MCPify your AWS Lambda

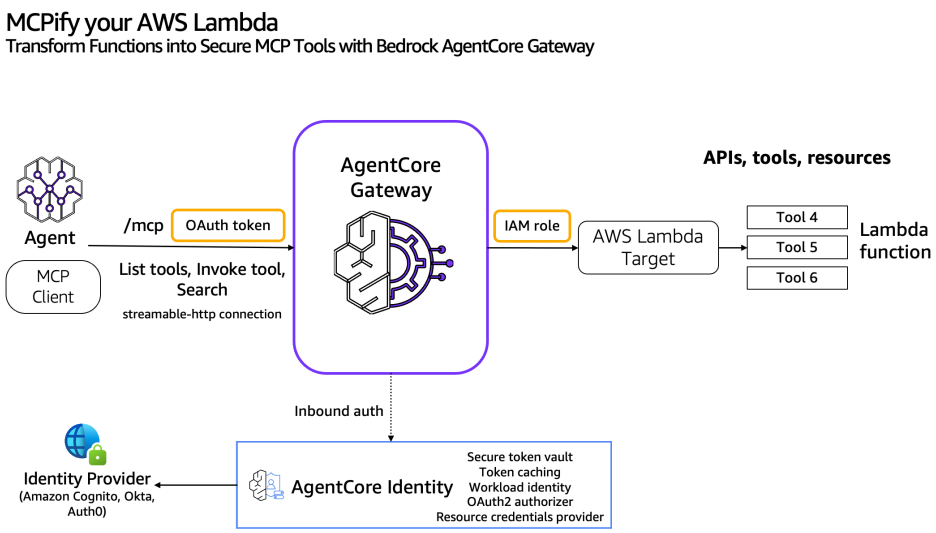
Create an MCP Server with an AWS Lambda function

Overview

The Bedrock AgentCore Gateway enables customers to convert existing or new AWS Lambda functions into fully managed MCP servers—without managing infrastructure or hosting. It provides a consistent Model Context Protocol (MCP) interface across all tools and supports a dual-authentication model for secure access:

- **Inbound Auth**: Authenticates and authorizes users accessing gateway targets (e.g., via OAuth).
- **Outbound Auth**: Securely connects to backend resources on behalf of authenticated users using IAM roles or OAuth tokens.

This setup creates a secure, standardized bridge between agents and backend tools, allowing seamless tool invocation via MCP.



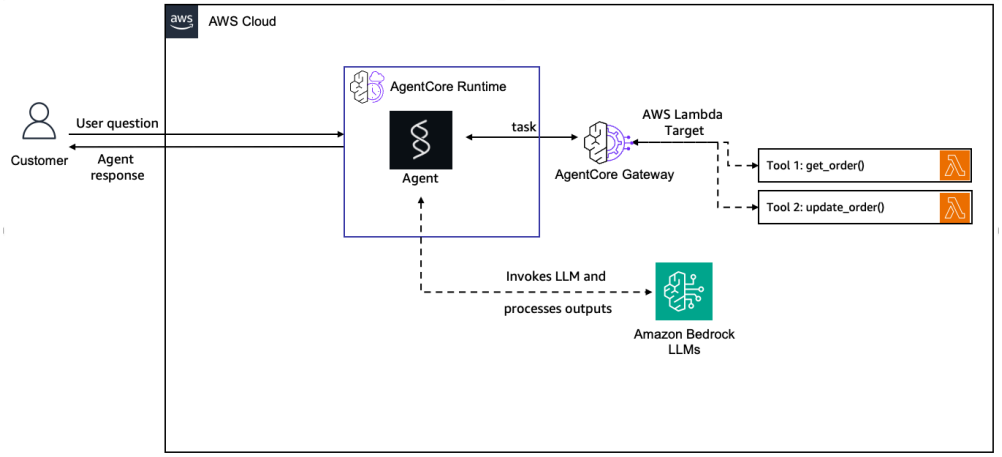
Why is this Important?

The Bedrock AgentCore Gateway is important because it streamlines the process of turning Lambda functions into secure, scalable MCP servers without managing infrastructure. By providing a consistent interface and built-in authentication, it reduces complexity, enhances security, and speeds up development for AI-powered applications.

High-Level Architecture

In this tutorial we will transform operations defined in AWS lambda function into MCP tools and host it in Bedrock AgentCore Gateway. For demonstration purposes, we will use a Strands Agent using Amazon Bedrock models

In our example we will use a very simple agent with two tools: **get_order** and **update_order**.



Architecture Flow:

1. **User Question Submitted:** The customer sends a question or request to the agent hosted within the AgentCore Runtime.
2. **Agent Receives Task:** The AgentCore Runtime receives the user input and interprets the task.
3. **Agent Invokes Gateway Tool:** The agent determines that the task requires execution of a Lambda-based tool and routes the task to the AgentCore Gateway.
4. **Lambda Tool Execution:** The AgentCore Gateway securely invokes the appropriate AWS Lambda target (e.g., get_order() or update_order()).
5. **Tool Returns Output:** The Lambda function processes the task and returns the result back through the Gateway to the Agent.
6. **Agent Invokes LLM (if needed):** The agent may invoke an Amazon Bedrock LLM to process natural language, generate responses, or enhance reasoning.
7. **Final Response Delivered:** The agent composes and returns the final response to the customer based on the tool output and/or LLM reasoning.

Prerequisites & Setup

Environment Requirements

Before starting this lab, ensure you have:

- Jupyter notebook (Python kernel)
- uv
- AWS credentials
- Amazon Cognito

Understanding the AgentCore Gateway and Lambda context object

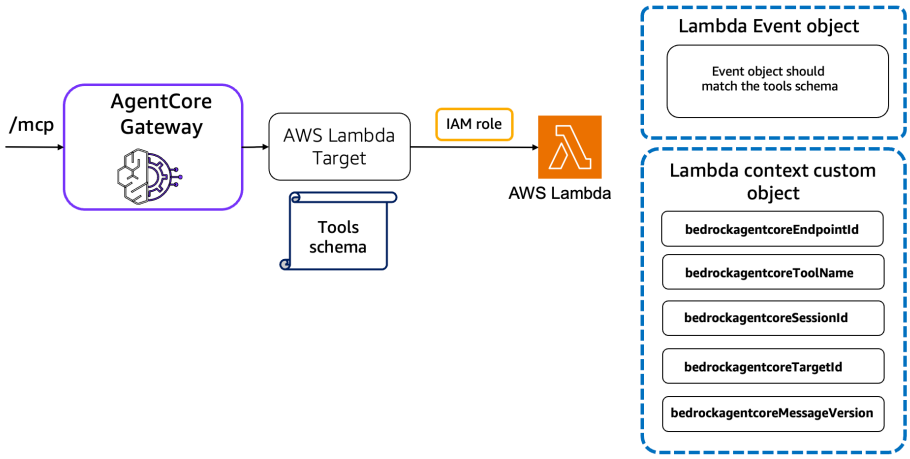
When Gateway invokes a Lambda function, it passes special context information through the `context.client_context object`. This context includes important metadata about the invocation, which your function can use to determine how to process the request.

The following properties are available in the `context.client_context.custom` object:

- **bedrockagentcoreEndpointId:** The ID of the Gateway endpoint that received the request.
- **bedrockagentcoreTargetId:** The ID of the Gateway target that routed the request to your function.
- **bedrockagentcoreMessageVersion:** The version of the message format used for the request.
- **bedrockagentcoreToolName:** The name of the tool being invoked. This is particularly important when your Lambda function implements multiple tools.
- **bedrockagentcoreSessionId:** The session ID for the current invocation, which can be used to correlate multiple tool calls within the same session.

You can access these properties in your Lambda function code to determine which tool is being invoked and to customize your function's behavior accordingly

AWS Lambda function tools for Bedrock AgentCore Gateway



Implementation of AgentCore Gateway and AWS Lambda

1. Configure Authentication for Incoming AgentCore Gateway Requests

The below code checks if the uv package manager is installed, and installs it if it's missing. Then, it uses uv to install the latest versions of the botocore and boto3 libraries. The script ensures dependency installation is handled in a consistent, reproducible way using uv.

▼ **Code Snippet**

```
# Make sure you download the latest botocore and boto3 libraries.
import shutil
import subprocess
import sys

def ensure_uv_installed():
    if shutil.which("uv") is None:
        print("🔧 'uv' not found. Installing with pip...")
        subprocess.check_call([sys.executable, "-m", "pip", "install", "uv"])
    else:
        print("✅ 'uv' is already installed.")

def uv_install(*packages):
    ensure_uv_installed()
    uv_path = shutil.which("uv")
    print(f"📦 Installing {', '.join(packages)} using uv...")
    subprocess.check_call([uv_path, "pip", "install", *packages])

uv_install("botocore", "boto3")
```

This next code snippet adds the parent directory to sys.path so that a utils.py module can be imported, even if the script is run from different environments like Jupyter.

▼ **Code Snippet**

```
import os
import sys

# Get the directory of the current script
if '__file__' in globals():
    current_dir = os.path.dirname(os.path.abspath(__file__))
else:
```

```
    current_dir = os.getcwd()  # Fallback if __file__ is not defined (e.g., Jupyter)

# Navigate to the directory containing utils.py (one level up)
utils_dir = os.path.abspath(os.path.join(current_dir, '..'))

# Add to sys.path
sys.path.insert(0, utils_dir)

# Now you can import utils
import utils
```

After, we create a sample AWS Lambda function from a ZIP file using a utility function, intending to convert it into an MCP tool. It then checks the response to confirm whether the function was created successfully and prints the Lambda function's ARN or an error message.

▼ **Code Snippet**

```
#### Create a sample AWS Lambda function that you want to convert into MCP tools
lambda_resp = utils.create_gateway_lambda("lambda_function_code.zip")

if lambda_resp is not None:
    if lambda_resp['exit_code'] == 0:
        print("Lambda function created with ARN: ", lambda_resp['lambda_function_arn'])
    else:
        print("Lambda function creation failed with message: ", lambda_resp['lambda_function_arn'
```

We then create an IAM role for AgentCore Gateway

▼ **Code Snippet**

```
#### Create an IAM role for the Gateway to assume
import utils
agentcore_gateway_iam_role = utils.create_agentcore_gateway_role("sample-lambdagateway")
print("Agentcore gateway role ARN: ", agentcore_gateway_iam_role['Role']['Arn'])
```

2. Create Amazon Cognito Pool for Inbound authorization to Gateway

Now, we are setting up Amazon Cognito resources required for authenticating requests to the MCP server. It creates (or retrieves) a user pool, a resource server with `read` and `write` scopes, and a machine-to-machine (M2M) client application. Finally, it generates the OpenID Connect discovery URL for the user pool, which is used for configuring JWT-based authorization in AgentCore Runtime.

▼ **Code Snippet**

```
# Creating Cognito User Pool
import os
import boto3
import requests
import time
from botocore.exceptions import ClientError

REGION = os.environ['AWS_DEFAULT_REGION']
USER_POOL_NAME = "sample-agentcore-gateway-pool"
RESOURCE_SERVER_ID = "sample-agentcore-gateway-id"
RESOURCE_SERVER_NAME = "sample-agentcore-gateway-name"
CLIENT_NAME = "sample-agentcore-gateway-client"
SCOPES = [
    {"ScopeName": "gateway:read", "ScopeDescription": "Read access"},
    {"ScopeName": "gateway:write", "ScopeDescription": "Write access"}
]
scopeString = f"{RESOURCE_SERVER_ID}/gateway:read {RESOURCE_SERVER_ID}/gateway:write"

cognito = boto3.client("cognito-idp", region_name=REGION)
```

```
20
21  user_pool_id = utils.get_or_create_user_pool(cognito, USER_POOL_NAME)
22
23  utils.get_or_create_resource_server(cognito, user_pool_id, RESOURCE_SERVER_ID, RESOURCE_SERVER_NA
24
25  client_id, client_secret = utils.get_or_create_m2m_client(cognito, user_pool_id, CLIENT_NAME, RE
26
27  # Get discovery URL
28  cognito_discovery_url = f'https://cognito-idp.{REGION}.amazonaws.com/{user_pool_id}/.well-known/
```

3. Create the Gateway with Amazon Cognito Authorizer for inbound authorization

This block creates an AgentCore Gateway using the `create_gateway` API from the Bedrock AgentCore control plane. The gateway is secured using a custom JWT authorizer backed by Amazon Cognito. The `client_id` and `discoveryUrl` come from the Cognito user pool and app client that were previously created. The IAM role passed to `roleArn` must have permissions for Gateway operations. After creation, the `gatewayId` and `gatewayUrl` are extracted from the response—these are needed to register a Gateway target or route MCP requests to a Lambda or other backend.

▼ **Code Snippet**

```
# CreateGateway with Cognito authorizer without CMK. Use the Cognito user pool created in pre
gateway_client = boto3.client('bedrock-agentcore-control', region_name = os.environ['AWS_DEFAULT_
auth_config = {
    "customJWTAuthorizer": {
        "allowedClients": [client_id],  # Client MUST match with the ClientId configured in Cogni
        "discoveryUrl": cognito_discovery_url
    }
}
create_response = gateway_client.create_gateway(name='TestGWforLambda',
    roleArn = agentcore_gateway_iam_role['Role']['Arn'], # The IAM Role must have permissions to
    protocolType='MCP',
    authorizerType='CUSTOM_JWT',
    authorizerConfiguration=auth_config,
    description='AgentCore Gateway with AWS Lambda target type'
)
print(create_response)
# Retrieve the GatewayID used for GatewayTarget creation
gatewayID = create_response["gatewayId"]
gatewayURL = create_response["gatewayUrl"]
print(gatewayID)
```

4. Create an AWS Lambda target and transform into MCP tools

This code registers an AWS Lambda function as the target for the previously created AgentCore Gateway. The `lambda_target_config` block defines the ARN of the Lambda function and specifies the tools it exposes (`get_order_tool` and `update_order_tool`) using inline schema definitions. These schemas describe the expected input format for each tool. The `credentialProviderConfigurations` block uses the Gateway's IAM role to authenticate and authorize invocation of the Lambda function. Finally, `create_gateway_target` registers the configuration with the Gateway under the name `LambdaUsingSDK`.

▼ **Code Snippet**

```
# Replace the AWS Lambda function ARN below
lambda_target_config = {
    "mcp": {
        "lambda": {
            "lambdaArn": lambda_resp['lambda_function_arn'], # Replace this with your AWS Lambda
            "toolSchema": {
                "inlinePayload": [
                    {
```

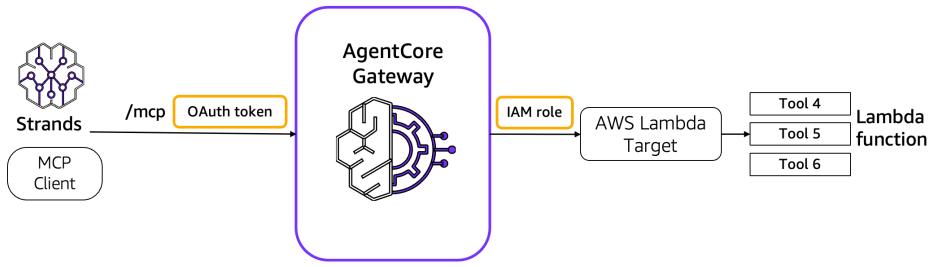
```
                    "name": "get_order_tool",
                    "description": "tool to get the order",
                    "inputSchema": {
                        "type": "object",
                        "properties": {
                            "orderId": {
                                "type": "string"
                            }
                        },
                        "required": ["orderId"]
                    }
                },
                {
                    "name": "update_order_tool",
                    "description": "tool to update the orderId",
                    "inputSchema": {
                        "type": "object",
                        "properties": {
                            "orderId": {
                                "type": "string"
                            }
                        },
                        "required": ["orderId"]
                    }
                }
            ]
        }
    }
}

credential_config = [
    {
        "credentialProviderType" : "GATEWAY_IAM_ROLE"
    }
]
targetname='LambdaUsingSDK'
response = gateway_client.create_gateway_target(
    gatewayIdentifier=gatewayID,
    name=targetname,
    description='Lambda Target using SDK',
    targetConfiguration=lambda_target_config,
    credentialProviderConfigurations=credential_config)
```

5. Call Bedrock AgentCore Gateway from a Strands Agent

The Strands agent connects securely to AWS services using the Bedrock AgentCore Gateway, which implements the Model Context Protocol (MCP). This Gateway provides standard MCP APIs—`ListTools` and `InvokeTools`—allowing agents to discover and invoke tools in a consistent and authorized manner.

Each tool invocation by the agent goes through an MCP-compliant authorization step, ensuring secure and controlled access. This setup allows any client or SDK that follows the MCP spec to communicate with AWS services via the Gateway, enabling a scalable and future-ready agent integration pattern.

Strands agent calling MCP tools of AWS Lambda using Bedrock AgentCore Gateway



Request the access token from Amazon Cognito for inbound authorization

▼ Code Snippet

```
print("Requesting the access token from Amazon Cognito authorizer...May fail for some time tl
token_response = utils.get_token(user_pool_id, client_id, client_secret,scopeString,REGION)
token = token_response["access_token"]
print("Token response:", token)
```

Strands agent calling MCP tools of AWS Lambda using Bedrock AgentCore Gateway

This code snippet sets up an MCP client for a Strands agent using a streamable HTTP transport to connect securely to the AgentCore Gateway. It also initializes a Bedrock model (us.amazon.nova-pro-v1:0) with a specified temperature for generating responses. IAM credentials are assumed to be configured for access to Amazon Bedrock.

▼ Code Snippet

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

def create_streamable_http_transport():
    return streamablehttp_client(gatewayURL,headers={"Authorization": f"Bearer {token}"})

client = MCPClient(create_streamable_http_transport)

## The IAM credentials configured in ~/.aws/credentials should have access to Bedrock model
yourmodel = BedrockModel(
    model_id="us.amazon.nova-pro-v1:0",
    temperature=0.7,
)
```

Strands Agent Tool Discovery and Invocation Example

We will now configure logging for the Strands framework and initializes an agent using a previously defined model and tool list from the MCP client. It demonstrates how to list available tools, invoke the agent with prompts, and explicitly call an MCP tool (e.g., get_order_tool) tied to an AWS Lambda function, then prints the tool's response.

▼ Code Snippet

```
from strands import Agent
import logging

```

```
4
5   # Configure the root strands logger. Change it to DEBUG if you are debugging the issue.
6   logging.getLogger("strands").setLevel(logging.INFO)
7
8   # Add a handler to see the logs
9   logging.basicConfig(
10      format="%(levelname)s | %(name)s | %(message)s",
11      handlers=[logging.StreamHandler()]
12  )
13
14  with client:
15      # Call the listTools
16      tools = client.list_tools_sync()
17      # Create an Agent with the model and tools
18      agent = Agent(model=yourmodel,tools=tools) ## you can replace with any model you like
19      print(f"Tools loaded in the agent are {agent.tool_names}")
20      print(f"Tools configuration in the agent are {agent.tool_config}")
21      # Invoke the agent with the sample prompt. This will only invoke  MCP listTools and retrieve
22      agent("Hi , can you list all tools available to you")
23      # Invoke the agent with sample prompt, invoke the tool and display the response
24      agent("Check the order status for order id 123 and show me the exact response from the tool"
25      # Call the MCP tool explicitly. The MCP Tool name and arguments must match with your AWS Laml
26      result = client.call_tool_sync(
27      tool_use_id="get-order-id-123-call-1", # You can replace this with unique identifier.
28      name=targetname+"___get_order_tool", # This is the tool name based on AWS Lambda target types
29      arguments={"orderId": "123"}
30      )
31      # Print the MCP Tool response
32      print(f"Tool Call result: {result['content'][0]['text']}")
```

Issue: if you get below error while executing below cell, it indicates incompatibily between pydantic and pydantic-core versions.

```
TypeError: model_schema() got an unexpected keyword argument 'generic_origin'
```

How to resolve?

You will need to make sure you have pydantic==2.7.2 and pydantic-core 2.27.2 that are both compatible. Restart the kernel once done.

Key Learnings

- **Gateway Functionality:** Bedrock AgentCore Gateway converts Lambda functions into managed MCP servers, providing a standardized interface and dual authentication without infrastructure management.

  Agent (with optional LLM) → User, creating a seamless interaction between AI agents and backend services.
- **Setup Requirements:** Implementing this system requires AWS credentials, Cognito setup, environment configuration, Lambda function preparation, and proper IAM roles.
- **Implementation Process:** The process involves configuring authentication, creating a Cognito pool, setting up the Gateway, transforming Lambda functions, configuring the agent, handling tokens, and implementing MCP tools.

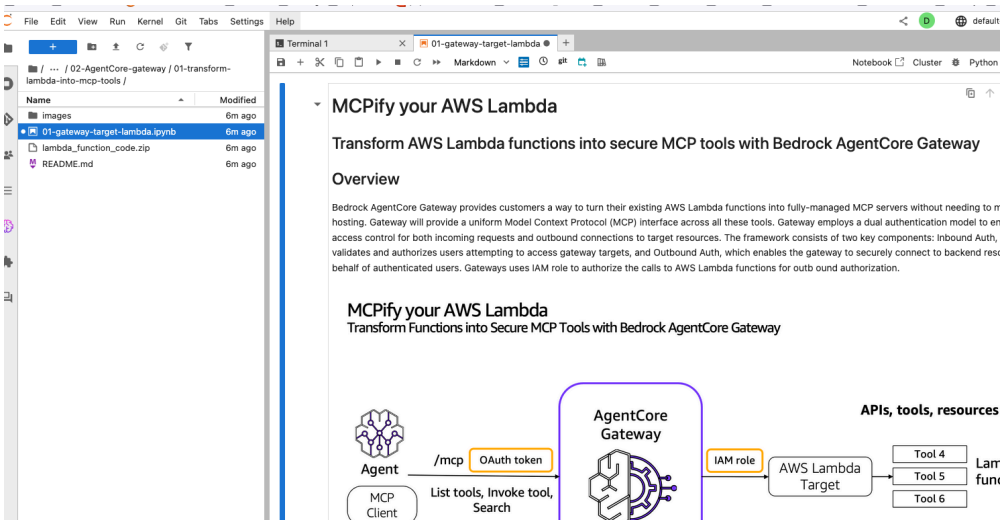
Try it out

At an AWS Event

If you are following the workshop via workshop studio, go to JupyterLab in SageMaker Studio. In the JupyterLab UI navigate to 02-AgentCore-gateway/01-transform-lambda-into-mcp-tools

Self-paced

Here's a notebook that lets you try out the above: 01-gateway-target-lambda.ipynb.



Previous Next