Assignment-1

S3:

Bucket policy:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "PublicReadGetObject",
         "Effect": "Allow",
         "Principal": "*",
         "Action": "s3:GetObject",
         "Resource": "arn:aws:s3:::ccassignment1st/*"
      }
   ]
}
```

Lambdas:

A **Lambda function** is a piece of code that runs in the cloud without needing a server to operate it. Imagine it as a mini-program that does one specific job automatically when something triggers it, like an event or request. In AWS (Amazon Web Services), Lambda functions are part of what's called **serverless computing**, which means you don't have to worry about managing servers to run your code.

Key Points About Lambda Functions:

1. **On-Demand Execution**:

- A Lambda function only runs when it's needed. For example, in the chatbot assignment, a Lambda function activates whenever a user makes a request for restaurant recommendations.

2. **Automatic Triggering**:

- Lambda functions can be set to automatically trigger based on certain events, like receiving a message in SQS or receiving an API request. In this assignment, one Lambda function is triggered when a message is added to the SQS queue.

3. **Cost-Efficient**:

- You're only charged for the time the Lambda function runs, not for idle time or server upkeep. This makes it ideal for tasks that don't need to run continuously.

4. **Specific Jobs**:

- Each Lambda function is designed to perform a specific job. In this assignment, different Lambda functions handle tasks like managing the chatbot conversation, retrieving restaurant suggestions, and sending emails.

Simple Analogy:

Think of a Lambda function like a toaster in a restaurant kitchen:

- It only operates (runs) when you need it, say, when you put bread in it.

- It does one job—making toast—without needing you to monitor it constantly.

- It shuts off when it's done, so it's only using energy (or costing money) when it's actually toasting.

So, an AWS Lambda function works similarly: it waits, triggers to do its job when needed, and then shuts off until it's required again. This makes it perfect for running quick tasks in response to events in a cloud environment.

LF0 (API gateway):

```
import json
import boto3

def lambda_handler(event, context):
    # Initialize the Lex client
    client = boto3.client('lex-runtime')

    # Define CORS headers
    cors_headers = {
        'Access-Control-Allow-Origin': '*',  # Replace '*' with your specific origin in production
        'Access-Control-Allow-Headers': 'Content-Type',
        'Access-Control-Allow-Methods': 'OPTIONS,POST'
    }

    # Extract messages from the event body
    try:
        body = json.loads(event.get('body', '{}'))
    except json.JSONDecodeError:
        return {
            'statusCode': 400,
            'headers': cors_headers,
            'body': json.dumps({'message': 'Invalid JSON format in request body'})
```

```
    }

messages = body.get('messages', [])

if not messages:
    return {
        'statusCode': 400,
        'headers': cors_headers,
        'body': json.dumps({'message': 'No messages provided'})
    }

# Extract message details
message = messages[0]
user_id = message.get('unstructured', {}).get('id', 'defaultUser')
text = message.get('unstructured', {}).get('text', '')

if not text:
    return {
        'statusCode': 400,
        'headers': cors_headers,
        'body': json.dumps({'message': 'Empty message text'})
    }

# Send the message to Lex and get a response
try:
    lex_response = client.post_text(
```

```
            botName='BookHotel',
            botAlias='chatBot',
            userId=user_id,
            inputText=text
        )

        # Extract Lex response details
        lex_message = lex_response.get('message', 'I’m still under development. Please come back later.')

        # Format Lex response into BotResponse format
        bot_response = {
            'messages': [{
                'type': 'unstructured',  # Must match frontend expectation
                'unstructured': {
                    'text': lex_message
                }
            }]
        }

        return {
            'statusCode': 200,
            'headers': cors_headers,
            'body': json.dumps(bot_response)
        }

    except Exception as e:
```

```
    # Log the exception details for debugging (optional)
    print(f"Error processing message: {e}")

    return {
        'statusCode': 500,
        'headers': cors_headers,
        'body': json.dumps({'code': 500, 'message': 'Internal server error'})
    }
```

LF1:

1) Dining suggestions:

```
import json
import boto3
import uuid
import logging

# Initialize the Lex client
sqs = boto3.client('sqs')

# Set up logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def handle_lex_request(event):
    """Handles a request from Lex."""
    slots = event['currentIntent']['slots']
    location = slots.get('location')
    cuisine = slots.get('cuisine')
    dining_time = slots.get('dining_time')
    number_people = slots.get('number_people')
```

```
    email = slots.get('email')

    # Check if all slots are filled
    if all([location, cuisine, dining_time, number_people, email]):
        # Push the collected information to an SQS queue
        params = {
            'MessageBody': json.dumps({
                'location': location,
                'cuisine': cuisine,
                'dining_time': dining_time,
                'number_people': number_people,
                'email': email,
            }),
            'QueueUrl': 'https://sqs.us-east-
1.amazonaws.com/423623832978/chatBot',
        }

        try:
            sqs.send_message(**params)
            response = {
                'dialogAction': {
                    'type': 'Close',
                    'fulfillmentState': 'Fulfilled',
                    'message': {
                        'contentType': 'PlainText',
                        'content': "You're all set. Expect my suggestions shortly.
Have a good day.",
                    },
                },
            }
            return response
        except Exception as e:
            logger.error(f"Error sending message to SQS: {str(e)}")
            return {
                'dialogAction': {
                    'type': 'Close',
                    'fulfillmentState': 'Failed',
```

```
                'message': {
                    'contentType': 'PlainText',
                    'content': 'Failed to process your request.',
                },
            },
        }

    # If not all slots are filled, delegate back to Lex
    return {
        'dialogAction': {
            'type': 'Delegate',
            'slots': slots,
        },
    }

def lambda_handler(event, context):
    """Main Lambda handler for Lex."""
    logger.info(f"Received event: {json.dumps(event)}")

    if 'currentIntent' in event:
        # Request from Lex
        return handle_lex_request(event)

    return {
        'statusCode': 400,
        'body': json.dumps({'code': 400, 'message': 'Invalid request
format'}),
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }
```

2) <u>City validation:</u>

```
import json
import logging

# Set up logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def validate_location(location):
    """Validates if the given location is Manhattan."""
    if location.lower() == 'manhattan':
        return True
    return False

def elicit_slot(intent_name, slots, slot_to_elicit, message):
    """Informs Lex to ask for a specific slot (e.g., location) again."""
    return {
        'dialogAction': {
            'type': 'ElicitSlot',
            'intentName': intent_name,
            'slots': slots,
            'slotToElicit': slot_to_elicit,
            'message': {
                'contentType': 'PlainText',
                'content': message
            }
```

```
        }
    }

def lambda_handler(event, context):
    """Main Lambda handler to validate the location slot."""
    logger.info(f"Received event: {json.dumps(event)}")

    intent_name = event['currentIntent']['name']
    slots = event['currentIntent']['slots']
    location = slots.get('location')

    # If location is provided, validate it
    if location:
        if validate_location(location):
            # If valid (Manhattan), continue with the next step of the conversation
            return {
                'dialogAction': {
                    'type': 'Delegate',
                    'slots': slots
                }
            }
        else:
            # If invalid, ask for the location again
            return elicit_slot(intent_name, slots, 'location', f"Sorry, we only support Manhattan as a location. Please provide a valid location.")

    # If no location is provided, delegate back to Lex
```

```
    return {
        'dialogAction': {
            'type': 'Delegate',
            'slots': slots
        }
    }
```

LF2:

```
import json
import boto3
import botocore.session
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
import urllib3
import random

# Initialize AWS clients
sqs = boto3.client('sqs')
dynamodb = boto3.resource('dynamodb')
ses = boto3.client('ses', region_name='us-east-1')  # Adjust region

QUEUE_URL = 'https://sqs.us-east-1.amazonaws.com/423623832978/chatBot'
DYNAMODB_TABLE = 'yelp-restaurants'
SES_SENDER_EMAIL = 'dhairyatemp007@gmail.com'
REGION = 'us-east-1'  # OpenSearch region
```

```
OPENSEARCH_ENDPOINT = "https://search-restaurants-index-3ggdee5zyaddryvtjypdfslt2m.aos.us-east-1.on.aws"
OPENSEARCH_INDEX = 'restaurants'

# Create a botocore session
session = botocore.session.get_session()
credentials = session.get_credentials()

def get_random_restaurant(cuisine):
    """Query OpenSearch for restaurants of the given cuisine."""
    method = 'GET'
    endpoint = f"{OPENSEARCH_ENDPOINT}/{OPENSEARCH_INDEX}/_search"

    # OpenSearch query to match the cuisine
    query = {
        "size": 10,  # Retrieve up to 10 restaurants to select from
        "query": {
            "match": {
                "cuisine": cuisine.lower()
            }
        }
    }
    body = json.dumps(query)

    # Prepare headers
    headers = {
        'Content-Type': 'application/json',
```

```
        'Host': OPENSEARCH_ENDPOINT.replace('https://', '').replace('http://', '')
    }

    # Create a botocore AWSRequest
    request = AWSRequest(method=method, url=endpoint, data=body,
headers=headers)

    # Sign the request using SigV4Auth
    SigV4Auth(credentials, 'es', REGION).add_auth(request)

    # Extract the signed headers
    signed_headers = dict(request.headers.items())

    # Send the request using urllib3
    http = urllib3.PoolManager()
    response = http.request(
        method,
        endpoint,
        body=body,
        headers=signed_headers
    )

    if response.status != 200:
        raise Exception(f"OpenSearch query failed: {response.data.decode('utf-
8')}")

    results = json.loads(response.data.decode('utf-8')).get('hits', {}).get('hits', [])
```

```
    if not results:
        raise ValueError(f"No restaurants found for cuisine: {cuisine}")

    # Pick a random restaurant ID from the results
    restaurant_id = random.choice(results)['_source']['restaurant_id']
    return restaurant_id

def get_restaurant_details(restaurant_id):
    """Fetch restaurant details from DynamoDB using the restaurant_id."""
    table = dynamodb.Table(DYNAMODB_TABLE)
    response = table.get_item(Key={'business_id': restaurant_id})

    if 'Item' not in response:
        raise ValueError(f"Restaurant ID {restaurant_id} not found in DynamoDB.")

    return response['Item']

def send_email(to_email, subject, body):
    """Send an email using SES."""
    ses.send_email(
        Source=SES_SENDER_EMAIL,
        Destination={'ToAddresses': [to_email]},
        Message={
            'Subject': {'Data': subject},
            'Body': {'Text': {'Data': body}}
        }
```

```
    )

def lambda_handler(event, context):
    """Main handler function for the Lambda."""
    # Pull a message from the SQS queue
    response = sqs.receive_message(
        QueueUrl=QUEUE_URL,
        MaxNumberOfMessages=1
    )

    messages = response.get('Messages', [])
    if not messages:
        print("No messages in the queue.")
        return {"statusCode": 200, "body": "No messages to process."}

    message = messages[0]
    body = json.loads(message['Body'])

    # Extract relevant data from the SQS message
    location = body.get('location', 'Unknown')
    cuisine = body['cuisine']  # Mandatory field
    dining_time = body.get('dining_time', 'N/A')
    num_people = body.get('number_people', 'N/A')
    email = body['email']  # Mandatory field

    try:
```

```
    # Get a random restaurant recommendation
    restaurant_id = get_random_restaurant(cuisine)
    restaurant = get_restaurant_details(restaurant_id)

    # Format the email content
    subject = f"Your {cuisine} Restaurant Recommendations"
    email_body = (
        f"Hello!\n\nHere is a {cuisine} restaurant suggestion for your dining in {location}:\n\n"
        f"Name: {restaurant['name']}\n"
        f"Address: {restaurant['address']}\n"
        f"Rating: {restaurant['rating']} stars\n"
        f"Number of Reviews: {restaurant['review_count']}\n"
        f"Dinner Time: {dining_time} for {num_people} people\n\n"
        f"Enjoy your meal!"
    )

    # Send the email
    send_email(email, subject, email_body)

    # Delete the processed message from the SQS queue
    sqs.delete_message(
        QueueUrl=QUEUE_URL,
        ReceiptHandle=message['ReceiptHandle']
    )
    print(f"Successfully sent email to {email}.")
    return {"statusCode": 200, "body": "Email sent successfully"}
```

```
except Exception as e:
    print(f"Error: {str(e)}")
    return {"statusCode": 500, "body": f"Failed to process message: {str(e)}"}
```

Swagger api yaml file:

```
swagger: '2.0'
info:
  title: AI Customer Service API
  description: 'AI Customer Service application, built during the Cloud and
Big Data course at Columbia University.'
  version: 1.0.0
schemes:
  - https
basePath: /v1
produces:
  - application/json
paths:
  /chatbot:
    post:
      summary: The endpoint for the Natural Language Understanding API.
      description: |
        This API takes in one or more messages from the client and returns
        one or more messages as a response. The API leverages the NLP
        backend functionality, paired with state and profile information
        and returns a context-aware reply.
      tags:
        - NLU
      operationId: sendMessage
      produces:
```

```
      - application/json
    parameters:
      - name: body
        in: body
        required: true
        schema:
          $ref: '#/definitions/BotRequest'
    responses:
      '200':
        description: A Chatbot response
        schema:
          $ref: '#/definitions/BotResponse'
      '403':
        description: Unauthorized
        schema:
          $ref: '#/definitions/Error'
      '500':
        description: Unexpected error
        schema:
          $ref: '#/definitions/Error'
definitions:
  BotRequest:
    type: object
    properties:
      messages:
        type: array
        items:
          $ref: '#/definitions/Message'
  BotResponse:
    type: object
    properties:
      messages:
        type: array
        items:
          $ref: '#/definitions/Message'
  Message:
    type: object
```

```
  properties:
    type:
      type: string
    unstructured:
      $ref: '#/definitions/UnstructuredMessage'
UnstructuredMessage:
  type: object
  properties:
    id:
      type: string
    text:
      type: string
    timestamp:
      type: string
      format: datetime
Error:
  type: object
  properties:
    code:
      type: integer
      format: int32
    message:
      type: string
```