore.identity.auth import requires_access_token, requires_api_key
```

Step 2: Set up an OAuth 2.0 Credential Provider

Create a new OAuth 2.0 Credential Provider with the Google client ID and client secret obtained earlier using the following AWS CLI command:

```
aws bedrock-agentcore-control create-oauth2-credential-provider \
  --region us-east-1 \
  --name "google-provider" \
  --credential-provider-vendor "GoogleOauth2" \
  --oauth2-provider-config-input '{
      "googleOauth2ProviderConfig": {
        "clientId": "<your-google-client-id>",
        "clientSecret": "<your-google-client-secret>"
      }
    }'
```

Behind the scenes, the SDK makes a call to the `CreateOauth2CredentialProvider` API.

Step 3: Obtain an OAuth 2.0 access token

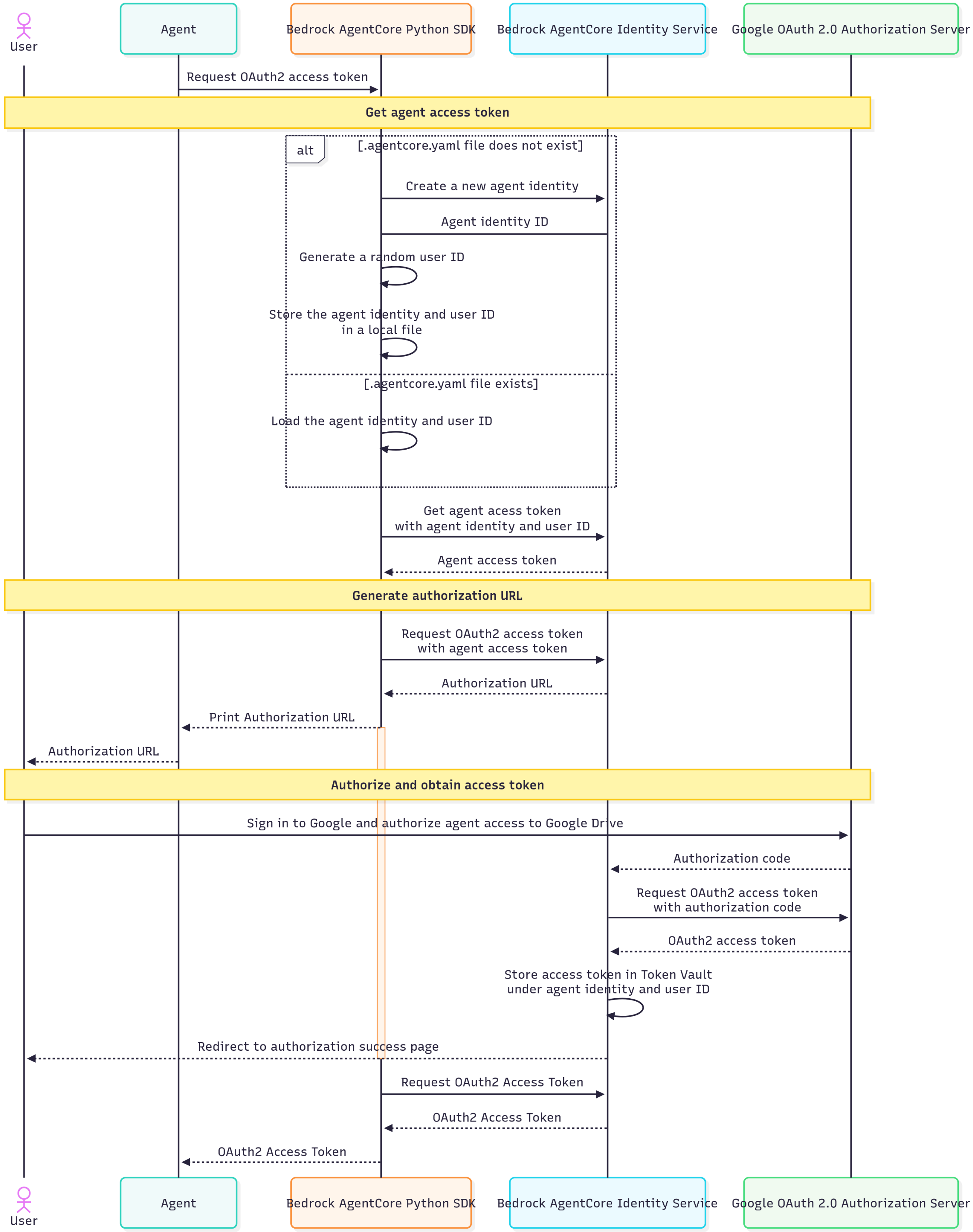
Once you have the Google Credential Provider created in the previous step, add the `@requires_access_token` decorator to your agent code that requires a Google access token.

Copy the authorization URL from your console output, then paste it in your browser and complete the consent flow with Google Drive.

```
import asyncio

# Injects Google Access Token
@requires_access_token(# Uses the same credential provider name created above
    provider_name= "google-provider",
    # Requires Google OAuth2 scope to access Google Drive
    scopes= ["https://www.googleapis.com/auth/drive.metadata.readonly"],
    # Sets to OAuth 2.0 Authorization Code flow
    auth_flow= "USER_FEDERATION",
    # Prints authorization URL to console
    on_auth_url= lambda x: print("\nPlease copy and paste this URL in your browser:\n"
 + x),
    # If false, caches obtained access token
    force_authentication= False,)async def write_to_google_drive(*, access_token: str):
    # Prints the access token obtained from Google
    print(access_token)asyncio.run(write_to_google_drive(access_token= ""))
```

Behind the scenes, the `@requires_access_token` decorator runs through the following sequence:


User
Agent
Bedrock AgentCore Python SDK
Bedrock AgentCore Identity Service
Google OAuth 2.0 Authorization Server
Request OAuth2 access token
Get agent access token
alt
[.agentcore.yaml file does not exist]
Create a new agent identity
Agent identity ID
Generate a random user ID
Store the agent identity and user ID in a local file
[.agentcore.yaml file exists]
Load the agent identity and user ID
Get agent acess token with agent identity and user ID
Agent access token
Generate authorization URL
Request OAuth2 access token with agent access token
Authorization URL
Print Authorization URL
Authorization URL
Authorize and obtain access token
Sign in to Google and authorize agent access to Google Drive
Authorization code
Request OAuth2 access token with authorization code
OAuth2 access token
Store access token in Token Vault under agent identity and user ID
Redirect to authorization success page
Request OAuth2 Access Token
OAuth2 Access Token
OAuth2 Access Token
User
Agent
Bedrock AgentCore Python SDK
Bedrock AgentCore Identity Service
Google OAuth 2.0 Authorization Server

1. The SDK makes API calls to `CreateWorkloadIdentity`, `GetWorkloadAccessToken`, and `GetResourceOauth2Token`.
2. When running the agent code locally, the SDK automatically generates an agent identity ID and a random user ID for local testing, and stores them in a local file called `.agentcore.yaml`.
3. When running the agent code with AgentCore Runtime, the SDK does not generate an agent identity ID or random user ID. Instead, it uses the agent identity ID assigned, and the user ID or JWT token passed in by the agent caller.
4. Agent access token is an encrypted (opaque) token that contains the agent identity ID and user ID.
5. AgentCore Identity service stores the Google access token in the Token Vault under the agent identity ID and user ID. This creates a binding among the agent identity, user identity, and the Google access token.

Step 4: Use OAuth2 Access Token to Invoke External Resource

Once the agent obtains a Google access token with the steps above, it can use the access token to access Google Drive. Here is a full example that lists the names and IDs of the first 10 files that the user has access to.

First, install the Google client library for Python:

```
pip install --upgrade google-api-python-client google-auth-httplib2 google-auth-
oauthlib
```

Then, copy the following code:

```
import asyncio
from bedrock_agentcore.identity.auth import requires_access_token, requires_api_key
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError

SCOPES= ["https://www.googleapis.com/auth/drive.metadata.readonly"]

def main(access_token):
    """Shows basic usage of the Drive v3 API.
```

```
   Prints the names and ids of the first 10 files the user has access to.
   """
   creds= Credentials(token= access_token, scopes= SCOPES)try:
       service= build("drive", "v3", credentials= creds)# Call the Drive v3 API
       results= (service.files().list(pageSize= 10, fields= "nextPageToken, files(id,
name)").execute())items= results.get("files", [])if not items:
           print("No files found.")return
       print("Files:")for item in items:
           print(f"{item['name']}({item['id']})")except HttpError as error:
       # TODO(developer)- Handle errors from drive API.
       print(f"An error occurred: {error}")if __name__= = "__main__":

   # This annotation helps agent developer to obtain access tokens from external
applications
   @requires_access_token(provider_name= "google-provider",
       scopes= ["https://www.googleapis.com/auth/drive.metadata.readonly"], # Google
OAuth2 scopes
       auth_flow= "USER_FEDERATION", # 3LO flow
       on_auth_url= lambda x: print("Copy and paste this authorization url to your
browser", x), # prints authorization URL to console
       force_authentication= True,)async def read_from_google_drive(*, access_token:
str):
       print(access_token)#You can see the access_token
       # Make API calls...
       main(access_token)asyncio.run(read_from_google_drive(access_token= ""))
```

What's Next?

The example in this section focuses on practical implementation patterns that you can adapt for your specific use cases. You can embed the code as part of an agent, or a Model Context Protocol (MCP) tool. If you want to host your Agent code or MCP Tool with AgentCore Runtime, follow Host agent or tools with Amazon Bedrock AgentCore Runtime to copy the code above to AgentCore Runtime.

Using the AgentCore Identity console

The AgentCore Identity console provides a centralized interface for managing your agent authentication configurations. You can use the console to set up outbound identity providers for external service access, configure inbound identity settings for agent authentication, and manage API keys for services that require key-based authentication. This section contains step-by-step procedures for all console-based AgentCore Identity tasks.

Topics

- Configure an OAuth client
- Configure an API key

Configure an OAuth client

An OAuth client enables your agent to securely access external services on behalf of users without requiring them to share their credentials directly. For example, your agent can access a user's Google Drive files or Microsoft calendar events through OAuth authentication.

Topics

- Add OAuth client using included provider
- Add OAuth client using custom provider
- Update OAuth client
- Delete OAuth client

Add OAuth client using included provider

Built-in providers offer streamlined setup for popular services including Google, GitHub, Slack, and Salesforce. These providers have pre-configured authorization server endpoints and provider-specific parameters to reduce development effort.

To add an OAuth client using an included provider

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, choose **Add OAuth client / API key**, and then select **Add OAuth client**.
3. For **Name**, you can either use the auto-generated name or enter your own descriptive name to help you identify this OAuth client in your account. Use alphanumeric characters, hyphens, and underscores only, with a maximum length of 50 characters.
4. For **Provider**, choose **Included provider**.
5. Choose your identity provider from the available options (Google, GitHub, Microsoft, Salesforce, or Slack).
6. In the **Provider configurations** section, enter your client credentials:

   a. For **Client ID**, enter the unique identifier you received when registering your application with the identity provider.

   b. For **Client secret**, enter the confidential key associated with your client ID. AgentCore Identity securely stores this value for authentication.

7. Choose **Add OAuth Client**.

After creating the OAuth client, AgentCore Identity provides an ARN that you can reference in your agent code to request authentication tokens without embedding sensitive credentials in your application. You can find this ARN in the properties page of the OAuth client (Choose the client name in the **Outbound Auth** section).

Add OAuth client using custom provider

Custom providers enable you to connect to any OAuth2-compatible resource server beyond the built-in provider options. You can configure custom providers by having the system retrieve configuration details automatically, or by providing the server information manually.

To add an OAuth client using a custom provider

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, choose **Add OAuth client / API key**, and then select **Add OAuth client**.
3. For **Name**, you can either use the auto-generated name or enter your own descriptive name to help you identify this OAuth client in your account. Use alphanumeric characters, hyphens, and underscores only, with a maximum length of 50 characters.
4. For **Provider**, choose **Custom provider**.
5. In the **Provider configurations** section, depending on your provider requirements, choose one of the following options:

   a. **Discovery URL** (recommended) – Choose this option to have AgentCore Identity automatically retrieve configuration details from your provider. You provide the discovery URL where your provider publishes its OpenID Connect configuration, and AgentCore Identity handles the endpoint discovery process. This is the recommended approach when available as it reduces manual configuration.

        i. For **Client ID**, enter the unique identifier you received when registering your application with the identity provider.

        ii. For **Client secret**, enter the confidential key associated with your client ID that AgentCore Identity securely stores for authentication.

        iii. For **Discovery URL**, enter the URL where your provider publishes its OpenID Connect configuration. Discovery URLs must end with `.well-known/openid-configuration`. For example, https://example.com/.well-known/openid-configuration.

    b. **Manual config** – Choose this option to specify server information directly when your provider doesn't support automatic discovery. You'll define each endpoint URL individually, giving you complete control over the configuration details.

        i. For **Client ID**, enter the unique identifier you received when registering your application with the identity provider.

        ii. For **Client secret**, enter the confidential key associated with your client ID that AgentCore Identity securely stores for authentication.

        iii. For **Issuer**, enter the base URL that identifies your authorization server. This value appears in the `iss` claim of issued tokens and helps verify token authenticity.

        iv. For **Authorization endpoint**, enter the URL where users will be directed to grant permission to your application. This is the entry point for the OAuth authorization flow.

        v. For **Token endpoint**, enter the URL where your agent exchanges authorization codes for access tokens. This endpoint handles the credential exchange process.

        vi. (Optional) In the **Response types** section, configure how your OAuth client receives authentication responses by choosing **Add response type** and selecting the token formats your provider should return. Common types include `code` for authorization code flow or `token` for implicit flow.

6. Choose **Add OAuth Client**.

After completing either configuration, AgentCore Identity securely stores your OAuth settings and provides an ARN you can reference in your agent code, enabling token requests without embedding sensitive credentials in your application. You can find this ARN in the properties page of the OAuth client (Choose the client name in the **Outbound Auth** section).

Update OAuth client

You can modify the configuration settings for your existing OAuth client. For example, you can update your client credentials (Client ID and Client secret) when they've been rotated or changed by your identity provider.

To update an OAuth client

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, select the OAuth client you want to update.
3. Choose **Edit**.
4. On the **Update OAuth Client** page, update the information as needed.
5. Choose **Update OAuth Client** to save your configuration settings.

The updated OAuth client configuration takes effect immediately and will be used for all subsequent authentication requests made by your agents.

Delete OAuth client

When you no longer need an OAuth client, you can delete it from your account. Deleting an OAuth client removes the stored configuration and credentials, making them unavailable to your agents. Any invocations that reference the deleted OAuth client will fail once it's removed, and this outbound authentication might be used across multiple runtimes and gateways.

To delete an OAuth client

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, select the OAuth client you want to delete.
3. Choose **Delete**.
4. In the confirmation dialog, type **`Delete`** to confirm the deletion.
5. Choose **Delete**.

The OAuth client is permanently removed from your account. Any agents or applications that reference this OAuth client's ARN will no longer be able to access the stored credentials.

Configure an API key

API keys provide key-based authentication for services that require direct key access with secure storage capabilities. An API key is a unique identifier used to authenticate and authorize access to a resource, enabling your agent to access external services without embedding sensitive credentials directly in your application code.

Topics

Add API key

API keys provide key-based authentication for services that require direct key access with secure storage capabilities. An API key is a unique identifier used to authenticate and authorize access to a resource, enabling your agent to access external services without embedding sensitive credentials directly in your application code.

To add an API key

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, choose **Add OAuth client / API key**, then choose **Add API key**.
3. For **Name**, you can either use the auto-generated name or enter your own descriptive name to help you identify this API key in your account. Use alphanumeric characters, hyphens, and underscores only, with a maximum length of 50 characters.
4. For **API key**, enter the key value provided by your external service. AgentCore Identity securely stores this value and makes it available to your agent at runtime.
5. Choose **Add**.

After creating the API key, AgentCore Identity provides an ARN that you can reference in your agent code to access the stored key without exposing sensitive information in your application. You can find this ARN in the properties page of the API key (Choose the API key name in the **Outbound Auth** section).

Update API key

You can update an existing API key to replace the key value when your external service provider rotates credentials. Updating the API key ensures your agents continue to have access to the external service with the current authentication information.

To update an API key

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, select the API key you want to update.
3. Choose **Edit**.
4. In the **Update API key** dialog, in **API key**, enter the updated key value provided by your external service. AgentCore Identity securely stores this new value and makes it available to your agent at runtime.
5. Choose **Update**.

The updated API key configuration takes effect immediately. Your agents will use the new API key for all subsequent requests to the external service.

Delete API key

When you no longer need an API key, you can delete it from your account. Deleting an API key removes the stored credentials and makes them unavailable to your agents. Any invocations that reference the deleted API key will fail once it's removed.

To delete an API key

1. Open the AgentCore Identity console.
2. In the **Outbound Auth** section, select the API key you want to delete.
3. Choose **Delete**.
4. In the confirmation dialog, type **`Delete`** to confirm the deletion.
5. Choose **Delete**.

The API key is permanently removed from your account. Any agents or applications that reference this API key's ARN will no longer be able to access the stored credentials.

Manage workload identities with AgentCore Identity

Agent identities in AgentCore Identity are implemented as workload identities with specialized attributes that enable agent-specific capabilities. This approach follows established industry patterns where workloads have granular properties that indicate their specific type and purpose. Unlike traditional service accounts that are tied to specific infrastructure, agent identities are designed to be environment-agnostic and can support multiple authentication credentials simultaneously. The AgentCore Identity directory acts as a centralized registry and management system for all agent identities.

Topics

Understanding workload identities

Workload identities represent the digital identity of your agents within the AWS environment. They serve as a stable anchor point that persists across different deployment environments and authentication schemes, allowing agents to maintain consistent identity whether they're using IAM roles for AWS resource access, OAuth2 tokens for external service integration, or API keys for third-party tool access. The identity system abstracts the complexity of managing multiple credential types while providing a unified interface for authentication and authorization operations.

Workload identities integrate seamlessly with the broader AgentCore Identity framework, including the token vault for secure credential storage (see Secure credential storage), Resource credential providers for external service access (see Configure credential provider), and the AgentCore Identity directory for centralized management

Create and configure workload identities

You can create agent identities using several methods, including the AWS CLI and the AgentCore SDK, depending on your workflow and integration requirements. AgentCore Identity provides multiple interfaces for identity creation including command-line tools for automation and scripting and programmatic APIs for integration with existing systems. Each creation method supports the full range of identity configuration options while providing appropriate interfaces for different use cases and user preferences.

Topics

- Manage identities with AWS CLI
- Create identities with the AgentCore SDK

Manage identities with AWS CLI

The AWS CLI provides a straightforward way to create and delete agent identities.

Create an identity:

```
aws bedrock-agentcore-control create-workload-identity \
    --name "my-agent"
```

Delete an identity:

```
aws bedrock-agentcore-control delete-workload-identity \
    --name "my-agent" \
```

Create identities with the AgentCore SDK

The AgentCore SDK provides support for creating workload identities in Python.

Python example:

```
from bedrock_agentcore.services.identity import IdentityClient

# Initialize the client
identity_client= IdentityClient("us-east-1")# Create a new workload identity for agent
response= identity_client.create_workload_identity(name= 'my-python-agent')agentArn=
 response['workloadIdentityArn']

print(f"Created agent identity with ARN: {agentArn}")
```

Manage credential providers with AgentCore Identity

Credential management is a core feature of Amazon Bedrock AgentCore Identity that addresses the complex challenge of securely storing, retrieving, and managing credentials across multiple trust domains and authentication systems. The service implements defense-in-depth security measures to protect sensitive authentication tokens, API keys, and certificates while providing agents with efficient access to the credentials they need for authorized operations. AgentCore Identity's credential management architecture separates credential storage from credential access, helping to ensure that agents never have direct access to long-term secrets or refresh tokens.

The credential management system supports multiple credential types including OAuth2 access tokens, API keys, client certificates, SAML assertions, and custom authentication tokens. Each credential type has specific handling requirements for storage encryption and access patterns. All credential operations are logged and audited to provide complete visibility into credential usage and access patterns.

Integration with the Resource Credential Provider enables AgentCore Identity to support cross-capability credential vending, where agents can access resources across different cloud providers, SaaS applications, and enterprise systems using a unified credential management interface.
The system maintains proper security boundaries while enabling necessary functionality, with comprehensive monitoring and alerting capabilities that detect unusual credential usage patterns or potential security threats.

Topics

Supported authentication patterns

AgentCore Identity supports two primary authentication patterns that address different agent use cases. Understanding these patterns will help you choose the right approach for your specific agent implementation.

For detailed examples of how these patterns apply to specific industries and agent types, see Example use cases.

Topics

User-delegated access (OAuth 2.0 authorization code grant)

The OAuth 2.0 authorization code grant flow enables agents to access user-specific data with explicit user consent. This pattern is essential when agents need to access personal data or perform actions on behalf of specific users. The flow includes a user consent step where the resource owner (user) explicitly authorizes the agent to access their data within specific scopes.

Key characteristics:

- Requires explicit user consent through an authorization prompt
- Provides access to user-specific data and resources
- Maintains clear separation between agent identity and user authorization
- Supports fine-grained scopes that limit what data the agent can access

Example scenario: A productivity agent needs to access a user's Google Calendar to schedule meetings, their Gmail to send emails, and their Google Drive to store documents. The agent uses the OAuth 2.0 authorization code grant to obtain user consent for each service, with specific scopes that limit access to only the necessary data. The user explicitly authorizes the agent through Google's consent screen, and AgentCore Identity securely stores the resulting credentials for future use.

This pattern is ideal for personal assistant agents, customer service agents, and any scenario where agents need access to user-specific data across multiple services. For detailed industry-specific examples, see Personal assistant agents and Customer service agents.

Machine-to-machine authentication (OAuth 2.0 client credentials grant)

The OAuth 2.0 client credentials grant flow enables direct authentication between systems without user interaction. This pattern is appropriate when agents need to access resources that aren't user-specific or when agents act themselves with pre-authorized user consent.

Key characteristics:

- No user interaction or consent required
- Agent authenticates directly with resource servers using its own credentials
- Suitable for background processes, scheduled tasks, and system-level operations
- Permissions are defined at the agent level rather than per-user

Example scenario: An enterprise data processing agent needs to collect data from multiple internal systems, process it, and store the results in a data warehouse. The agent uses the OAuth 2.0 client credentials grant to authenticate directly with each system using its own identity and pre-configured permissions. No user interaction is required, and the agent can operate when agents act themselves with pre-authorized user consent on scheduled intervals.

This pattern is ideal for enterprise automation agents, data processing workflows, and DevOps automation. For detailed industry-specific examples, see Enterprise automation agents, Data processing and analytics agents, and Development and DevOps agents.

Choosing the right authentication pattern

When designing your agent authentication strategy, consider these factors to determine which pattern is most appropriate:

Authentication pattern selection guide

Factor	User-delegated access (OAuth 2.0 authorization code grant)	Machine-to-machine authentic ation (OAuth 2.0 client credentials grant)
Data ownership	User-specific data (emails, documents, personal calendars)	System or organization-owned data (analytics, logs, shared resources)
User interaction	User is present and can provide consent	No user interaction required or available
Operation timing	Interactive, real-time operations	Background, scheduled, or batch operations
Permission scope	Permissions vary by user and their consent choices	Consistent permissions defined at the agent level

Many agent implementations will require both patterns for different aspects of their functionality. For example, a customer service agent might use user-delegated access to retrieve a specific customer's data while using machine-to-machine authentication to access company knowledge bases and internal systems. AgentCore Identity supports both patterns simultaneously, allowing agents to use the most appropriate authentication mechanism for each resource they need to access.

Both authentication patterns benefit from AgentCore Identity's core capabilities:

- Secure credential storage without exposing secrets to agent code
- Consistent authentication interfaces across multiple resource types
- Comprehensive audit logging for security and compliance
- Fine-grained access controls based on identity and context
- Simplified integration through the AgentCore SDK

Configure credential provider

Resource credential providers in AgentCore Identity act as intelligent intermediaries that manage the complex relationships between agents, identity providers, and resource servers. Each provider encapsulates the specific endpoint configuration required for a particular service or identity system. The service provides built-in providers for popular services including Google, GitHub, Slack, and Salesforce, with authorization server endpoints and provider-specific parameters pre-configured to reduce development effort. AgentCore Identity supports custom configurations through configurable OAuth2 credential providers that can be tailored to work with any OAuth2-compatible resource server.

Resource credential providers integrate deeply with the token vault to provide seamless credential lifecycle management. When an agent requests access to a resource, the provider handles the authentication flow, stores the resulting credentials in the token vault, and provides the agent with the necessary access tokens.

Creating an OAuth 2.0 credential provider

Provider configurations in AgentCore Identity define the basic parameters needed for credential management with different resources and authentication systems. The following example demonstrates how to use the AgentCore SDK to configure an OAuth 2.0 credential provider to use with GitHub.

```
from bedrock_agentcore.services.identity import IdentityClient
identity_client = IdentityClient("us-east-1")
github_provider = identity_client.create_oauth2_credential_provider({
    "name": "github-provider",
    "credentialProviderVendor": "GithubOauth2",
    "oauth2ProviderConfigInput": {
      "githubOauth2ProviderConfig": {
```

```
    "clientId": "your-github-client-id",
    "clientSecret": "your-github-client-secret",
   }
  },
 })
```

Creating an API key credential provider

For services that use API keys for authentication rather than OAuth, AgentCore Identity will securely store and retrieve keys for your agents. The example below illustrates using the AgentCore SDK to store an API key.

```
from bedrock_agentcore.services.identity import IdentityClient
identity_client= IdentityClient("us-east-1")apikey_provider=
 identity_client.create_api_key_credential_provider({
  "name": "your-service-name",
  "apiKey": "your-api-key"
 })
```

Obtain credentials

AgentCore Identity uses a workload access token to authorize agent access to credentials stored in the vault, and this token contains both the identity of the agent and the identity of the end user on whose behalf the agent is working. AgentCore Runtime will automatically provide a token when invoking an agent that it is hosting. Agents hosted on other systems can retrieve their agent token using the AgentCore SDK.

Topics

Get workload access token

There are two patterns to use to retrieve the workload access token depending on how you are able to identify the end user of the agent:

- If the agent's caller has a JWT identifying the end user, request a workload access token based on the agent's identity and the end-user JWT. When you provide a JWT, AgentCore Identity will validate the JWT to ensure it is correctly signed and unexpired, and it will use its “iss” and “sub” claims to uniquely identify the user. Credentials stored by the agent on behalf of the user will be associated with this information, and future retrievals by the agent will require a valid workload access token containing the same information.

- If the agent’s caller does not have a JWT identifying the end user, request a workload access token based on the agent’s identity and a unique string identifying the user.

The examples below illustrate using the AgentCore SDK to retrieve a workload access token using these two methods:

```
from bedrock_agentcore.services.identity import IdentityClient

identity_client= IdentityClient("us-east-1")# Obtain a token using the IAM identity of
 the caller to authenticate the agent and providing a JWT containing the identity of
 the end user.
# This is the recommended pattern whenever a JWT is available for the user.
workload_access_token= identity_client.get_workload_access_token(workload_name= "my-
demo-agent", user_token= "insert-jwt-here")# Obtain a token using the IAM identity of
 the caller to authenticate the agent and providing a string representing the identity
 of the end user.
# Use this pattern when a JWT is not available for the user.
workload_access_token= identity_client.get_workload_access_token(workload_name= "my-
demo-agent", user_id= "insert-user-name-or-identifier")
```

Obtain OAuth 2.0 access token

AgentCore Identity enables developers to obtain OAuth tokens for either user-delegated access or machine-to-machine authentication based on the configured OAuth 2.0 credential providers. The service will orchestrate the authentication process between the user or application to the downstream authorization server, and it will retrieve and store the resulting token. Once the token is available in the AgentCore Identity vault, authorized agents can retrieve it and use it to authorize calls to resource servers. For example, the sample code below will retrieve a token to interact with Google Drive on behalf of an end user. For more information, see Getting started with Amazon Bedrock AgentCore Identity for the complete example.

```
# Injects Google Access Token
@requires_access_token(# Uses the same credential provider name created above
 #provider_name= "google-provider",
##### Requires Google OAuth2 scope to access Google Drive
```

```
 #scopes= ["https://www.googleapis.com/auth/drive.metadata.readonly"],
##### Sets to OAuth 2.0 Authorization Code flow
 #auth_flow= "USER_FEDERATION",
##### Prints authorization URL to console
 #on_auth_url= lambda x: print("\nPlease copy and paste this URL in your browser:\n" +
 x),
  # If false, caches obtained access token
 #force_authentication= False,)async#def#write_to_google_drive(*, access_token: str):
  # Use the token to call Google Drive
asyncio.run(write_to_google_drive(access_token= ""))
```

The process is similar to obtain a token for machine-to-machine calls, as shown in the following example:

```
import#asyncio
from#bedrock_agentcore.identity.auth#import#requires_access_token, requires_api_key
@requires_access_token(provider_name= "my-api-key-provider", # replace with your own
 credential provider name
 #scopes= [],
 #auth_flow= 'M2M',)async#def#need_token_2LO_async(*, access_token: str):
 ## Use the access token
asyncio.run(need_token_2LO_async(access_token= ""))
```

Obtain API key

Once you have stored your API keys in the AgentCore Identity vault, you can retrieve them directly in your agent using the AgentCore SDK and the @requires_api_key annotation. For example, the code below will retrieve the API key from the “your-service-name” API key provider so that you can use it in the need_api_key function.

```
import asyncio
from bedrock_agentcore.identity.auth import requires_api_key
@requires_api_key(provider_name= " your-service-name" # replace with your own
 credential provider name)async def need_api_key(*, api_key: str):
 ## Use the key in api_key
asyncio.run(need_api_key(api_key= ""))
```

Identity provider setup and configuration

Amazon Bedrock AgentCore Identity provides managed OAuth 2.0 supported providers for both inbound and outbound authentication. Each provider encapsulates the specific authentication protocols, endpoint configurations, and credential formats required for a particular service or identity system. The service provides built-in providers for popular services including Google, GitHub, Slack, and Salesforce with authorization server endpoints and provider-specific parameters pre-configured to reduce development effort. The providers abstract away the complexity of different OAuth 2.0 implementations, API authentication schemes, and token formats, presenting a unified interface to agents while handling the underlying protocol variations and edge cases.

Built-in providers are maintained by the AgentCore Identity team and automatically updated to handle changes in external service APIs, security requirements, and best practices.

Supported identity providers include:

Topics

Amazon Cognito

To add Cognito as an identity provider for accessing AgentCore Gateway and Runtime, you must:

- Configure discovery URL from your IDP directory. This helps AgentCore Identity get the metadata related to your OAuth authorization server and token verification keys.
- Provide valid `clientId` or `aud` claims for the token. This helps validate the tokens coming from your IDP and allow access for tokens that contain expected claims.

Amazon Cognito can be used as an identity provider for inbound authentication.

Inbound

To create a Cognito user pool as an inbound identity provider for user authentication with AgentCore Runtime

Create a file named `setup_cognito.sh` with the following content:

```
#!/bin/bash

# Create User Pool and capture Pool ID directly
export POOL_ID= $(aws cognito-idp create-user-pool \
  --pool-name "MyUserPool" \
  --policies '{"PasswordPolicy":{"MinimumLength":8}}' \
  --region us-east-1 | jq -r '.UserPool.Id')# Create App Client and capture Client ID
 directly
export CLIENT_ID= $(aws cognito-idp create-user-pool-client \
  --user-pool-id $POOL_ID \
  --client-name "MyClient" \
  --no-generate-secret \
  --explicit-auth-flows "ALLOW_USER_PASSWORD_AUTH" "ALLOW_REFRESH_TOKEN_AUTH" \
  --region us-east-1 | jq -r '.UserPoolClient.ClientId')# Create User
aws cognito-idp admin-create-user \
  --user-pool-id $POOL_ID \
  --username "testuser" \
  --temporary-password "Temp123!" \
  --region us-east-1 \
  --message-action SUPPRESS > /dev/null

# Set Permanent Password
aws cognito-idp admin-set-user-password \
  --user-pool-id $POOL_ID \
  --username "testuser" \
  --password "MyPassword123!" \
  --region us-east-1 \
  --permanent > /dev/null

# Authenticate User and capture Access Token
export BEARER_TOKEN= $(aws cognito-idp initiate-auth \
  --client-id "$CLIENT_ID" \
  --auth-flow USER_PASSWORD_AUTH \
  --auth-parameters USERNAME= 'testuser',PASSWORD= 'MyPassword123!' \
  --region us-east-1 | jq -r '.AuthenticationResult.AccessToken')# Output the required
 values
```

```
echo "Pool id: $POOL_ID"
echo "Discovery URL: https://cognito-idp.us-east-1.amazonaws.com/$POOL_ID/.well-known/
openid-configuration"
echo "Client ID: $CLIENT_ID"
echo "Bearer Token: $BEARER_TOKEN"
```

Run the script to create the Cognito resources:

```
source setup_cognito.sh
```

Note the output values, which will look similar to:

```
Pool id: us-east-1_poolid
Discovery URL: https://cognito-idp.us-east-1.amazonaws.com/us-east-1_userpoolid/.well-
known/openid-configuration
Client ID: clientid
Bearer Token: bearertoken
```

You'll need these values in the next steps.

This script creates a Cognito user pool, a user pool client, adds a user, and generates a bearer token for the user. The token is valid for 60 minutes by default.

To create a Cognito user pool as an inbound identity provider for machine-to-machine authentication with AgentCore Gateway

1. Create a user pool:

   ```
   aws cognito-idp create-user-pool \
     --region us-west-2 \
     --pool-name "gateway-user-pool"
   ```

2. Note the user pool ID from the response or retrieve it using:

   ```
   aws cognito-idp list-user-pools \
     --region us-west-2 \
     --max-results 60
   ```

3. Create a resource server for the user pool:

   ```
   aws cognito-idp create-resource-server \
     --region us-west-2 \
   ```

```
  --user-pool-id <UserPoolId> \
  --identifier "gateway-resource-server" \
  --name "GatewayResourceServer" \
  --scopes '[{"ScopeName":"read","ScopeDescription":"Read access"},
 {"ScopeName":"write","ScopeDescription":"Write access"}]'
```

4. Create a client for the user pool:

```
aws cognito-idp create-user-pool-client \
  --region us-west-2 \
  --user-pool-id <UserPoolId> \
  --client-name "gateway-client" \
  --generate-secret \
  --allowed-o-auth-flows client_credentials \
  --allowed-o-auth-scopes "gateway-resource-server/read" "gateway-resource-server/
write" \
  --allowed-o-auth-flows-user-pool-client \
  --supported-identity-providers "COGNITO"
```

   Note the client ID and client secret from the response.

5. If needed, create a domain for your user pool:

```
aws cognito-idp create-user-pool-domain \
  --domain <UserPoolIdWithoutUnderscore> \
  --user-pool-id <UserPoolId> \
  --region us-west-2
```

> **Note**
>
> Remove any underscore from the `UserPoolId` when creating the domain. For example, if your user pool ID is "us-west-2_gmSGKKGr9", use "us-west-2gmSGKKGr9" as the domain.

6. Construct the discovery URL for your Cognito user pool:

```
https://cognito-idp.us-west-2.amazonaws.com/<UserPoolId>/.well-known/openid-
configuration
```

7. Configure the Gateway Inbound Auth with the following values:

   - **Discovery URL**: The URL constructed in the previous step

- **Allowed clients**: The client ID obtained when creating the user pool client

Microsoft

Microsoft can be set up as an inbound provider using Microsoft Entra ID or as an outbound provider.

To add Microsoft Entra ID as an identity provider for accessing AgentCore Gateway and Runtime, you must:

- Configure discovery URL from your IDP directory. This helps AgentCore Identity get the metadata related to your OAuth authorization server and token verification keys.
- Provide valid `clientId` or `aud` claims for the token. This helps validate the tokens coming from your IDP and allow access for tokens that contain expected claims.

You can configure these as part of configuration of Gateway and Runtime inbound configuration.

Inbound

We support Microsoft Entra ID for v2.0 Id Tokens.

Configurations for v2.0 Id Tokens

In custom authorizer:

- **Discovery URL**: Discovery URL should be of the form: `https://login.microsoftonline.com/${tenantId}/v2.0/.well-known/openid-configuration`
- **Allowed audiences**: `aud` should be the Application Id

On Microsoft Entra:

- While configuring the application, Enable ID Token Issuance in Application Registration.
- Include mandatory `openid` scope while calling the authorize and token endpoint for Microsoft Entra Id during Ingress Flows.

Outbound

To configure the outbound Microsoft resource provider, use the following:

```
{
        "name": "NAME",
        "credentialProviderVendor": "MicrosoftOAuth2",
        "oauth2ProviderConfigInput": {
            "microsoftOauth2ProviderConfig": {
                "clientId": "your-client-id",
                "clientSecret": "your-client-secret",
            }
        },
    }
```

Auth0 by Okta

Auth0 by Okta can be set up as an inbound provider or as an outbound provider.

To add Auth0 as an identity provider for accessing AgentCore Gateway and Runtime, you must:

- Configure discovery URL from your IDP directory. This helps AgentCore Identity get the metadata related to your OAuth authorization server and token verification keys.
- Provide valid aud claims for the token. This helps validate the tokens coming from your IDP and allows access for tokens that contain expected claims.

Inbound

Follow these steps to set up Auth0 and obtain the necessary configuration values for Gateway authentication:

1. Create an API in Auth0:
   - Log in to your Auth0 dashboard.
   - Navigate to "APIs" and click "Create API".
   - Provide a name and identifier for your API (e.g., "gateway-api").
   - Select the signing algorithm (RS256 recommended).
   - Click "Create".
2. Configure API scopes:

- In the API settings, go to the "Scopes" tab.
- Add scopes such as "invoke:gateway" and "read:gateway".

3. Create an application:
   - Navigate to "Applications" and click "Create Application".
   - Select "Machine to Machine Application".
   - Select the API you created in step 1.
   - Authorize the application for the scopes you created.
   - Click "Create".
4. Note the client ID and client secret from the application settings.
5. Construct the discovery URL for your Auth0 tenant:

   ```
   https://<your-domain>/.well-known/openid-configuration
   ```

   Where `<your-domain>` is your Auth0 tenant domain (e.g., "dev-example.us.auth0.com").
6. Configure Inbound Auth with the following values:
   - **Discovery URL**: The URL constructed in the previous step
   - **Allowed audiences**: The API identifier you created in step 1

GitHub

Outbound

To configure the outbound GitHub resource provider, use the following:

```
{
        "name": "NAME",
        "credentialProviderVendor": "GithubOauth2",
        "oauth2ProviderConfigInput": {
            "GithubOauth2ProviderConfigInput": {
                "clientId": "your-client-id",
                "clientSecret": "your-client-secret",
            }
        },
    }
```

Google

Outbound

To configure the outbound Google resource provider, use the following:

```
{
        "name": "NAME",
        "credentialProviderVendor": "GoogleOauth2",
        "oauth2ProviderConfigInput": {
            "GoogleOauth2ProviderConfigInput": {
                "clientId": "your-client-id",
                "clientSecret": "your-client-secret",
            }
        },
    }
```

Salesforce

Outbound

To configure the outbound Salesforce resource provider, use the following:

```
{
        "name": "NAME",
        "credentialProviderVendor": "SalesforceOauth2",
        "oauth2ProviderConfigInput": {
            "SalesforceOauth2ProviderConfigInput": {
                "clientId": "your-client-id",
                "clientSecret": "your-client-secret",
            }
        },
    }
```

Slack

Outbound

To configure the outbound Slack resource provider, use the following:

```
{
        "name": "NAME",
        "credentialProviderVendor": "SlackOauth2",
        "oauth2ProviderConfigInput": {
            "SlackOauth2ProviderConfigInput": {
                "clientId": "your-client-id",
                "clientSecret": "your-client-secret",
            }
        },
    }
```

Data protection in Amazon Bedrock AgentCore Identity

The AWS shared responsibility model applies to data protection in Amazon Bedrock AgentCore Identity. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with IAM. That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Amazon Bedrock AgentCore Identity or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Amazon Bedrock AgentCore Identity or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

Topics

Data encryption

Data encryption typically falls into two categories: encryption at rest and encryption in transit.

Encryption at rest

Data within Amazon Bedrock AgentCore Identity is encrypted at rest in accordance with industry standards.

By default, Amazon Bedrock AgentCore Identity encrypts customer data in token vaults with AWS owned keys. You can also configure your token vaults to instead encrypt your information with customer managed keys.

AWS owned key

> Amazon Bedrock AgentCore Identity encrypts the data in your token vault with an AWS owned KMS key. Keys of this type aren't visible in AWS KMS.

Customer managed key

> Amazon Bedrock AgentCore Identity encrypts the data in your token vault with a customer managed key. You own the administration of customer managed key policies, rotation, and scheduled deletion.

Things to know about token vault encryption with customer managed keys

- Data in your token vault (access tokens) are encrypted at rest with the customer managed key you configure. The token vault ARN is captured in the EncryptionContext.
- All customer data in your token vault is encrypted at rest, even if you take no action to configure encryption settings.
- You can't configure token vault encryption at rest with multi-Region keys or asymmetric keys. Amazon Bedrock AgentCore Identity supports only single-region symmetric KMS keys for token vault encryption at rest.

- You can configure token vault encryption only with a KMS key ARN, not an alias.
- You can configure CMK for credential provider secrets using AWS Secrets Manager. Learn more.

The following procedures configure encryption at rest in your token vault. For more information about KMS key policies that delegate access to AWS services like Amazon Cognito, see Permissions for AWS services in key policies.

Set customer managed key policy

Note

Currently we don't support configuring CMK on token vault through console.

To use a customer managed key, your key must trust an Amazon Bedrock AgentCore Identity service principal to perform encryption and decryption operations on the key. Configure the key policy of your KMS key as shown in the following example. The IAM principal that writes this policy must have write access to your KMS key, with `kms:PutKeyPolicy` permission.

Configure with API operations or an AWS SDK

Set your key configuration in a `SetTokenVaultCMK` API request. The following partial example request body sets the token vault to use the provided customer managed key.

```
"KmsConfiguration": {
    "KeyType": "CUSTOMER_MANAGED_KEY",
    "KmsKeyArn": "arn:aws:kms:us-east-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE22222"
}
```

The following partial example request body sets a token vault to use an AWS owned key.

```
"KmsConfiguration": {
    "KeyType": "AWS_OWNED_KEY"
}
```

If your `GetTokenVault` response doesn't include a `KmsConfiguration` parameter, your token vault is configured to encrypt data at rest with an AWS owned key.

Security in Amazon Bedrock AgentCore

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are are designed to help meet with the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Bedrock AgentCore, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using AgentCore. The following topics show you how to configure AgentCore to help meet your security and compliance objectives You also learn how to use other AWS services that help you to monitor and secure your AgentCore resources.

Topics

Data protection in Amazon Bedrock AgentCore

The AWS shared responsibility model applies to data protection in Amazon Bedrock AgentCore. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with AgentCore or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Topics

- Protecting data using VPC and AWS PrivateLink

Protecting data using VPC and AWS PrivateLink

You can use Amazon Virtual Private Cloud (Amazon VPC) and AWS PrivateLink to create private connections between your VPC and Amazon Bedrock AgentCore. This section covers two main approaches:

- **Interface VPC endpoints** - Create a private connection between your VPC and Amazon Bedrock AgentCore using AWS PrivateLink
- **VPC connectivity for Amazon Bedrock AgentCore Runtime** - Configure Amazon Bedrock AgentCore Runtime to access resources within your VPC

Use interface VPC endpoints (AWS PrivateLink) to create a private connection between your VPC and your AgentCore resources

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Bedrock AgentCore. You can access AgentCore as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to access AgentCore.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for AgentCore.

For more information, see Access AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Considerations for AgentCore

Before you set up an interface endpoint for AgentCore, review Considerations in the *AWS PrivateLink Guide*.

AgentCore supports the following through interface endpoints:

- Data plane operations (runtime APIs)
- Invoking gateways

Note

Privatelink is currently not supported for Amazon Bedrock AgentCore control plane endpoints.

AgentCore interface endpoints are available in the following AWS Regions:

- US East (N. Virginia)
- US West (Oregon)
- Europe (Frankfurt)
- Asia Pacific (Sydney)

Authorization considerations for data plane APIs

The data plane APIs support both AWS Signature Version 4 (SigV4) headers for authentication and Bearer Token (OAuth) authentication. VPC endpoint policies can only restrict callers based on IAM principals and not OAuth users. For OAuth-based requests to succeed through the VPC endpoint, the principal must be set to `*` in the endpoint policy. Otherwise, only SigV4 allowlisted callers can make successful calls over the VPC endpoint.

AWS IAM global condition context keys are supported. By default, full access to AgentCore is allowed through the interface endpoint. You can control access by attaching an endpoint policy to the interface endpoint or by associating a security group with the endpoint network interfaces.

Create an interface endpoint for AgentCore

You can create an interface endpoint for AgentCore using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

Create an interface endpoint for AgentCore using the following service name format:

- Data plane operations: `com.amazonaws.`*`region`*`.bedrock-agentcore`
- For AgentCore Gateway : `com.amazonaws.`*`region`*`.bedrock-agentcore.gateway`

If you enable private DNS for the interface endpoint, you can make API requests to AgentCore using its default Regional DNS name. For example, `bedrock-agentcore.us-east-1.amazonaws.com`.

Create an endpoint policy for your interface endpoint

An endpoint policy is an IAM resource that you can attach to an interface endpoint. The default endpoint policy allows full access to AgentCore through the interface endpoint. To control the access allowed to AgentCore from your VPC, attach a custom endpoint policy to the interface endpoint.

An endpoint policy specifies the following information:

- The principals that can perform actions (AWS accounts, IAM users, and IAM roles).
  - For AgentCore Gateway , if your gateway ingress isn't AWS Signature Version 4 (SigV4)-based (for example, if you use OAuth instead), you must specify the `Principal` field as the wildcard `*`. SigV4 -based authentication allows you to define the `Principal` as a specific AWS identity.
- The actions that can be performed.
- The resources on which the actions can be performed.

For more information, see Control access to services using endpoint policies in the *AWS PrivateLink Guide*.

Endpoint policies for various primitives

The following examples show endpoint policies for different AgentCore components:

Runtime

The following endpoint policy allows specific IAM principals to invoke agent runtime resources.

```
{
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws::iam::ACCOUNT_ID:user/USERNAME"
         },
         "Action": [
            "bedrock-agentcore:InvokeAgentRuntime"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-
east-1:ACCOUNT_ID:runtime/RUNTIME_ID"
      }
   ]
}
```

Mixed IAM and OAuth authentication

The `InvokeAgentRuntime` API supports two modes of VPC endpoint authorization. The following example policy allows both IAM principals and OAuth callers to access different agent runtime resources.

```
{
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws::iam::ACCOUNT_ID:root"
         },
         "Action": [
            "bedrock-agentcore:InvokeAgentRuntime"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-
east-1:ACCOUNT_ID:runtime/customAgent1"
      },
      {
         "Effect": "Allow",
         "Principal": "*",
         "Action": [
            "bedrock-agentcore:InvokeAgentRuntime"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-
east-1:ACCOUNT_ID:runtime/customAgent2"
      }
   ]
}
```

The above policy allows only the IAM principal to make `InvokeAgentRuntime` calls to `customAgent1`. It also allows both IAM principals and OAuth callers to make `InvokeAgentRuntime` calls to `customAgent2`.

Code Interpreter

The following endpoint policy allows specific IAM principals to invoke Code Interpreter resources.

```
{
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws::iam::ACCOUNT_ID:root"
         },
         "Action": [
            "bedrock-agentcore:InvokeCodeInterpreter"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-east-1:ACCOUNT_ID:code-
interpreter/CODE_INTERPRETER_ID"
      }
   ]
}
```

Browser Tool

The following endpoint policy allows specific IAM principals to connect to Browser Tool resources.

```
{
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws::iam::ACCOUNT_ID:root"
         },
         "Action": [
            "bedrock-agentcore:ConnectBrowserAutomationStream"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-
east-1:ACCOUNT_ID:browser/BROWSER_ID"
      }
   ]
}
```

Gateway

The following is an example of a custom endpoint policy. When you attach this policy to your interface endpoint, it allows all principals to invoke the gateway specified in the `Resource` field.

```
{
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": "*",
         "Action": [
            "*"
         ],
         "Resource": "arn:aws::bedrock-agentcore:us-east-1::gateway/my-gateway"
      }
   ]
}
```

Configuring Amazon Bedrock AgentCore Runtime and tools for VPC

You can configure Amazon Bedrock AgentCore Runtime and built-in tools (Code Interpreter and Browser Tool) to connect to resources in your Amazon Virtual Private Cloud (VPC). By configuring VPC connectivity, you enable secure access to private resources such as databases, internal APIs, and services within your VPC.

VPC connectivity for Amazon Bedrock AgentCore Runtime and tools

To enable Amazon Bedrock AgentCore Runtime and built-in tools to securely access resources in your private VPC, Amazon Bedrock AgentCore provides VPC connectivity capabilities. This feature allows your runtime and tools to:

- Connect to private resources without exposing them to the internet
- Maintain secure communications within your organization's network boundaries
- Access enterprise data stores and internal services while preserving security

When you configure VPC connectivity for Amazon Bedrock AgentCore Runtime and tools:

- Amazon Bedrock creates elastic network interfaces (ENIs) in your VPC using the service-linked role `AWSServiceRoleForBedrockAgentCoreNetwork`
- These ENIs enable your Amazon Bedrock AgentCore Runtime and tools to securely communicate with resources in your VPC
- Each ENI is assigned a private IP address from the subnets you specify
- Security groups attached to the ENIs control which resources your runtime and tools can communicate with

Prerequisites

Before configuring Amazon Bedrock AgentCore Runtime and tools for VPC access, ensure you have:

- An Amazon VPC with appropriate subnets for your runtime and tool requirements. For example, to configure your subnets to have internet access, see Internet access considerations.
- Subnets located in supported Availability Zones for your region. For information about supported Availability Zones, see Supported Availability Zones.
- Appropriate security groups defined in your VPC for runtime and tool access patterns. For example, to configure your security groups to connect to Amazon RDS, see Example: Connecting to an Amazon RDS database.
- Required IAM permissions to create and manage the service-linked role (already included in the AWS managed policy BedrockAgentCoreFullAccess). For information about required permissions, see IAM permissions.
- Required VPC endpoints if your VPC doesn't have internet access. For example, to configure your VPC endpoints, see VPC endpoint configuration.
- Understanding of your runtime and tool network requirements (databases, APIs, web resources). If you need to use Browser tool which requires internet access, then your VPC should have internet access through NAT Gateway. For example, see Security group considerations.

> ⚠ **Important**
>
> Amazon Bedrock AgentCore creates a network interface in your account with a private IP address. Using a public subnet does not provide internet connectivity. To enable internet access, place it in private subnets with a route to a NAT Gateway.

Supported Availability Zones

Amazon Bedrock AgentCore supports VPC connectivity in specific Availability Zones within each supported region. When configuring subnets for your Amazon Bedrock AgentCore Runtime and built-in tools, ensure that your subnets are located in the supported Availability Zones for your region.

The following table shows the supported Availability Zone IDs for each region:

Region	Region Code	Supported Availability Zones
US West (Oregon)	us-west-2	• usw2-az1 (us-west-2a) • usw2-az2 (us-west-2b) • usw2-az3 (us-west-2c)
US East (N. Virginia)	us-east-1	• use1-az1 (us-east-1a) • use1-az2 (us-east-1b) • use1-az4 (us-east-1d)
Europe (Frankfurt)	eu-central-1	• euc1-az1 (eu-central-1a) • euc1-az2 (eu-central-1b) • euc1-az3 (eu-central-1c)
Asia Pacific (Sydney)	ap-southeast-2	• apse2-az1 (ap-southeast-2a) • apse2-az2 (ap-southeast-2b) • apse2-az3 (ap-southeast-2c)

> ⚠ **Important**
>
> Subnets must be located in the supported Availability Zones listed above. If you specify subnets in unsupported Availability Zones, the configuration will fail during resource creation.

To identify the Availability Zone ID of your subnets, you can use the following CLI command:

```
aws ec2 describe-subnets --subnet-ids subnet-12345678 --query
 'Subnets[0].AvailabilityZoneId'
```

IAM permissions

Amazon Bedrock AgentCore uses the service-linked role `AWSServiceRoleForBedrockAgentCoreNetwork` to create and manage network interfaces in your VPC. This role is automatically created when you first configure Amazon Bedrock AgentCore Runtime or AgentCore built-in tools to use VPC connectivity.

If you need to create this role manually, your IAM entity needs the following permissions:

```
{
    "Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/network.bedrock-
agentcore.amazonaws.com/AWSServiceRoleForBedrockAgentCoreNetwork",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName": "network.bedrock-agentcore.amazonaws.com"
        }
    }
}
```

This permission is already included in the AWS managed policy BedrockAgentCoreFullAccess.

Best practices

For optimal performance and security with VPC-connected Amazon Bedrock AgentCore Runtime and built-in tools:

- **High Availability**:
  - Configure at least two private subnets in different Availability Zones. For a list of supported Availability Zones, see Supported Availability Zones.
  - Deploy dependent resources (such as databases or caches) with multi-AZ support to avoid single points of failure.
- **Network Performance**:
  - Place Amazon Bedrock AgentCore Runtime or built-in tools subnets in the same Availability Zones as the resources they connect to. This reduces cross-AZ latency and data transfer costs.

  - Use VPC endpoints for AWS services whenever possible. Endpoints provide lower latency, higher reliability, and avoid NAT gateway charges for supported services.
- **Security**:
  - Apply the principle of least privilege when creating security group rules.
  - Enable VPC Flow Logs for auditing and monitoring. Review logs regularly to identify unexpected traffic patterns.
- **Internet Access**:
  - To provide internet access from Amazon Bedrock AgentCore Runtime or built-in tools inside a VPC, configure a NAT gateway in a public subnet. Update the route table for private subnets to send outbound traffic (0.0.0.0/0) to the NAT gateway.
  - We recommend using VPC endpoints for AWS services instead of internet routing to improve security and reduce costs.

Configuring VPC access for runtime and tools

You can configure VPC access for Amazon Bedrock AgentCore Runtime and built-in tools using the AWS Management Console, AWS CLI, or AWS SDKs.

Runtime configuration

AWS Management Console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Navigate to the AgentCore section
3. Select or create an Amazon Bedrock AgentCore Runtime configuration
4. Choose your ECR image
5. Under the Network configuration section, choose **VPC**
6. Select your VPC from the dropdown list
7. Select the appropriate subnets for your application needs
8. Select one or more security groups to apply to the ENIs
9. Save your configuration

AWS CLI

```
aws bedrock-agentcore create-runtime \
  --runtime-name "MyAgentRuntime" \
  --network-configuration '{
      "networkMode": "VPC",
      "networkModeConfig": {
        "subnets": ["subnet-0123456789abcdef0", "subnet-0123456789abcdef1"],
        "securityGroups": ["sg-0123456789abcdef0"]
      }
    }'
```

AWS SDK (Python)

```
import boto3

client = boto3.client('bedrock-agentcore')

response = client.create_runtime(
    runtimeName='MyAgentRuntime',
    networkConfiguration={
        'networkMode': 'VPC',
        'networkModeConfig': {
            'subnets': ['subnet-0123456789abcdef0', 'subnet-0123456789abcdef1'],
            'securityGroups': ['sg-0123456789abcdef0']
        }
    }
)
```

Code Interpreter configuration

AWS Management Console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. Navigate to AgentCore → Built-in Tools → Code Interpreter
3. Select **Create Code Interpreter** or modify existing configuration
4. Provide a tool name (optional)
5. Configure execution role with necessary permissions

6. Under Network configuration, choose **VPC**
7. Select your VPC from the dropdown
8. Choose appropriate subnets (recommend private subnets across multiple AZs with NAT gateway)
9. Select security groups for ENI access control
10. Configure execution role with necessary permissions
11. Save your configuration

AWS CLI

```
aws bedrock-agentcore-control create-code-interpreter \
  --region <Region> \
  --name "my-code-interpreter" \
  --description "My Code Interpreter with VPC mode for data analysis" \
  --execution-role-arn "arn:aws:iam::123456789012:role/my-execution-role" \
  --network-configuration '{
    "networkMode": "VPC",
    "networkModeConfig": {
      "subnets": ["subnet-0123456789abcdef0", "subnet-0123456789abcdef1"],
      "securityGroups": ["sg-0123456789abcdef0"]
    }
  }'
```

AWS SDK (Python)

```
import boto3

# Initialize the boto3 client
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore-control.<Region>.amazonaws.com"
)

# Create a Code Interpreter
response = cp_client.create_code_interpreter(
    name="myTestVpcCodeInterpreter",
    description="Test code sandbox for development",
    executionRoleArn="arn:aws:iam::123456789012:role/my-execution-role",
    networkConfiguration={
        'networkMode': 'VPC',
        'networkModeConfig': {
            'subnets': ['subnet-0123456789abcdef0', 'subnet-0123456789abcdef1'],
            'securityGroups': ['sg-0123456789abcdef0']
        }
    }
)

# Print the Code Interpreter ID
code_interpreter_id = response["codeInterpreterId"]
print(f"Code Interpreter ID: {code_interpreter_id}")
```

Browser Tool configuration

AWS Management Console

1. Open the AgentCore console at https://console.aws.amazon.com/bedrock-agentcore/home#.
2. In the navigation pane, choose **Built-in tools**
3. Choose **Create Browser tool**
4. Provide a tool name (optional) and description (optional)
5. Set execution role permissions
6. Under the Network configuration section, choose **VPC** mode
7. Select your VPC and subnets
8. Configure security groups for web access requirements
9. Set execution role permissions
10. Save your configuration

AWS CLI

```
aws bedrock-agentcore-control create-browser \
  --region <Region> \
  --name "my-browser" \
  --description "My browser for web interaction" \
  --network-configuration '{
    "networkMode": "VPC",
    "networkModeConfig": {
      "subnets": ["subnet-0123456789abcdef0", "subnet-0123456789abcdef1"],
      "securityGroups": ["sg-0123456789abcdef0"]
    }
  }' \
  --recording '{
    "enabled": true,
    "s3Location": {
      "bucket": "my-bucket-name",
      "prefix": "sessionreplay"
    }
  }' \
  --execution-role-arn "arn:aws:iam::123456789012:role/my-execution-role"
```

AWS SDK (Python)

```
import boto3

# Initialize the boto3 client
cp_client = boto3.client(
    'bedrock-agentcore-control',
    region_name="<Region>",
    endpoint_url="https://bedrock-agentcore-control.<Region>.amazonaws.com"
)

# Create a Browser
response = cp_client.create_browser(
    name="myTestVpcBrowser",
    description="Test browser with VPC mode for development",
    networkConfiguration={
        'networkMode': 'VPC',
        'networkModeConfig': {
            'subnets': ['subnet-0123456789abcdef0', 'subnet-0123456789abcdef1'],
            'securityGroups': ['sg-0123456789abcdef0']
        }
    },
    executionRoleArn="arn:aws:iam::123456789012:role/Sessionreplay",
    recording={
        "enabled": True,
        "s3Location": {
            "bucket": "session-record-123456789012",
            "prefix": "replay-data"
        }
    }
)
```

Security group considerations

Security groups act as virtual firewalls for your Amazon Bedrock AgentCore Runtime or built-in tool when connected to a VPC. They control inbound and outbound traffic at the instance level. To configure security groups for your runtime:

- **Outbound rules** – Define outbound rules to allow your Amazon Bedrock AgentCore Runtime to connect to required VPC resources.
- **Inbound rules** – Ensure that the target resource's security group allows inbound connections from the security group associated with your Amazon Bedrock AgentCore Runtime.
- **Least privilege** – Apply the principle of least privilege by allowing only the minimum required traffic.

Example: Connecting to an Amazon RDS database

When your Amazon Bedrock AgentCore Runtime connects to an Amazon RDS database, configure the security groups as follows:

Amazon Bedrock AgentCore Runtime security group

- **Outbound** – Allow TCP traffic to the RDS database's security group on port 3306 (MySQL).
- **Inbound** – Not required. The runtime only initiates outbound connections.

Amazon RDS database security group

- **Inbound** – Allow TCP traffic from the Amazon Bedrock AgentCore Runtime security group on port 3306.
- **Outbound** – Not required. Return traffic is automatically allowed because security groups are stateful.

VPC endpoint configuration

When running Amazon Bedrock AgentCore Runtime in a private VPC without internet access, you must configure the following VPC endpoints to ensure proper functionality:

Required VPC endpoints

- **Amazon ECR Requirements**:
  - Docker endpoint: `com.amazonaws.`*`region`*`.ecr.dkr`
  - ECR API endpoint: `com.amazonaws.`*`region`*`.ecr.api`
- **Amazon S3 Requirements**:
  - Gateway endpoint for ECR docker layer storage: `com.amazonaws.`*`region`*`.s3`
- **CloudWatch Requirements**:
  - Logs endpoint: `com.amazonaws.`*`region`*`.logs`

> **Note**
>
> Be sure to replace *`region`* with your specific region if different.

Internet access considerations

When you connect Amazon Bedrock AgentCore Runtime or a built-in tool to a Virtual Private Cloud (VPC), it does not have internet access by default. By default, these resources can communicate only with resources inside the same VPC. If your runtime or tool requires access to both VPC resources and the internet, you must configure your VPC accordingly.

Internet access architecture

To enable internet access for your VPC-connected Amazon Bedrock AgentCore Runtime or built-in tool, configure your VPC with the following components:

- **Private subnets** – Place the Amazon Bedrock AgentCore Runtime or tool's network interfaces in private subnets.
- **Public subnets with a NAT gateway** – Deploy a NAT gateway in one or more public subnets to provide outbound internet access for private resources.
- **Internet gateway (IGW)** – Attach an internet gateway to your VPC to enable communication between the NAT gateway and the internet.

Routing configuration

Update your subnet route tables as follows:

- **Private subnet route table –** Add a default route (0.0.0.0/0) that points to the NAT gateway. This allows outbound traffic from the runtime or tool to reach the internet.
- **Public subnet route table –** Add a default route (0.0.0.0/0) that points to the internet gateway. This allows the NAT gateway to communicate with the internet.

> ⚠ **Important**
>
> Connecting Amazon Bedrock AgentCore Runtime and built-in tools to public subnets does not provide internet access. Always use private subnets with NAT gateways for internet connectivity.

Monitoring and troubleshooting

To monitor and troubleshoot your VPC-connected Amazon Bedrock AgentCore Runtime and tools:

CloudWatch Logs

Enable CloudWatch Logs for your Amazon Bedrock AgentCore Runtime to identify any connectivity issues:

- Check error messages related to VPC connectivity
- Look for timeout errors when connecting to VPC resources
- Monitor initialization times (VPC connectivity may increase session startup times)

Common issues and solutions

- **Connection timeouts**:
  - Verify security group rules are correct
  - Ensure route tables are properly configured
  - Check that the target resource is running and accepting connections
- **DNS resolution failures**:
  - Ensure that DNS resolution is enabled in your VPC
  - Verify that your DHCP options are configured correctly
- **Missing ENIs**:
  - Check the IAM permissions to ensure the service-linked role has appropriate permissions

  - Look for any service quotas that may have been reached

Code Interpreter issues

- **Code Interpreter invoke call timeouts when trying to call a public endpoint**:
  - Verify that VPC is configured with NAT gateway for internet access
- **Invoke calls for a Code Interpreter with private VPC endpoints throw "AccessDenied" errors**:
  - Make sure the execution role passed during Code Interpreter creation has the right permissions for AWS service for which VPC endpoint was configured
- **Invoke calls for a Code Interpreter with some private VPC endpoints show "Unable to locate Credentials" error**:
  - Check that the execution role has been provided while creating the code interpreter

Browser Tool issues

- **Live-View/Connection Stream is unable to load webpages and fails with connection timeouts**:
  - Check if the browser was created with Private Subnet with NAT Gateway

Testing VPC connectivity

To verify that your Amazon Bedrock AgentCore Runtime and tools have proper VPC connectivity, you can test connections to your private resources and verify that network interfaces are created correctly in your specified subnets.

To verify that your Amazon Bedrock AgentCore tool has internet access, you can configure a Code Interpreter with your VPC configuration and use the `Invoke` API with `executeCommand` that attempts to connect to a public API or website using `curl` command and check the response. If the connection times out, review your VPC configuration, particularly your route tables and NAT gateway setup.

```
# Using awscurl
awscurl -X POST \
  "https://bedrock-agentcore.<Region>.amazonaws.com/code-interpreters/
<code_interpreter_id>/tools/invoke" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "x-amzn-code-interpreter-session-id: your-session-id" \
  --service bedrock-agentcore \
  --region <Region> \
  -d '{
    "name": "executeCommand",
    "arguments": {
      "command": "curl amazon.com"
    }
  }'
```

Identity and access management for Amazon Bedrock AgentCore

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AgentCore resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Bedrock AgentCore works with IAM
- Identity-based policy examples for Amazon Bedrock AgentCore
- AWS managed policies for Amazon Bedrock AgentCore
- Using service-linked roles for Amazon Bedrock AgentCore
- Understanding Credentials Management in Amazon Bedrock AgentCore
- Troubleshooting Amazon Bedrock AgentCore identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AgentCore.

Service user – If you use the AgentCore service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AgentCore features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AgentCore, see Troubleshooting Amazon Bedrock AgentCore identity and access.

Service administrator – If you're in charge of AgentCore resources at your company, you probably have full access to AgentCore. It's your job to determine which AgentCore features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AgentCore, see How Amazon Bedrock AgentCore works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AgentCore. To view example AgentCore identity-based policies that you can use in IAM, see Identity-based policy examples for Amazon Bedrock AgentCore.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's

permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Bedrock AgentCore works with IAM

Before you use IAM to manage access to AgentCore, learn what IAM features are available to use with AgentCore.

IAM feature	AgentCore support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how AgentCore and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for AgentCore

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for AgentCore

To view examples of AgentCore identity-based policies, see Identity-based policy examples for Amazon Bedrock AgentCore.

Resource-based policies within AgentCore

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for AgentCore

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of AgentCore actions, see Actions Defined by Amazon Bedrock AgentCore in the *Service Authorization Reference*.

Policy actions in AgentCore use the following prefix before the action:

```
bedrock-agentcore
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "bedrock-agentcore:action1",
      "bedrock-agentcore:action2"
         ]
```

To view examples of AgentCore identity-based policies, see Identity-based policy examples for Amazon Bedrock AgentCore.

Policy resources for AgentCore

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its [Amazon Resource Name (ARN)](). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of AgentCore resource types and their ARNs, see [Resources Defined by Amazon Bedrock AgentCore]() in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions Defined by Amazon Bedrock AgentCore]().

To view examples of AgentCore identity-based policies, see [Identity-based policy examples for Amazon Bedrock AgentCore]().

Policy condition keys for AgentCore

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use [condition operators](), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of AgentCore condition keys, see Condition Keys for Amazon Bedrock AgentCore in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions Defined by Amazon Bedrock AgentCore .

To view examples of AgentCore identity-based policies, see Identity-based policy examples for Amazon Bedrock AgentCore.

ACLs in AgentCore

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with AgentCore

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with AgentCore

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for AgentCore

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for AgentCore

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> ⚠ **Warning**
>
> Changing the permissions for a service role might break AgentCore functionality. Edit service roles only when AgentCore provides guidance to do so.

Service-linked roles for AgentCore

Supports service-linked roles: No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon Bedrock AgentCore

By default, users and roles don't have permission to create or modify AgentCore resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by AgentCore, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Amazon Bedrock AgentCore in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete AgentCore resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions –** To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions –** When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access –** You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions –** IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA) –** If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the AgentCore console

To access the Amazon Bedrock AgentCore console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AgentCore resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the AgentCore console, also attach the AgentCore `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policies for Amazon Bedrock AgentCore

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: BedrockAgentCoreFullAccess

You can attach BedrockAgentCoreFullAccess to your users, groups, and roles.

This policy grants permissions that allow full access to the Amazon Bedrock AgentCore.

Permissions details

This policy includes the following permissions:

- `bedrock-agentcore` (Amazon Bedrock Agent Core) – Allows principals full access to all Amazon Bedrock Agent Core resources.
- `iam` (AWS Identity and Access Management) – Allows principals to list and get information about roles and policies, and to pass roles with "BedrockAgentCore" in the name to the bedrock-agentcore service. Also allows creating service-linked roles for CloudWatch Application Signals and Amazon Bedrock AgentCore network.
- `secretsmanager` (AWS Secrets Manager) – Allows principals to create, update, retrieve, and delete secrets with names that begin with "bedrock-agentcore".
- `kms` (AWS Key Management Service) – Allows principals to list and describe keys, and to decrypt data within the same AWS account when called via the Bedrock Agent Core service.
- `s3` (Amazon Simple Storage Service) – Allows principals to get objects from S3 buckets with names that begin with "bedrock-agentcore-gateway-" when called via the Bedrock Agent Core service.
- `lambda` (AWS Lambda) – Allows principals to list Lambda functions.
- `logs` (Amazon CloudWatch Logs) – Allows principals to access, query, and manage log data in log groups related to Bedrock Agent Core and Application Signals, including creating log groups and streams.
- `application-autoscaling` (Application Auto Scaling) – Allows principals to describe scaling policies.
- `application-signals` (Amazon CloudWatch Application Signals) – Allows principals to retrieve information about application signals and start discovery.
- `autoscaling` (Amazon EC2 Auto Scaling) – Allows principals to describe Auto Scaling resources.

- `cloudwatch` (Amazon CloudWatch) – Allows principals to retrieve and list metrics, generate queries, and access other CloudWatch resources.
- `oam` (Amazon CloudWatch Observability Access Manager) – Allows principals to list sinks.
- `rum` (Amazon CloudWatch RUM) – Allows principals to retrieve and list RUM resources.
- `synthetics` (Amazon CloudWatch Synthetics) – Allows principals to describe and get information about Synthetics resources.
- `xray` (AWS X-Ray) – Allows principals to retrieve trace information, manage trace segment destinations, and work with indexing rules.

AWS managed policy: BedrockAgentCoreNetworkServiceRolePolicy

This policy is attached to a service-linked role that allows the service to perform actions on your behalf. You cannot attach this policy to your users, groups, or roles.

This policy grants permissions that allow AgentCore to create and manage network interfaces in your VPC when running in VPC mode.

Permissions details

This policy includes the following permissions:

- `ec2` (Amazon Elastic Compute Cloud) – Allows the service to create, manage, and delete network interfaces in your VPC, assign and unassign private IP addresses, and describe VPC resources. Network interfaces are tagged with "AmazonBedrockAgentCoreManaged" to ensure the service only manages resources it creates.

You can view this policy at BedrockAgentCoreNetworkServiceRolePolicy.

For more information about the service-linked role that uses this policy, see Using service-linked roles for Amazon Bedrock AgentCore.

AWS managed policy: AmazonBedrockAgentCoreMemoryBedrockModelInferenceExecutionRolePolicy

You can attach AmazonBedrockAgentCoreMemoryBedrockModelInferenceExecutionRolePolicy to your users, groups, and roles.

This policy grants permissions that allow full access to the Amazon Bedrock Agent Core Memory.

Permissions details

This policy includes the following permissions.

- `bedrock` – Allows principals to call the Amazon Bedrock `Invokemodel` and `InvokeModelWithResponseStream` actions. This is required so that an agent can store memories.

AgentCore updates to AWS managed policies

View details about updates to AWS managed policies for AgentCore since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the AgentCore Document history page.

Change	Description	Date
BedrockAgentCoreFullAccess – Updated policy	Added permission to create the Amazon Bedrock AgentCore network service-linked role.	September 19, 2025
BedrockAgentCoreNetworkServiceRolePolicy – New policy	Added a new AWS managed policy that allows AgentCore to create and manage network interfaces in your VPC when running in VPC mode.	September 19, 2025
AgentCore started tracking changes	AgentCore started tracking changes for its AWS managed policies.	July 16, 2025

Using service-linked roles for Amazon Bedrock AgentCore

Amazon Bedrock AgentCore uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to AgentCore. Service-linked roles are predefined by AgentCore and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes using AgentCore easier because you don't have to manually add the necessary permissions. AgentCore defines the permissions of its service-linked roles, and unless defined otherwise, only AgentCore can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your AgentCore resources because you can't inadvertently remove permission to access the resources.

AgentCore service-linked role permissions

AgentCore uses the service-linked role named `AWSServiceRoleForBedrockAgentCoreNetwork` to allow AgentCore to create and manage network interfaces in your VPC on your behalf.

The `AWSServiceRoleForBedrockAgentCoreNetwork` service-linked role trusts the following services to assume the role:

- `network.bedrock-agentcore.amazonaws.com`

The role permissions policy allows AgentCore to complete the following actions on the specified resources:

You can view the complete policy at BedrockAgentCoreNetworkServiceRolePolicy. For more information about this policy, see BedrockAgentCoreNetworkServiceRolePolicy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateEniInAnySubnet",
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterface",
            "Resource": "arn:aws:ec2:*:*:subnet/*"
        },
        {
            "Sid": "AllowCreateEniWithSecurityGroups",
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterface",
            "Resource": "arn:aws:ec2:*:*:security-group/*"
        },
        {
            "Sid": "AllowCreateEniWithBedrockManagedRequestTag",
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterface",
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "aws:TagKeys": [
                        "AmazonBedrockAgentCoreManaged"
                    ]
                },
                "StringEquals": {
                    "aws:RequestTag/AmazonBedrockAgentCoreManaged": "true"
                }
            }
        },
        {
            "Sid": "AllowTagEniOnCreate",
            "Effect": "Allow",
            "Action": "ec2:CreateTags",
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
                "StringEquals": {
                    "ec2:CreateAction": "CreateNetworkInterface"
                }
            }
        },
        {
            "Sid": "AllowManageEniWhenBedrockManaged",
            "Effect": "Allow",
            "Action": [
                "ec2:DeleteNetworkInterface",
                "ec2:AssignPrivateIpAddresses",
                "ec2:UnassignPrivateIpAddresses",
                "ec2:CreateNetworkInterfacePermission"
            ],
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/AmazonBedrockAgentCoreManaged": "true"
                }
            }
        },
        {
            "Sid": "AllowGetSecurityGroupsForVpc",
            "Effect": "Allow",
            "Action": [
                "ec2:GetSecurityGroupsForVPC"
            ],
            "Resource": "arn:aws:ec2:*:*:vpc/*"
        },
        {
            "Sid": "AllowDescribeNetworkingResources",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs"
            ],
            "Resource": "*"
        }
    ]
}
```

Creating a service-linked role for AgentCore

You don't need to manually create a service-linked role. When you create an AgentCore Runtime, Code Interpreter, or Browser resources with VPC configuration, AgentCore creates the service-linked role for you automatically.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create an AgentCore Runtime, Code Interpreter, or Browser resources with VPC configuration, AgentCore creates the service-linked role for you again.

Permissions required to create a service-linked role

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. The IAM entity needs to have the following permission:

```
{
    "Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/network.bedrock-
agentcore.amazonaws.com/AWSServiceRoleForBedrockAgentCoreNetwork",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName": "network.bedrock-agentcore.amazonaws.com"
        }
    }
}
```

This permission is already included in the AWS managed policy BedrockAgentCoreFullAccess.

Editing a service-linked role for AgentCore

AgentCore does not allow you to edit the `AWSServiceRoleForBedrockAgentCoreNetwork` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role.

Deleting a service-linked role for AgentCore

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all your AgentCore Runtime, Code Interpreter, and Browser resources with VPC configuration before you can delete the service-linked role.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**, and then choose the name (not the check box) of the `AWSServiceRoleForBedrockAgentCoreNetwork` role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.

4. On the **Access Advisor** tab, review the recent activity for the service-linked role.

> **Note**
>
> If you are unsure whether AgentCore is using the `AWSServiceRoleForBedrockAgentCoreNetwork` role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the Regions where the role is being used. If the role is being used, then you must wait for the session to end before you can delete the role. You cannot revoke the session for a service-linked role.

If you want to remove the `AWSServiceRoleForBedrockAgentCoreNetwork` role, you must first delete all your AgentCore Runtime, Code Interpreter, and Browser resources with VPC configuration.

Manually delete the service-linked role

Use the IAM console, the AWS CLI, or the IAM API to delete the `AWSServiceRoleForBedrockAgentCoreNetwork` service-linked role. For more information, see Deleting a service-linked role.

Understanding Credentials Management in Amazon Bedrock AgentCore

MicroVM Metadata Service (MMDS)

When you configure an execution role with Amazon Bedrock AgentCore Browser, AgentCore Code Interpreter, or AgentCore Runtime, the underlying compute uses MMDS to access credentials, similar to how EC2 instances use the Instance Metadata Service (IMDS). This allows the service within the VM to retrieve temporary AWS credentials for operations like S3 access. This is independent of the network mode that the AgentCore Code Interpreter, AgentCore Browser, or AgentCore Runtime is running in.

> **Important**
>
> When configuring the execution role permissions, use careful consideration since any code or actor running inside the VM can access these credentials by calling the metadata endpoint.

Best Practices for Role Setup

- Follow the principle of least privilege when setting up the execution role. Especially when using these tools with LLMs, that can generate arbitrary code, it's crucial to limit permissions to only what you intend.
- Avoid privilege escalation by ensuring that the execution role associated with your resource has equal or fewer privileges than the users who can invoke it.

The following shows an example of properly scoped permissions:

```
{
  "Effect": "Allow",
  "Action": ["s3:GetObject", "s3:PutObject"],
  "Resource": "arn:aws:s3:::<your-specific-bucket>/*"
}
```

Troubleshooting Amazon Bedrock AgentCore identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AgentCore and IAM.

Topics

I am not authorized to perform an action in AgentCore

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `bedrock-agentcore:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 bedrock-agentcore:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `bedrock-agentcore:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AgentCore.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AgentCore. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AgentCore resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AgentCore supports these features, see How Amazon Bedrock AgentCore works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.

- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation for Amazon Bedrock AgentCore

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your

compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.

- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon Bedrock AgentCore

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, AgentCore offers several features to help support your data resiliency and backup needs.

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon Bedrock AgentCore gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:`*`servicename`*`:*:`*`123456789012`*`:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in AgentCore to prevent the confused deputy problem.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AssumeRolePolicy",
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock-agentcore.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "123456789012"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:bedrock-agentcore:us-
east-1:123456789012:*"
            }
       }
    }
  ]
}
```

Tagging AgentCore resources

A *tag* is a label that you assign to an AWS resource. Each tag consists of a *key* and an optional *value*, both of which you define.

Tags enable you to categorize your AWS resources in different ways, for example, by purpose, owner, or environment. This is useful when you have many resources of the same type—you can quickly identify a specific resource based on the tags you've assigned to it.

Tagging overview

Each tag consists of a key and an optional value. For example, you could define a set of tags for your account's AgentCore resources that helps you track each resource's owner and stack level.

We recommend that you devise a set of tag keys that meets your needs for each resource type. Using a consistent set of tag keys makes it easier for you to manage your resources. You can search and filter the resources based on the tags you add.

Tags don't have any semantic meaning to AgentCore and are interpreted strictly as a string of characters. Also, tags are not automatically assigned to your resources. You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags for the resource are also deleted.

Resources that support tagging

The following AgentCore resources support tagging:

- Agent runtimes
- Agent runtime endpoints
- Code interpreters
- Browsers

Tag restrictions

The following basic restrictions apply to tags:

- Maximum number of tags per resource – 50
- For each resource, each tag key must be unique, and each tag key can have only one value.
- Maximum key length – 128 Unicode characters in UTF-8
- Maximum value length – 256 Unicode characters in UTF-8
- If your tagging schema is used across multiple services and resources, remember that other services may have restrictions on allowed characters. Generally allowed characters are: letters, numbers, and spaces representable in UTF-8, and the following characters: + - = . _ : / @.
- Tag keys and values are case-sensitive.
- Don't use `aws:`, `AWS:`, or any upper or lowercase combination of such as a prefix for either keys or values as it is reserved for AWS use. You can't edit or delete tag keys or values with this prefix. Tags with this prefix do not count against your tags per resource limit.

Working with tags

You can add, edit, or delete tags for AgentCore resources using the console, API, or CLI.

Adding tags

Console

You can add tags when you create the following AgentCore resources:

- Agent runtimes
- Agent runtime endpoints
- Code interpreters
- Browsers

To add tags when creating a resource

1. Open the AgentCore console.
2. Navigate to the resource type you want to create.
3. Follow the steps to create the resource. When you reach the **Tags** section, choose **Add tag**.
4. Enter a tag key and optionally a tag value.
5. To add more tags, choose **Add tag** again.

6. Complete the resource creation process.

To add tags to an existing resource

1. Open the AgentCore console.
2. Navigate to the resource type and select the specific resource.
3. Choose the **Tags** tab.
4. Choose **Add tag**, then enter a key and optionally a value.
5. Choose **Save changes**.

API

You can add tags when you create resources or to existing resources using the AgentCore API.

Adding tags when creating resources

You can add tags when you create resources using the following API operations:

- CreateAgentRuntime – Include tags in the `tags` parameter.
- CreateAgentRuntimeEndpoint – Include tags in the `tags` parameter.
- CreateCodeInterpreter – Include tags in the `tags` parameter.
- CreateBrowser – Include tags in the `tags` parameter.

Adding tags to existing resources

Use the TagResource operation to add tags to existing resources.

CLI

You can use the CLI to add tags to AgentCore resources.

To add tags to a resource

Use the `tag-resource` command:

```
aws bedrock-agentcore tag-resource \
    --resource-arn arn:aws:bedrock-agentcore:us-east-1:123456789012:agent-runtime/
example-runtime \
    --tags Key=Environment,Value=Production Key=Team,Value=AI
```

Managing tags

Console

You can edit tag values for existing resources in the console.

To edit tags for an existing resource

1. Open the AgentCore console.
2. Navigate to the resource type and select the specific resource.
3. Choose the **Tags** tab.
4. To edit a tag, modify the key or value directly.
5. Choose **Save changes**.

API

You can use the AgentCore API to list and manage tags for resources.

Listing tags

Use the ListTagsForResource operation to list the tags for a resource.

Updating tags

Use the TagResource operation to update existing tags by adding new values for existing keys.

CLI

You can use the CLI to list and manage tags for AgentCore resources.

To list tags for a resource

Use the `list-tags-for-resource` command:

```
aws bedrock-agentcore list-tags-for-resource \
    --resource-arn arn:aws:bedrock-agentcore:us-east-1:123456789012:agent-runtime/
example-runtime
```

To update tags for a resource

Use the `tag-resource` command to update existing tags:

```
aws bedrock-agentcore tag-resource \
    --resource-arn arn:aws:bedrock-agentcore:us-east-1:123456789012:agent-runtime/
example-runtime \
    --tags Key=Environment,Value=Staging
```

Deleting tags

Console

You can remove tags from resources in the console.

To delete tags from a resource

1. Open the AgentCore console.
2. Navigate to the resource type and select the specific resource.
3. Choose the **Tags** tab.
4. To delete a tag, choose **Remove** next to the tag.
5. Choose **Save changes**.

API

You can use the AgentCore API to remove tags from resources.

Removing tags

Use the UntagResource operation to remove tags from a resource.

CLI

You can use the CLI to remove tags from AgentCore resources.

To remove tags from a resource

Use the `untag-resource` command:

```
aws bedrock-agentcore untag-resource \
    --resource-arn arn:aws:bedrock-agentcore:us-east-1:123456789012:agent-runtime/
example-runtime \
    --tag-keys Environment Team
```

Quotas for Amazon Bedrock AgentCore

Your AWS account has default quotas, formerly referred to as limits, for each AWS service. Unless otherwise noted, each quota is Region-specific. You can request increases for some quotas, and other quotas cannot be increased.

To view the quotas for AgentCore, open the Service Quotas console. In the navigation pane, choose **AWS services** and select **AgentCore**.

To request a quota increase, contact AWS support.

Topics

AgentCore Runtime Service Quotas

When working with AgentCore Runtime, you need to be aware of the service limits that apply to your account. These limits help ensure service stability and availability for all users.

Resource allocation limits

The following table describes the resource allocation limits for AgentCore Runtime:

Resource allocation limits

Limit	Default Value	Adjustable	Notes
Active Session workloads per account	500 in US East (N. Virginia) and Asia Pacific (Sydney), and 250 in Europe (Frankfurt) and US West (Oregon).	Yes	Can be increased via support ticket

Limit	Default Value	Adjustable	Notes
Total agents per account	1000	Yes	Can be increased via support ticket
Versions per agent	1000	Yes	Inactive versions deleted after 45 days
Endpoints (aliases) per agent	10	Yes	Can be increased via support ticket
Maximum size for a Docker image in an AgentCore Runtime	1 GB	No	
Hardware configuration per session	2vCPU/8GB	No	The maximum memory/CPU usage and configuration per account

Invocation limits

The following table describes the invocation limits for AgentCore Runtime:

Invocation limits

Limit	Value	Adjustable	Notes
Request timeout	15 minutes	No	Maximum time for synchronous requests
Max payload size	100 MB	No	Maximum size for request/response payloads
Streaming max duration	60 mins	No	Maximum time for streaming connectio ns
Async job max duration	8 hours	No	Maximum execution time for asynchron ous jobs

Limit	Value	Adjustable	Notes
Invocations per second	100 per endpoint	Yes	Rate limit for API calls

Lifetime session lifecycle parameters

The following table describes the lifetime session lifecycle parameters for AgentCore Runtime:

Lifetime session lifecycle parameters

Phase	Timeout	Adjustable	Notes
Session Termination	15 minutes of inactivity	No	Execution Environment is terminated. Customer will get a new Execution environment for the Session
Max Session Duration	8 hrs	No	

AgentCore Memory Service Quotas

AgentCore Memory limits

Limit	Value
Number of AgentCore Memory resources per AWS Region in an AWS account.	50
Max Number of memory strategies per AgentCore Memory instance	6
Minimum EventExpirationDuration days in CreateEvent operation	7
Maximum EventExpirationDuration days in CreateEvent operation	365
Prompt Size (AppendToPrompt) for Custom Memory Strategy (Extraction/Consolidation)	30 KB
Max Number of messages per CreateEvent operation	100

Limit	Value
Max message size in CreateEvent operation	9 KB
Max event size in CreateEvent operation	10 MB
Max TPS for CreateEvent (per account)	5
Max TPS for CreateEvent (per combination of account, actor, session)	0.25
Max TPS for RetrieveMemoryRecords (per account)	5
Max TPS for all other AgentCore Memory APIs	20

AgentCore Identity Service Quotas

When working with AgentCore Runtime, you need to be aware of the service limits that apply to your account. These limits help ensure service stability and availability for all users.

AgentCore Identity Limits

Limit	Value	Adjustable	Notes
API keys stored per AWS account per AWS Region	50	Yes	Can be increased via Support ticket
OAuth client credentials stored per AWS account per AWS Region	50	Yes	Can be increased via Support ticket

AgentCore Gateway Service Quotas

This section provides information about Amazon Bedrock AgentCore Gateway endpoints and service limits.

Endpoints

Amazon Bedrock AgentCore Gateway provides AWS Region-specific endpoints for management operations and runtime access.

The Amazon Bedrock AgentCore Gateway control plane endpoints use the following format, where you can replace *<region>* with any of the AWS Regions listed in AWS Regions.

```
bedrock-agentcore-control.<region>.amazonaws.com
```

The AgentCore Gateway URLs for runtime access have the following format:

```
https://{gateway-Id}.gateway.bedrock-agentcore.{Region}.amazonaws.com
```

Where:

- **{gateway-Id}** is the unique identifier for your gateway
- **{Region}** is the AWS Region where your gateway is deployed

Gateway ARNs have the following format:

```
arn:${Partition}:bedrock-agentcore:${Region}:${Account}:gateway/${gateway-Id}
```

The AgentCore service principal is: `bedrock-agentcore.amazonaws.com`

Service quotas

Amazon Bedrock AgentCore Gateway has the following service quotas. You can request increases for some quotas using the Service Quotas console.

Amazon Bedrock AgentCore Gateway service quotas

Quota	Default value	Adjustable	Notes
Number of gateways per account	100	Yes	
Number of targets per gateway	10	Yes	
Number of tools per target	200	Yes	
Timeout for a gateway invocatio n	55 seconds	Yes	Aligns with Lambda function timeout
Maximum inline schema size	1 MB	Yes	

Quota	Default value	Adjustable	Notes
Maximum S3 payload schema size	2 MB	Yes	
Tool name character limit	256 characters	Yes	
CreateGateway API rate	5 TPM	Yes	Transactions per minute
UpdateGateway API rate	5 TPM	Yes	Transactions per minute
GetGateway API rate	5 TPS	Yes	Transactions per second
ListGateways API rate	5 TPS	Yes	Transactions per second
DeleteGateway API rate	5 TPM	Yes	Transactions per minute
CreateGatewayTarget API rate	5 TPM	Yes	Transactions per minute
UpdateGatewayTarget API rate	5 TPM	Yes	Transactions per minute
GetGatewayTarget API rate	5 TPS	Yes	Transactions per second
ListGatewayTargets API rate	5 TPS	Yes	Transactions per second
DeleteGatewayTarget API rate	5 TPM	Yes	Transactions per minute
CallTool/ListTool/SearchTool API rate at gateway level	5 TPS	Yes	Transactions per second
CallTool/ListTool/SearchTool API rate at account level	5 TPS	Yes	Transactions per second
Maximum CallTool/ListTool/SearchTool payload size	6 MB	Yes	

For more information about service quotas and how to request increases, see Requesting a quota increase in the *Service Quotas User Guide*.

AgentCore Browser Service Quotas

The Browser tool has the following service quotas and considerations that apply to your account.

Browser service quotas

Quota	Default Value	Adjustable	Notes
Session duration	15 minutes	Yes	Can be extended up to 8 hours
Concurrent active sessions per account	500	Yes	Can be increased via support ticket
Total Browser tool configurations per account	100	Yes	Can be increased via support ticket
CDP stream and live view stream per session	1 each	No	Allows a single agent and end user to interact with the browser
Hardware configuration per session	1vCPU/4GB	No	The maximum memory/CPU usage and configuration per account

AgentCore Code Interpreter Service Quotas

The Code Interpreter tool has the following service quotas and considerations that apply to your account.

Code Interpreter service quotas

Quota	Default Value	Adjustable	Notes
Execution time	15 minutes	Yes	Can be extended up to 8 hours
Concurrent active sessions per account	500	Yes	Can be increased via support ticket

Quota	Default Value	Adjustable	Notes
Total Code Interpreter tool configurations per account	100	Yes	Can be increased via support ticket
Hardware configuration per session	2vCPU/8GB	No	The maximum memory/CPU usage and configuration per account

Document history for the AgentCore User Guide

The following table describes the documentation releases for Amazon Bedrock AgentCore.

Change	Description	Date
Managed policy updates	Updated managed policy documentation. Added documentation for the BedrockAgentCoreNetworkServiceRolePolicy managed policy. Updated BedrockAgentCoreFullAccess policy to include permission for creating the Amazon Bedrock AgentCore network service-linked role.	September 19, 2025
Initial release (preview)	Initial release of the Amazon Bedrock AgentCore Developer Guide	July 16, 2025