**Project Title**: Event-Driven Stock Tracker with AWS Lambda, DynamoDB, and SNS

### ****Setup and Requirements****

1. **AWS Account**: Ensure you have access to the AWS Management Console.
2. **AWS CLI Installed**: Configure it with your credentials.
3. **IAM Roles**:
   * A role with access to Lambda, DynamoDB, and SNS.
   * Attach policies: AWSLambdaBasicExecutionRole, AmazonDynamoDBFullAccess, AmazonSNSFullAccess.

### ****Step 1: Create a DynamoDB Table****

1. Go to the DynamoDB service in the AWS Console.
2. Click **"Create Table"**:
   * **Table name**: StockTracker
   * **Primary key**: StockID (String)
3. Leave the rest of the settings as default and create the table.

### ****Step 2: Create an SNS Topic****

1. Go to the SNS service in the AWS Console.
2. Click **"Create Topic"**:
   * **Type**: Standard
   * **Name**: StockAlerts
3. Click **"Create Subscription"**:
   * **Protocol**: Email
   * **Endpoint**: Provide your email address.
4. Confirm the subscription through the email you'll receive.

### ****Step 3: Write and Deploy the Lambda Function****

1. Go to the Lambda service in the AWS Console.
2. Create a new Lambda function:
   * **Function name**: StockHandler
   * **Runtime**: Python 3.9 (or any supported runtime)
   * **Permissions**: Attach the IAM role created earlier.
3. Use the following Lambda function code:

import boto3

import json

import os

# Initialize DynamoDB and SNS clients

dynamodb = boto3.client('dynamodb')

sns = boto3.client('sns')

# Environment variables for DynamoDB table and SNS topic

TABLE\_NAME = os.environ['TABLE\_NAME']

SNS\_TOPIC\_ARN = os.environ['SNS\_TOPIC\_ARN']

def lambda\_handler(event, context):

# Parse incoming event data

stock\_id = event['StockID']

stock\_value = event['Value']

# Store stock data in DynamoDB

dynamodb.put\_item(

TableName=TABLE\_NAME,

Item={

'StockID': {'S': stock\_id},

'Value': {'N': str(stock\_value)}

}

)

# Trigger an alert if the stock value crosses a threshold

threshold = 100 # Example threshold

if stock\_value > threshold:

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=f"Alert! Stock {stock\_id} has exceeded the threshold with a value of {stock\_value}.",

Subject="Stock Value Alert"

)

return {

'statusCode': 200,

'body': json.dumps('Stock processed successfully!')

}

 Set environment variables for the Lambda function:

* **TABLE\_NAME**: StockTracker
* **SNS\_TOPIC\_ARN**: ARN of the StockAlerts topic created earlier.

 Deploy the Lambda function.

### ****Step 4: Test the Lambda Function****

1. Go to the Lambda function and click **"Test"**.
2. Create a new test event with the following payload:

{

"StockID": "AAPL",

"Value": 120

}

1. Execute the test:
   * If the value exceeds the threshold (100 in this example), you'll receive an email notification through SNS.
   * The stock data will also be stored in DynamoDB.

### ****Verification****

* **DynamoDB**: Check the StockTracker table for entries.
* **SNS**: Confirm that the email notification is received when conditions are met.

Part:2

**Enhanced Project Title**: Advanced Event-Driven Stock Tracker with Analytics and Notifications

**New Features to Add**

1. **Stock Historical Tracking**:
   * Maintain a history of stock values in the DynamoDB table.
   * Calculate the average value of a stock over time.
2. **Trigger Lambda from API Gateway**:
   * Create a REST API using API Gateway to allow external clients to submit stock data.
3. **S3 Integration for Data Backup**:
   * Periodically back up stock data to an S3 bucket using another Lambda function triggered by Amazon EventBridge.
4. **CloudWatch Metrics and Alerts**:
   * Publish custom metrics to CloudWatch (e.g., stock submissions count).
   * Trigger alerts based on metrics (e.g., no stock data received in the last hour).
5. **Enhanced Notification System with Multiple Channels**:
   * Send notifications to multiple SNS protocols (e.g., email, SMS).

**Updated Architecture Diagram**

1. **API Gateway** → **Lambda Function** (Data Ingestion)
2. **DynamoDB** (Data Storage)
3. **Lambda Function** (Backup to S3)
4. **CloudWatch** (Metrics and Alerts)
5. **SNS** (Email and SMS Notifications)

### ****Implementation Steps****

#### ****Step 1: Update DynamoDB Schema for Historical Tracking****

* Add a **Sort Key** to your table:
  + **Primary key**: StockID (String)
  + **Sort key**: Timestamp (String)
* Use a unique timestamp (e.g., ISO 8601) for each stock entry to track historical values

#### ****Step 2: API Gateway Integration****

1. Create a REST API in **API Gateway**.
2. Configure a **POST method** for a resource (e.g., /stocks).
3. Integrate the API with the Lambda function created earlier (StockHandler).
4. Test the API with tools like **Postman** or **curl**.

Below are the steps only for API Gateway—you have to cont. with step -3

###################----API-Gateway—steps—starts--######################

### ****Step 1: Open API Gateway****

1. Log in to the AWS Management Console.
2. Navigate to **API Gateway**.
3. Select **Create API** and then choose **REST API**:
   * **API Type**: Choose **REST API (Private or Regional)**.
   * **API Name**: Enter a meaningful name, e.g., StockTrackerAPI.
   * **Description**: Provide a brief description, e.g., "API for Stock Tracking Application".
4. Click **Create API**.

### ****Step 2: Create a Resource****

1. In the left navigation pane, click on the **Resources** section.
2. Click on **Actions** → **Create Resource**.
3. Configure the resource:
   * **Resource Name**: stocks
   * **Resource Path**: It will automatically be /stocks.
4. Click **Create Resource**.

### ****Step 3: Add a POST Method to the Resource****

1. Select the /stocks resource in the resources tree.
2. Click on **Actions** → **Create Method**.
3. Select **POST** from the dropdown menu and click the checkmark to confirm.
4. Choose the **Integration Type**:
   * **Integration Type**: Select **Lambda Function**.
   * **Lambda Function**: Enter the name of your Lambda function (e.g., StockHandler).
5. Click **Save**. AWS will prompt you to give API Gateway permissions to invoke your Lambda function. Click **OK** to allow this.

### ****Step 4: Deploy the API****

1. In the left navigation pane, click on **Actions** → **Deploy API**.
2. Select the **Deployment Stage**:
   * If no stage exists, create one (e.g., prod).
   * **Stage Name**: Enter prod (or any preferred name).
3. Click **Deploy**.

### ****Step 5: Retrieve the Endpoint URL****

1. After deployment, you will see the **Invoke URL** for your API, which will look something like:

php

Copy code

https://<api-id>.execute-api.<region>.amazonaws.com/prod/stocks

1. Note this URL; you will use it to test the API.

### ****Step 6: Test the API****

You can test the API using tools like **Postman**, **curl**, or directly in your application.

#### ****Using Postman****

1. Open Postman and create a new request.
2. Select the **POST** method.
3. Enter the API endpoint:

php

Copy code

https://<api-id>.execute-api.<region>.amazonaws.com/prod/stocks

1. Go to the **Body** tab, select **raw**, and choose **JSON** format.
2. Add a JSON payload similar to the one below:

json

Copy code

{

"StockID": "AAPL",

"Value": 120

}

1. Click **Send**. You should receive a response from your Lambda function.

#### ****Using curl****

1. Open your terminal.
2. Run the following command (replace <api-endpoint> with your API URL):

bash

Copy code

curl -X POST <api-endpoint> \

-H "Content-Type: application/json" \

-d '{"StockID": "AAPL", "Value": 120}'

1. You should see a JSON response from the Lambda function.

###################---APIGateway—steps-ends-----#########################

#### ****Step 3: Update Lambda Function for Historical Tracking and Analytics****

Replace the earlier Lambda function with the following updated version:

import boto3

import json

import os

from datetime import datetime

# Initialize AWS clients

dynamodb = boto3.client('dynamodb')

sns = boto3.client('sns')

cloudwatch = boto3.client('cloudwatch')

# Environment variables

TABLE\_NAME = os.environ['TABLE\_NAME']

SNS\_TOPIC\_ARN = os.environ['SNS\_TOPIC\_ARN']

THRESHOLD = int(os.environ['THRESHOLD'])

def lambda\_handler(event, context):

# Parse event data

stock\_id = event['StockID']

stock\_value = event['Value']

timestamp = datetime.utcnow().isoformat()

# Store historical stock data in DynamoDB

dynamodb.put\_item(

TableName=TABLE\_NAME,

Item={

'StockID': {'S': stock\_id},

'Timestamp': {'S': timestamp},

'Value': {'N': str(stock\_value)}

}

)

# Publish custom CloudWatch metric

cloudwatch.put\_metric\_data(

Namespace='StockTracker',

MetricData=[

{

'MetricName': 'StockSubmissions',

'Value': 1,

'Unit': 'Count'

}

]

)

# Analyze historical data

response = dynamodb.query(

TableName=TABLE\_NAME,

KeyConditionExpression='StockID = :stock\_id',

ExpressionAttributeValues={':stock\_id': {'S': stock\_id}}

)

values = [int(item['Value']['N']) for item in response['Items']]

avg\_value = sum(values) / len(values)

# Trigger alert if threshold is crossed

if stock\_value > THRESHOLD:

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=(

f"Alert! Stock {stock\_id} has exceeded the threshold. "

f"Current value: {stock\_value}. Average value: {avg\_value:.2f}"

),

Subject="Stock Value Alert"

)

return {

'statusCode': 200,

'body': json.dumps(f"Stock {stock\_id} processed successfully!")

}

**New Features**:

* Historical tracking with timestamps.
* Calculation of average stock value.
* CloudWatch custom metrics for submissions.

#### ****Step 4: S3 Backup Integration****

1. Create an S3 bucket (e.g., stock-tracker-backup).
2. Create a new Lambda function (BackupToS3) that:
   * Queries all data from DynamoDB.
   * Saves the data to an S3 object (e.g., backup-<date>.json).
3. Trigger this Lambda function periodically using **EventBridge**.

**Backup Lambda Code**:

import boto3

import json

from datetime import datetime

dynamodb = boto3.client('dynamodb')

s3 = boto3.client('s3')

TABLE\_NAME = os.environ['TABLE\_NAME']

BUCKET\_NAME = os.environ['BUCKET\_NAME']

def lambda\_handler(event, context):

# Scan all items from DynamoDB

response = dynamodb.scan(TableName=TABLE\_NAME)

data = response['Items']

# Save data to S3 as JSON

backup\_file = f"backup-{datetime.utcnow().isoformat()}.json"

s3.put\_object(

Bucket=BUCKET\_NAME,

Key=backup\_file,

Body=json.dumps(data)

)

return {

'statusCode': 200,

'body': json.dumps('Backup successful!')

}

#### ****Step 5: Enhanced Notifications****

1. Add **SMS subscription** to your SNS topic.
2. In the Lambda function, send different types of alerts based on the stock value:
   * **Critical alert**: SMS and email.
   * **Info alert**: Email only.

#### ****Step 6: Monitor with CloudWatch Alarms****

1. Create a CloudWatch alarm for the **StockSubmissions** metric:
   * Trigger an alarm if no submissions are received for an hour.
2. Set the alarm to send an SNS notification when triggered.

########################--for-Event-Bridge--------starts------###############

**Step 1: Create an S3 Bucket**

1. Log in to the AWS Management Console.
2. Navigate to the **S3** service.
3. Click **Create bucket**.
4. Configure the bucket:
   * **Bucket name**: stock-tracker-backup (or a unique name).
   * **Region**: Choose a region.
   * Leave other settings as default unless specific configurations are needed.
5. Click **Create bucket**.

**Step 2: Create a Backup Lambda Function**

1. Navigate to the **Lambda** service and click **Create Function**.
2. Configure the Lambda function:
   * **Function name**: BackupToS3.
   * **Runtime**: Python 3.9 (or another preferred runtime).
   * **Role**: Attach or create an IAM role with these permissions:
     + AmazonDynamoDBReadOnlyAccess
     + AmazonS3FullAccess
     + AWSLambdaBasicExecutionRole
3. Deploy the following code for the Lambda function:

import boto3

import json

from datetime import datetime

# Initialize AWS clients

dynamodb = boto3.client('dynamodb')

s3 = boto3.client('s3')

# Environment variables

TABLE\_NAME = 'StockTracker' # Replace with your DynamoDB table name

BUCKET\_NAME = 'stock-tracker-backup' # Replace with your S3 bucket name

def lambda\_handler(event, context):

try:

# Scan all data from the DynamoDB table

response = dynamodb.scan(TableName=TABLE\_NAME)

data = response['Items']

# Generate a backup file name with a timestamp

timestamp = datetime.utcnow().strftime('%Y-%m-%dT%H-%M-%SZ')

file\_name = f"backup-{timestamp}.json"

# Convert data to JSON and save to S3

s3.put\_object(

Bucket=BUCKET\_NAME,

Key=file\_name,

Body=json.dumps(data),

ContentType='application/json'

)

return {

'statusCode': 200,

'body': json.dumps(f"Backup successful! Saved as {file\_name}.")

}

except Exception as e:

return {

'statusCode': 500,

'body': json.dumps(f"Error during backup: {str(e)}")

}

1. Set the following environment variables for the Lambda function:
   * **TABLE\_NAME**: StockTracker
   * **BUCKET\_NAME**: stock-tracker-backup
2. Save and test the function with a manual invocation.

**Step 3: Create an EventBridge Rule**

1. Navigate to the **EventBridge** service in the AWS Management Console.
2. Click **Rules** in the left menu and then **Create rule**.
3. Configure the rule:
   * **Name**: DynamoDBBackupSchedule.
   * **Description**: "Periodically triggers the backup Lambda function."
   * **Event bus**: Default.
   * **Rule type**: **Schedule**.
4. Under **Define pattern**, choose **Schedule**:
   * Enter a cron expression, e.g., cron(0 0 \* \* ? \*) for a daily backup at midnight UTC. [Learn more about cron expressions in AWS](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-create-rule-schedule.html).
5. Click **Next**.

**Step 4: Add the Lambda Target**

1. Under **Select targets**, choose **AWS service** → **Lambda function**.
2. In the **Function** dropdown, select the BackupToS3 Lambda function.
3. Click **Next** and review the settings.
4. Click **Create rule**.

**Step 5: Verify the Integration**

1. Wait for the scheduled time or manually invoke the EventBridge rule to test it:
   * Go to **Rules** in EventBridge.
   * Select the DynamoDBBackupSchedule rule.
   * Click **Actions** → **Test** to trigger the rule.
2. Check the S3 bucket (stock-tracker-backup) for the newly created backup file.

**Step 6: Monitor and Logs**

* **CloudWatch Logs**:
  + Navigate to the **CloudWatch Logs** section.
  + Review the logs generated by the BackupToS3 Lambda function for debugging or validation.
* **S3**:
  + Verify that the backup files are uploaded with the correct timestamped filenames.

#########################--for-event-bridge----ends---------#####################