Java code using LLM

Thursday, January 23, 2025

3:13 PM

To use AWS services to call a Language Model (LLM) and utilize Retrieval-Augmented Generation (RAG) with Java, we typically need to use Amazon's SDK for Java. Here's a step-by-step guide to demonstrate how you can achieve this:

### Prerequisites

1. \*\*AWS SDK for Java\*\*: Make sure you have AWS SDK for Java set up in your development environment. You can add it to your project using Maven or Gradle.

2. \*\*AWS Account\*\*: You'll need access to AWS services such as Amazon S3, Amazon Lambda, or Amazon SageMaker, depending on how you plan to deploy or interact with the LLM.

### 1. Setting Up AWS SDK

\*\*Maven Dependency\*\*:

Make sure you include AWS SDK and other required libraries in your `pom.xml`:

```xml

<dependency>

<groupId>software.amazon.awssdk</groupId>

<artifactId>sdk-bundle</artifactId>

<version>2.x.x</version> <!-- replace with the SDK version you are using -->

</dependency>

```

### 2. Invoking a Language Model on AWS

Suppose the LLM is hosted on Amazon SageMaker. To invoke it, you might utilize the `AmazonSageMakerRuntime` client.

#### Code to Invoke the LLM

```java

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.sagemaker.SageMakerRuntimeClient;

import software.amazon.awssdk.services.sagemaker.model.InvokeEndpointRequest;

import software.amazon.awssdk.services.sagemaker.model.InvokeEndpointResponse;

public class LLMInvocation {

private static final String ENDPOINT\_NAME = "your-endpoint-name"; // replace with your endpoint name

public static void main(String[] args) {

Region region = Region.US\_WEST\_2; // replace with your region

SageMakerRuntimeClient sageMakerClient = SageMakerRuntimeClient.builder()

.region(region)

.build();

String inputPayload = "{\"inputs\": \"Your query or prompt text here\"}";

InvokeEndpointRequest request = InvokeEndpointRequest.builder()

.contentType("application/json")

.accept("application/json")

.endpointName(ENDPOINT\_NAME)

.body(SdkBytes.fromUtf8String(inputPayload))

.build();

InvokeEndpointResponse response = sageMakerClient.invokeEndpoint(request);

System.out.println(response.body().asUtf8String());

sageMakerClient.close();

}

}

```

### 3. Creating a LLM with Retrieval-Augmented Generation (RAG)

Creating a full RAG setup from scratch involves several steps, including setting up data retrieval components, integrating them with an LLM, and customizing for specific tasks. A simplified Java-based implementation with an outline for creating an LLM:

#### Simplified Outline for RAG

1. \*\*Document Retrieval\*\*: You need to have a retrieval system. For simplicity, assume documents are stored in an Amazon S3 bucket.

2. \*\*LLM Processing\*\*: Utilize the SageMaker endpoint to generate responses based on retrieved documents.

#### Java Code Structure for RAG

```java

import software.amazon.awssdk.services.s3.S3Client;

import software.amazon.awssdk.services.s3.model.GetObjectRequest;

import software.amazon.awssdk.services.s3.model.S3Object;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;

import java.util.List;

public class RAGExample {

private static final String BUCKET\_NAME = "your-bucket-name"; // replace with your bucket name

public static void main(String[] args) {

Region region = Region.US\_WEST\_2; // Adjust region

try (S3Client s3 = S3Client.builder().region(region).credentialsProvider(ProfileCredentialsProvider.create()).build()) {

List<S3Object> objects = listBucketObjects(s3, BUCKET\_NAME);

// Retrieve and process documents as needed

objects.forEach(s3Object -> {

// Assume each document is stored as a plain text file

String documentContent = getDocumentContent(s3, s3Object);

// Pass document content to your LLM for processing

});

}

}

private static List<S3Object> listBucketObjects(S3Client s3, String bucketName) {

return s3.listObjectsV2Request(builder -> builder.bucket(bucketName)).contents();

}

private static String getDocumentContent(S3Client s3, S3Object s3Object) {

String key = s3Object.key();

GetObjectRequest getObjectRequest = GetObjectRequest.builder().bucket(BUCKET\_NAME).key(key).build();

return s3.getObjectAsBytes(getObjectRequest).asUtf8String();

}

}

```

### Summary

- Replace `"your-endpoint-name"`, `"your-bucket-name"`, and other placeholders with your actual AWS service details.

- The example above provides a basic RAG setup where document retrieval is done from an S3 bucket, and responses are generated using a SageMaker-hosted LLM.

- Make sure you have proper IAM roles and permissions setup to allow access to the respective AWS resources.

This guide assumes familiarity with AWS, Java, and related tools for deeper configuration and deployment tasks required for a production-grade setup. If any specific area needs further clarification or detailed implementation steps, feel free to ask!

Creating a Language Learning Model (LLM) from scratch is a complex task that usually involves substantial resources and expertise in machine learning, data science, and software engineering. However, I can guide you on how to deploy a pre-trained model using Amazon SageMaker, which allows you to leverage powerful models without building them from the ground up. Below is an example of how you can set up such a service using Java to deploy and utilize an LLM on AWS.

### Prerequisites

1. \*\*AWS SageMaker\*\*: You need access to an AWS account with SageMaker capabilities.

2. \*\*AWS SDK for Java\*\*: Ensure you have the AWS SDK installed and configured.

3. \*\*Pre-trained Model\*\*: Typically, ML models are trained using frameworks like TensorFlow or PyTorch, which can then be deployed on SageMaker.

### Deploying a Pre-trained Model on AWS SageMaker

To deploy a model, follow these generalized steps. I'm going to use the AWS SDK in Java to orchestrate the deployment of a model. However, the training or fine-tuning part is typically done in Python because of its rich ecosystem for machine learning frameworks.

\*\*Note\*\*: This example presupposes you have a model artifact uploaded to an S3 bucket.

#### Step 1: Define the Deployment Job

We'll write a Java application to create a SageMaker endpoint using a pre-trained model.

```java

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.sagemaker.SageMakerClient;

import software.amazon.awssdk.services.sagemaker.model.CreateEndpointConfigRequest;

import software.amazon.awssdk.services.sagemaker.model.CreateEndpointRequest;

import software.amazon.awssdk.services.sagemaker.model.CreateModelRequest;

import software.amazon.awssdk.services.sagemaker.model.CreateModelResponse;

import software.amazon.awssdk.services.sagemaker.model.ContainerDefinition;

import software.amazon.awssdk.services.sagemaker.model.ProductionVariant;

import software.amazon.awssdk.services.sagemaker.model.SageMakerException;

public class DeployLLM {

private static final String MODEL\_NAME = "your-model-name";

private static final String ROLE\_ARN = "your-role-arn"; // SageMaker execution role ARN

private static final String PRIMARY\_CONTAINER\_IMAGE = "model-container-image";

private static final String S3\_MODEL\_ARTIFACT = "s3://path-to-your-model-artifact";

private static final String INSTANCE\_TYPE = "ml.m5.large"; // Choose instance type

private static final String ENDPOINT\_CONFIG\_NAME = "your-endpoint-config-name";

private static final String ENDPOINT\_NAME = "your-endpoint-name";

public static void main(String[] args) {

Region region = Region.US\_WEST\_2; // Change as required

SageMakerClient sageMakerClient = SageMakerClient.builder()

.region(region)

.credentialsProvider(ProfileCredentialsProvider.create())

.build();

try {

createModel(sageMakerClient);

createEndpointConfig(sageMakerClient);

createEndpoint(sageMakerClient);

System.out.println("LLM Deployed Successfully!");

} catch (SageMakerException e) {

System.err.println(e.awsErrorDetails().errorMessage());

}

}

private static void createModel(SageMakerClient sageMakerClient) {

ContainerDefinition containerDefinition = ContainerDefinition.builder()

.image(PRIMARY\_CONTAINER\_IMAGE)

.modelDataUrl(S3\_MODEL\_ARTIFACT)

.build();

CreateModelRequest modelRequest = CreateModelRequest.builder()

.modelName(MODEL\_NAME)

.primaryContainer(containerDefinition)

.executionRoleArn(ROLE\_ARN)

.build();

CreateModelResponse response = sageMakerClient.createModel(modelRequest);

System.out.println("Model ARN: " + response.modelArn());

}

private static void createEndpointConfig(SageMakerClient sageMakerClient) {

ProductionVariant productionVariant = ProductionVariant.builder()

.modelName(MODEL\_NAME)

.variantName("AllTraffic")

.instanceType(INSTANCE\_TYPE)

.initialInstanceCount(1)

.build();

CreateEndpointConfigRequest configRequest = CreateEndpointConfigRequest.builder()

.endpointConfigName(ENDPOINT\_CONFIG\_NAME)

.productionVariants(productionVariant)

.build();

sageMakerClient.createEndpointConfig(configRequest);

}

private static void createEndpoint(SageMakerClient sageMakerClient) {

CreateEndpointRequest endpointRequest = CreateEndpointRequest.builder()

.endpointName(ENDPOINT\_NAME)

.endpointConfigName(ENDPOINT\_CONFIG\_NAME)

.build();

sageMakerClient.createEndpoint(endpointRequest);

}

}

```

### Explanation

- \*\*Model Creation\*\*: The script begins by defining a container that holds the model data (from S3) and the inference image required.

- \*\*Endpoint Configuration\*\*: A set of resources (like instance types) is specified for handling inference requests.

- \*\*Endpoint Creation\*\*: This final step makes the model available for invocation.

### Additional Configuration

- \*\*IAM Role\*\*: The `ROLE\_ARN` is an IAM Role allowing SageMaker to interact with AWS services on your behalf (e.g., S3).

- \*\*Model Container Image\*\*: A pre-built container image compatible with the framework used by your model (e.g., TensorFlow, PyTorch).

### Deploy and Use

1. \*\*Adjust placeholders\*\*: Replace all placeholder strings (e.g., `"your-model-name"`, etc.) with actual values from your AWS setup.

2. \*\*Run the Java Application\*\*: Ensure you have the necessary SDK and tools configured to execute the deployment script.

3. \*\*Invoke Model\*\*: Once deployed, you can invoke your model using the endpoint similarly to the previous example provided for invoking the LLM.

Building a full LLM from scratch within Java is unusual due to its lack of machine learning framework support. Such tasks are more efficiently handled using Python-based libraries and subsequently deployed on services like AWS for production-level operations. If you have further questions or need assistance with setup, feel free to ask!

To add Retrieval-Augmented Generation (RAG) capabilities to the previously provided code, we need to integrate a retrieval mechanism that fetches relevant documents or data before passing them to the LLM for augmentation. Here's a modified version of the code that includes a simplistic RAG setup using AWS services like Amazon S3 for document retrieval, along with using AWS SageMaker to deploy and utilize an LLM.

\*\*Note\*\*: We'll assume the SageMaker model has been designed to accept both a query and related documents as inputs and is capable of handling RAG tasks.

### Complete Java Code with RAG Functionality

```java

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;

import software.amazon.awssdk.core.SdkBytes;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.s3.S3Client;

import software.amazon.awssdk.services.s3.model.GetObjectRequest;

import software.amazon.awssdk.services.s3.model.ListObjectsV2Request;

import software.amazon.awssdk.services.s3.model.ListObjectsV2Response;

import software.amazon.awssdk.services.s3.model.S3Object;

import software.amazon.awssdk.services.sagemaker.SageMakerClient;

import software.amazon.awssdk.services.sagemaker.SageMakerRuntimeClient;

import software.amazon.awssdk.services.sagemaker.model.\*;

import java.util.ArrayList;

import java.util.List;

public class RAGSystem {

// Configuration constants

private static final String S3\_BUCKET\_NAME = "your-bucket-name";

private static final String MODEL\_NAME = "your-model-name";

private static final String ROLE\_ARN = "your-role-arn"; // SageMaker execution role ARN

private static final String PRIMARY\_CONTAINER\_IMAGE = "model-container-image";

private static final String S3\_MODEL\_ARTIFACT = "s3://path-to-your-model-artifact";

private static final String INSTANCE\_TYPE = "ml.m5.large"; // Modify as needed

private static final String ENDPOINT\_CONFIG\_NAME = "your-endpoint-config-name";

private static final String ENDPOINT\_NAME = "your-endpoint-name";

private static final Region REGION = Region.US\_WEST\_2; // Adjust based on your setup

public static void main(String[] args) {

SageMakerClient sageMakerClient = SageMakerClient.builder()

.region(REGION)

.credentialsProvider(ProfileCredentialsProvider.create())

.build();

SageMakerRuntimeClient sageMakerRuntimeClient = SageMakerRuntimeClient.builder()

.region(REGION)

.credentialsProvider(ProfileCredentialsProvider.create())

.build();

try {

// Setup and deployment of the LLM (STEP 1)

createModel(sageMakerClient);

createEndpointConfig(sageMakerClient);

createEndpoint(sageMakerClient);

// Step 2: Document retrieval from S3

List<String> documents = retrieveDocumentsFromS3();

// Step 3: Query input and invocation of LLM

String query = "What is the impact of climate change?";

String retrievedText = String.join(" ", documents); // Concatenate retrieved texts or select relevant ones

String response = invokeLLM(sageMakerRuntimeClient, query, retrievedText);

System.out.println("Augmented Response: " + response);

} catch (SageMakerException e) {

System.err.println(e.awsErrorDetails().errorMessage());

}

}

private static List<String> retrieveDocumentsFromS3() {

List<String> documents = new ArrayList<>();

try (S3Client s3Client = S3Client.builder()

.region(REGION)

.credentialsProvider(ProfileCredentialsProvider.create())

.build()) {

ListObjectsV2Request listObjectsReq = ListObjectsV2Request.builder()

.bucket(S3\_BUCKET\_NAME)

.build();

ListObjectsV2Response listObjectsResponse;

do {

listObjectsResponse = s3Client.listObjectsV2(listObjectsReq);

for (S3Object s3Object : listObjectsResponse.contents()) {

String documentContent = getDocumentContent(s3Client, s3Object.key());

documents.add(documentContent);

}

listObjectsReq = listObjectsReq.toBuilder()

.continuationToken(listObjectsResponse.nextContinuationToken())

.build();

} while (listObjectsResponse.isTruncated());

}

return documents;

}

private static String getDocumentContent(S3Client s3Client, String key) {

GetObjectRequest getObjectRequest = GetObjectRequest.builder()

.bucket(S3\_BUCKET\_NAME)

.key(key)

.build();

return s3Client.getObjectAsBytes(getObjectRequest).asUtf8String();

}

private static String invokeLLM(SageMakerRuntimeClient sageMakerClient, String query, String context) {

try {

String inputPayload = "{\"query\": \"" + query + "\", \"context\": \"" + context + "\"}";

InvokeEndpointRequest request = InvokeEndpointRequest.builder()

.contentType("application/json")

.accept("application/json")

.endpointName(ENDPOINT\_NAME)

.body(SdkBytes.fromUtf8String(inputPayload))

.build();

InvokeEndpointResponse response = sageMakerClient.invokeEndpoint(request);

return response.body().asUtf8String();

} catch (SageMakerException e) {

System.err.println("Failed to invoke LLM: " + e.awsErrorDetails().errorMessage());

return "";

}

}

private static void createModel(SageMakerClient sageMakerClient) {

ContainerDefinition containerDefinition = ContainerDefinition.builder()

.image(PRIMARY\_CONTAINER\_IMAGE)

.modelDataUrl(S3\_MODEL\_ARTIFACT)

.build();

CreateModelRequest modelRequest = CreateModelRequest.builder()

.modelName(MODEL\_NAME)

.primaryContainer(containerDefinition)

.executionRoleArn(ROLE\_ARN)

.build();

CreateModelResponse response = sageMakerClient.createModel(modelRequest);

System.out.println("Model ARN: " + response.modelArn());

}

private static void createEndpointConfig(SageMakerClient sageMakerClient) {

ProductionVariant productionVariant = ProductionVariant.builder()

.modelName(MODEL\_NAME)

.variantName("AllTraffic")

.instanceType(INSTANCE\_TYPE)

.initialInstanceCount(1)

.build();

CreateEndpointConfigRequest configRequest = CreateEndpointConfigRequest.builder()

.endpointConfigName(ENDPOINT\_CONFIG\_NAME)

.productionVariants(productionVariant)

.build();

sageMakerClient.createEndpointConfig(configRequest);

}

private static void createEndpoint(SageMakerClient sageMakerClient) {

CreateEndpointRequest endpointRequest = CreateEndpointRequest.builder()

.endpointName(ENDPOINT\_NAME)

.endpointConfigName(ENDPOINT\_CONFIG\_NAME)

.build();

sageMakerClient.createEndpoint(endpointRequest);

}

}

```

### Explanation

1. \*\*Deployment\*\*: The same `createModel`, `createEndpointConfig`, and `createEndpoint` methods are used to deploy the pre-trained LLM on AWS SageMaker.

2. \*\*Document Retrieval (RAG part)\*\*: This retrieves documents from the specified Amazon S3 bucket. You can customize the retrieval mechanism to filter and select only the most relevant documents.

3. \*\*LLM Invocation\*\*: The `invokeLLM` method now takes both `query` and `context` (i.e., the retrieved documents) as inputs, assuming that the SageMaker model can handle this for RAG. It sends these inputs to a SageMaker endpoint to get augmented responses.

### Steps to Run

- \*\*Replace\*\* all placeholder values with actual configurations from your AWS setup.

- \*\*Ensure\*\* you have correct permissions configured in your AWS environment for accessing S3 and SageMaker.

- \*\*Run\*\* the Java application in an environment configured with AWS SDK.

This code is a high-level abstraction and assumes that the LLM deployed is a complex service capable of handling intricate queries and documents. For an actual deployment, you'd need to ensure the LLM service is appropriately configured with RAG architectures.

To utilize the Java code from the previous RAG setup and invoke the SageMaker endpoint to get a response, we will focus on creating a simple client application. This application will interact with the deployed LLM model and process queries using the infrastructure established in the previous code. The example will assume that you have your SageMaker endpoint already up and running, based on the model deployment from the previous example.

### Key Steps for the Client Application

1. \*\*Query Input\*\*: Accept a query input that you want to process with the RAG model.

2. \*\*Retrieve Documents\*\*: Interact with S3 to fetch any specific documents if needed (assuming pre-retrieval is handled, or static data is available).

3. \*\*Invoke SageMaker Endpoint\*\*: Send the query (and optional context or documents) to your SageMaker endpoint and get a response.

### Simple Java Client to Invoke RAG Model

```java

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;

import software.amazon.awssdk.core.SdkBytes;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.sagemaker.SageMakerRuntimeClient;

import software.amazon.awssdk.services.sagemaker.model.InvokeEndpointRequest;

import software.amazon.awssdk.services.sagemaker.model.InvokeEndpointResponse;

import software.amazon.awssdk.services.sagemaker.model.SageMakerException;

import java.util.Scanner;

public class RAGClientApplication {

private static final String ENDPOINT\_NAME = "your-endpoint-name";

private static final Region REGION = Region.US\_WEST\_2; // Adjust as per your region

public static void main(String[] args) {

SageMakerRuntimeClient sageMakerRuntimeClient = SageMakerRuntimeClient.builder()

.region(REGION)

.credentialsProvider(ProfileCredentialsProvider.create())

.build();

Scanner scanner = new Scanner(System.in);

System.out.println("Enter your query for the RAG model:");

String query = scanner.nextLine();

// Optional: Provide additional context or documents if necessary

// For now, we're assuming the RAG model can work with just the query or has internal retrieval logic

String context = "Optional context or related documents"; // Adjust as needed

try {

String response = invokeRAGModel(sageMakerRuntimeClient, query, context);

System.out.println("Response from LLM: " + response);

} catch (SageMakerException e) {

System.err.println("Failed to invoke RAG model: " + e.awsErrorDetails().errorMessage());

}

}

private static String invokeRAGModel(SageMakerRuntimeClient sageMakerClient, String query, String context) {

String inputPayload = "{\"query\": \"" + query + "\", \"context\": \"" + context + "\"}";

InvokeEndpointRequest request = InvokeEndpointRequest.builder()

.contentType("application/json")

.accept("application/json")

.endpointName(ENDPOINT\_NAME)

.body(SdkBytes.fromUtf8String(inputPayload))

.build();

InvokeEndpointResponse response = sageMakerClient.invokeEndpoint(request);

return response.body().asUtf8String();

}

}

```

### Explanation

- \*\*SageMakerRuntimeClient\*\*: The AWS SDK client used to communicate with the SageMaker endpoint.

- \*\*User Input\*\*: Captures a query from the user input using Java's `Scanner`.

- \*\*Invoke LLM\*\*: Uses the `invokeRAGModel` method to call the endpoint with the query and any additional context, such as optional related documents. The response from SageMaker is printed through standard output.

### Setup Instructions

1. \*\*Configure AWS SDK\*\*:

- Ensure AWS credentials are properly set up in your environment. This can be done through the AWS CLI or by setting up credentials manually in your script.

2. \*\*Replace Placeholders\*\*:

- Replace `ENDPOINT\_NAME` with your actual SageMaker endpoint name from the previous code example's deployment.

- Adjust `REGION` as per your SageMaker endpoint's AWS region.

3. \*\*Run the Client Application\*\*:

- Compile and run this Java program in an environment where AWS SDK for Java is configured.

- Enter a query prompt as requested by the program to interact with the RAG model.

### Note

- The successful functioning of this client depends on the correct setup and deployment of your SageMaker model endpoint.

- The context string is a placeholder for where you might include additional preprocessing logic to retrieve relevant documents dynamically, depending on your application's design.