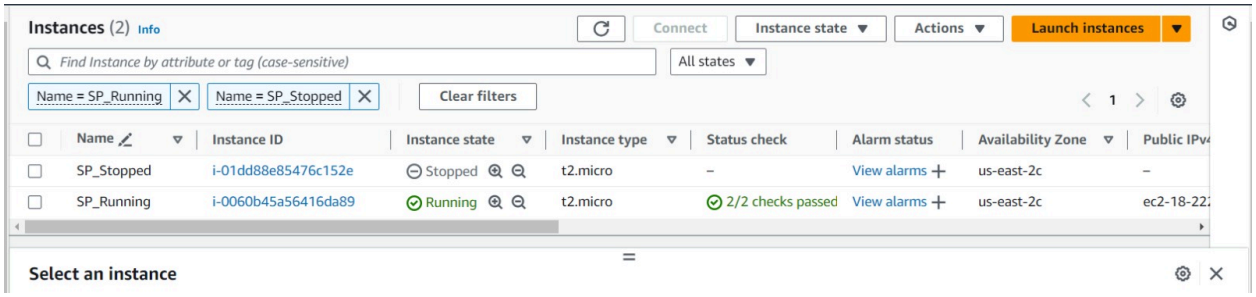
Graded Assignment On Serverless Architecture

Git Repository link : https://github.com/igSpanser/Assignment_Serverless_Architecture.git

Assignment 1:
Automated Instance Management Using AWS Lambda and Boto3.
In this assignment, you will gain hands-on experience with AWS Lambda and Boto3, Amazon's SDK for Python. You will create a Lambda function that will automatically manage EC2 instances based on their tags.

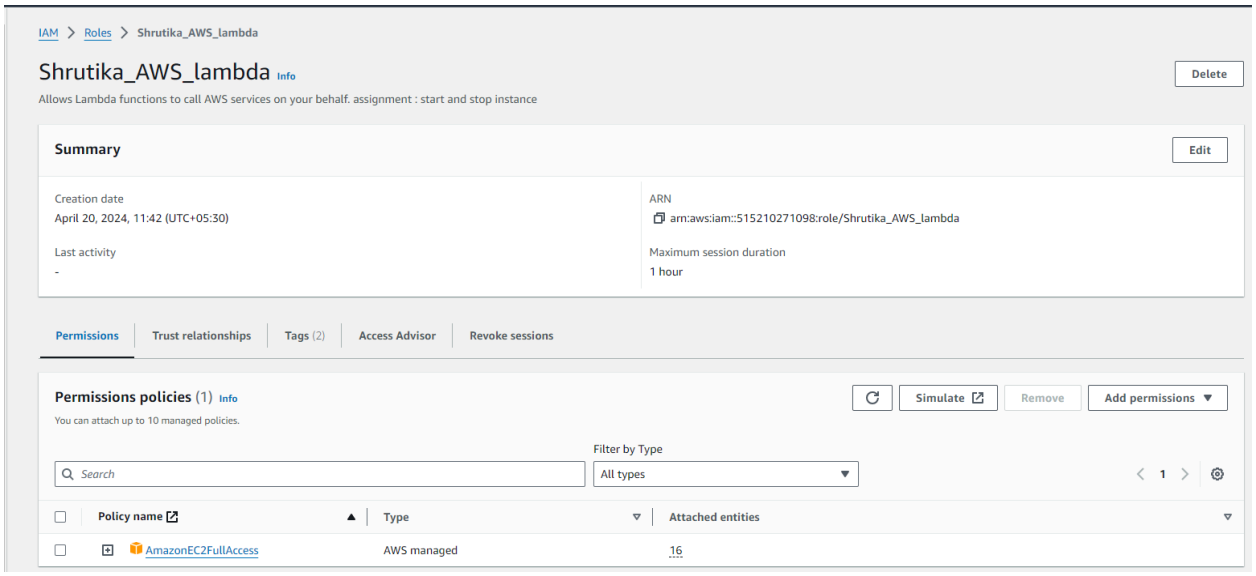
1. EC2 Setup:

- Navigate to the EC2 dashboard and create two new t2.micro instances (or any other available free-tier type).
- Tag the first instance with a key `Action` and value `Auto-Stop`.
- Tag the second instance with a key `Action` and value `Auto-Start`.



2. Lambda IAM Role:

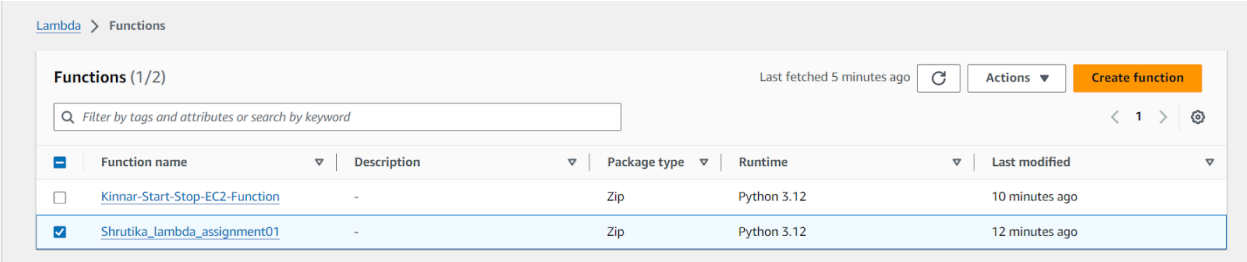
- In the IAM dashboard, create a new role for Lambda.
- Attach the `AmazonEC2FullAccess` policy to this role. (Note: In a real-world scenario, you would want to limit permissions for better security.)



3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.

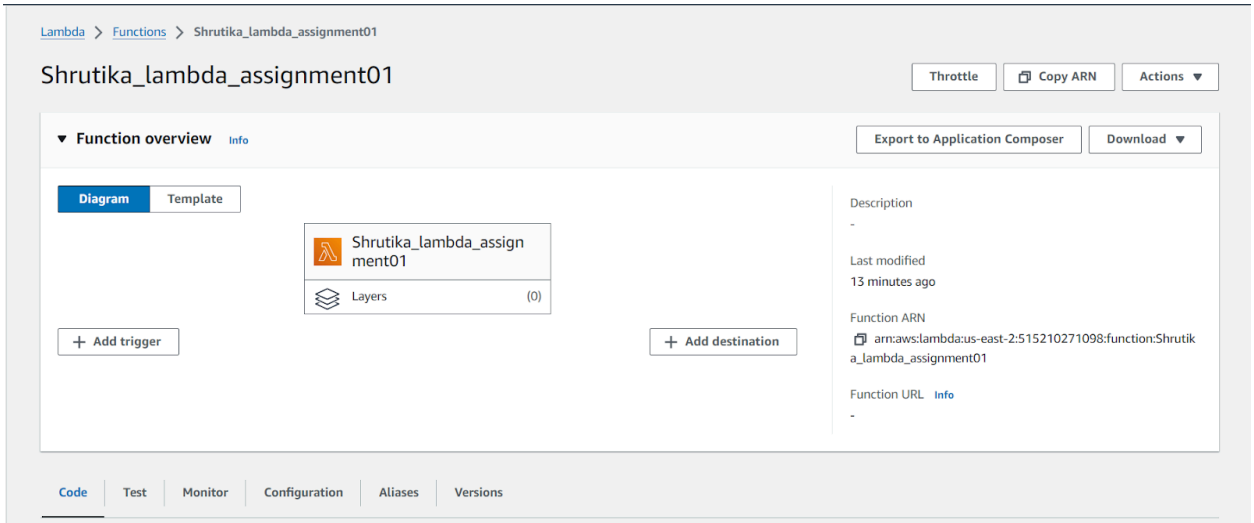
- ➢ Go to the AWS Management Console and sign in with your AWS account credentials. Navigate to Lambda and Click on the "Create function" button.
- ➢ Choose the option to author from scratch or use a blueprint. For your case, you'll be authoring from scratch.
- ➢ Choose the runtime. Since you mentioned Python 3.x, select the Python runtime version you prefer.
- ➢ Choose an existing role or create a new role with permissions for your Lambda function. Since you're working with EC2 instances, you'll need permissions to describe EC2 instances and perform actions on them. Make sure the role you choose or create has the necessary permissions.
- ➢ If you need more granular control over permissions, you can create a custom IAM role with specific policies attached to it.

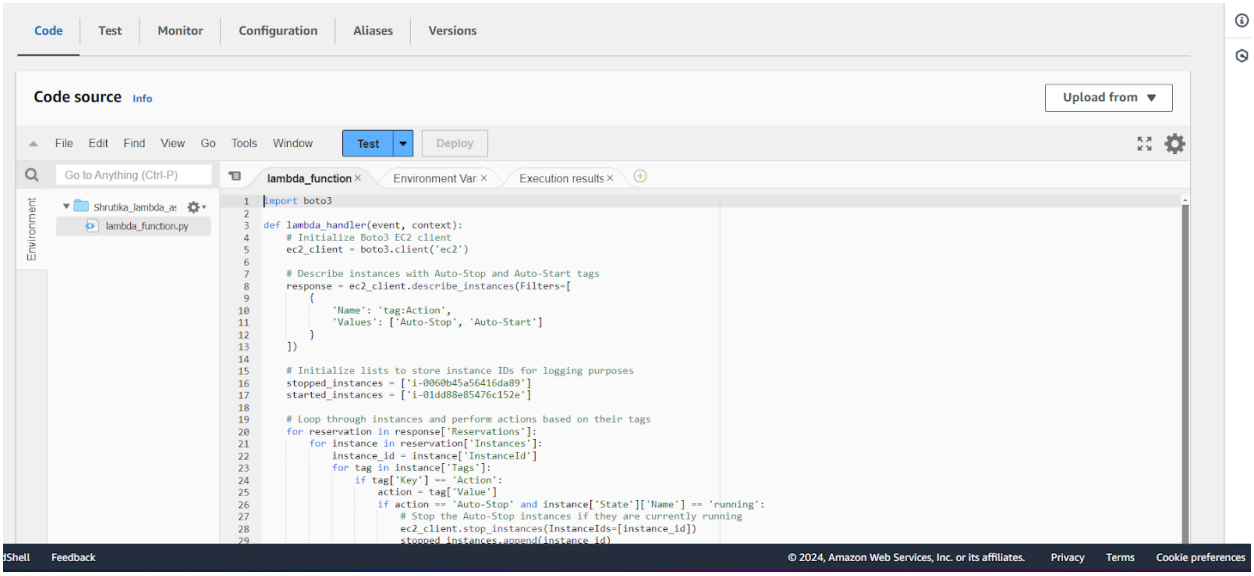


- Write the Boto3 Python script to:

1. Initialize a boto3 EC2 client.

2. Describe instances with `Auto-Stop` and `Auto-Start` tags.
3. Stop the `Auto-Stop` instances and start the `Auto-Start` instances.
4. Print instance IDs that were affected for logging purposes.





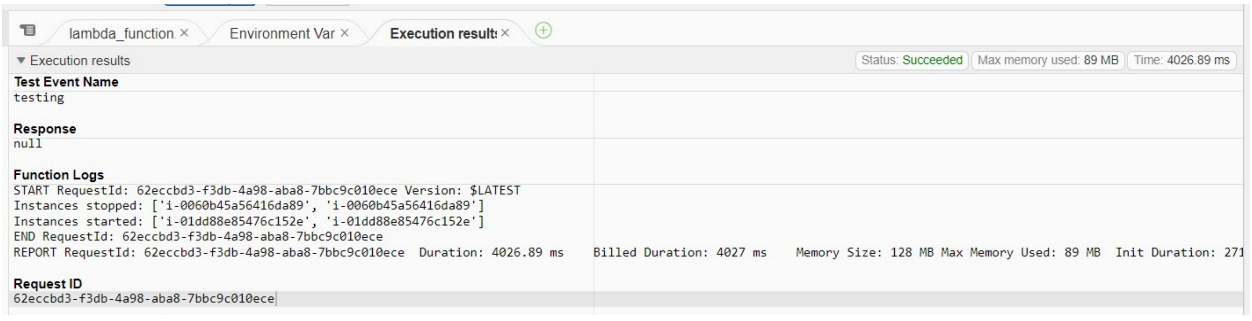
Python Script :

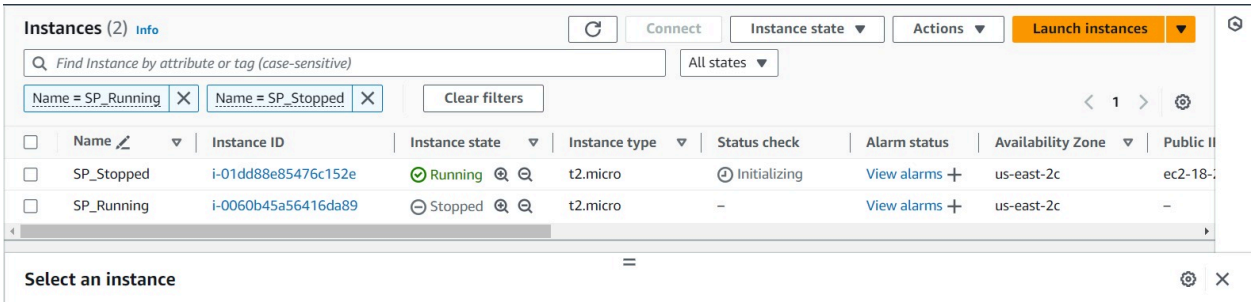
Assignment01.py > lambda_handler

```python
import boto3

def lambda_handler(event, context):
    # Initialize Boto3 EC2 client
    ec2_client = boto3.client('ec2')

    # Describe instances with Auto-Stop and Auto-Start tags
    response = ec2_client.describe_instances(Filters=[
        {
            'Name': 'tag:Action',
            'Values': ['Auto-Stop', 'Auto-Start']
        }
    ])

    # Initialize lists to store instance IDs for logging purposes
    stopped_instances = ['i-0060b45a56416da89']
    started_instances = ['i-01dd88e85476c152e']

    # Loop through instances and perform actions based on their tags
    for reservation in response['Reservations']:
        for instance in reservation['Instances']:
            instance_id = instance['InstanceId']
            for tag in instance['Tags']:
                if tag['Key'] == 'Action':
                    action = tag['Value']
                    if action == 'Auto-Stop' and instance['State']['Name'] == 'running':
                        # Stop the Auto-Stop instances if they are currently running
                        ec2_client.stop_instances(InstanceIds=[instance_id])
                        stopped_instances.append(instance_id)
                    elif action == 'Auto-Start' and instance['State']['Name'] == 'stopped':
                        # Start the Auto-Start instances if they are currently stopped
                        ec2_client.start_instances(InstanceIds=[instance_id])
                        started_instances.append(instance_id)

    # Print instance IDs that were affected for logging purposes
    print("Instances stopped:", stopped_instances)
    print("Instances started:", started_instances)

# For testing locally
if __name__ == "__main__":
    lambda_handler(None, None)
```

4. Manual Invocation:

- After saving your function, manually trigger it.
- Go to the EC2 dashboard and confirm that the instances' states have changed according to their tags.





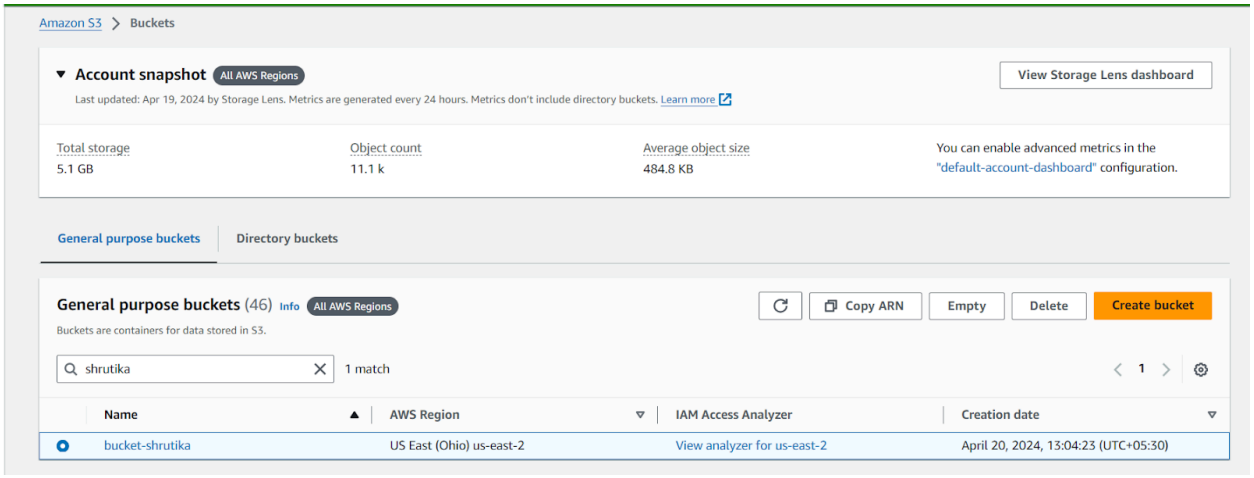
Assignment 2 :

Automated S3 Bucket Cleanup Using AWS Lambda and Boto3.

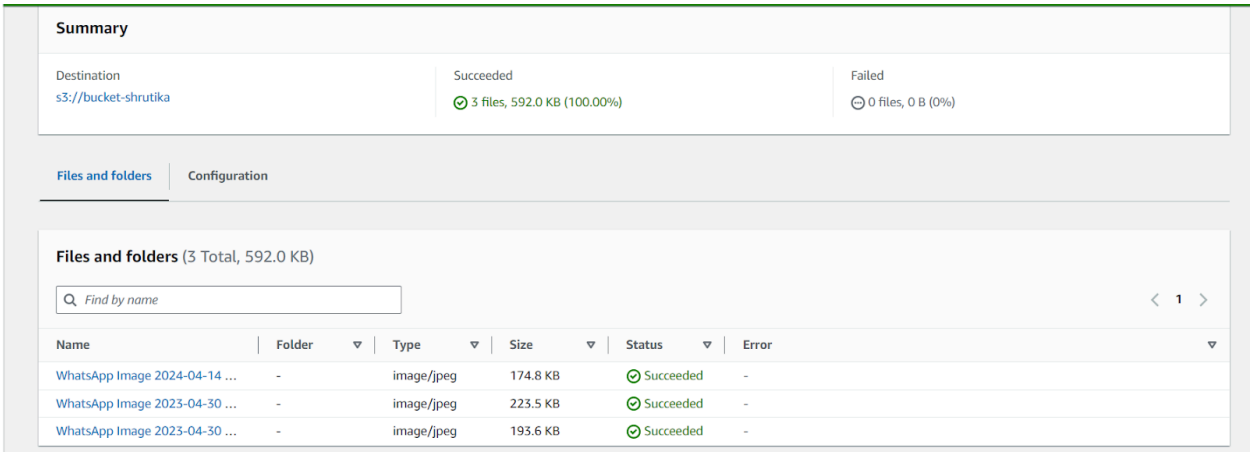
To gain experience with AWS Lambda and Boto3 by creating a Lambda function that will automatically clean up old files in an S3 bucket.

1. S3 Setup:

- Navigate to the S3 dashboard and create a new bucket.



- Upload multiple files to this bucket, ensuring that some files are older than 30 days (you may need to adjust your system's date temporarily for this or use old files).



(Steps 2&3 will be same as of first assignment)

2. Lambda IAM Role:

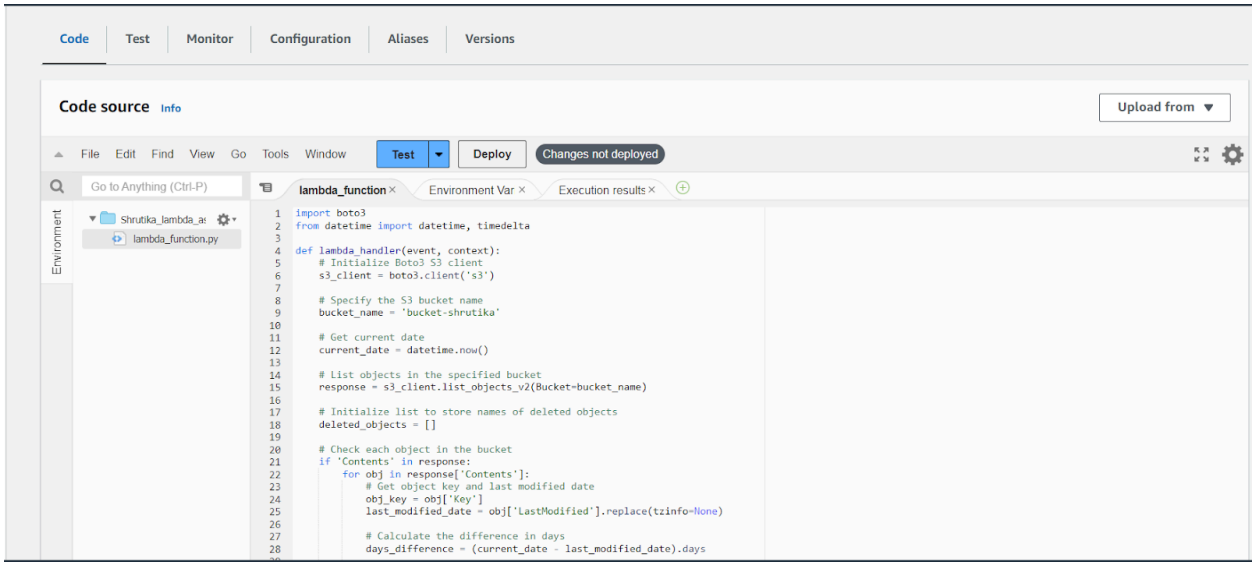
- In the IAM dashboard, create a new role for Lambda.
- Attach the `AmazonS3FullAccess` policy to this role. (Note: For enhanced security in real-world scenarios, use more restrictive permissions.)

3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.
- Choose Python 3.x as the runtime.
- Assign the IAM role created in the previous step.

- Write the Boto3 Python script to:

1. Initialize a boto3 S3 client.
2. List objects in the specified bucket.
3. Delete objects older than 30 days.
4. Print the names of deleted objects for logging purposes.



Python Script :

```python
import boto3
from datetime import datetime, timedelta

def lambda_handler(event, context):
    # Initialize Boto3 S3 client
    s3_client = boto3.client('s3')

    # Specify the S3 bucket name
    bucket_name = 'bucket-shrutika'

    # Get current date
    current_date = datetime.now()

    # List objects in the specified bucket
    response = s3_client.list_objects_v2(Bucket=bucket_name)

    # Initialize list to store names of deleted objects
    deleted_objects = []

    # Check each object in the bucket
    if 'Contents' in response:
        for obj in response['Contents']:
            # Get object key and last modified date
            obj_key = obj['Key']
            last_modified_date = obj['LastModified'].replace(tzinfo=None)

            # Calculate the difference in days
            days_difference = (current_date - last_modified_date).days

            # Delete objects older than 30 days
            if days_difference > 30:
                s3_client.delete_object(Bucket=bucket_name, Key=obj_key)
                deleted_objects.append(obj_key)

    # Print the names of deleted objects for logging purposes
    print("Deleted objects:", deleted_objects)
```

4. Manual Invocation:

- After saving your function, manually trigger it.
- Go to the S3 dashboard and confirm that only files newer than 30 days remain.

Assignment 3:
Automated RDS Snapshot Using AWS Lambda and Boto3.
To become familiar with automating RDS tasks using AWS Lambda and Boto3. You will create a Lambda function that takes a snapshot of an RDS instance.

What is RDS :
Amazon RDS (Relational Database Service) is a managed relational database service provided by Amazon Web Services (AWS). It enables you to easily set up, operate, and scale relational databases in the cloud without managing the infrastructure.

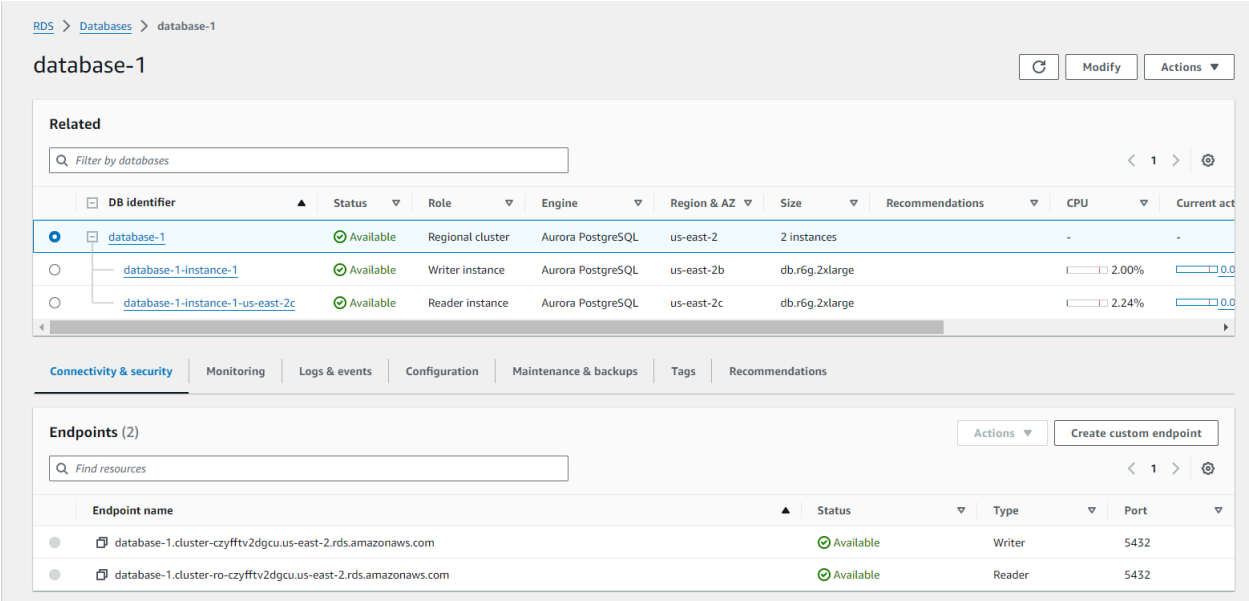
Key features of Amazon RDS include:

- **Managed Servic**e: Amazon RDS handles routine database tasks such as provisioning, patching, backup, recovery, and scaling, allowing you to focus on your application development instead of managing database infrastructure.
- **Multiple Database Engines:** Amazon RDS supports several popular database engines, including Amazon Aurora (a MySQL and PostgreSQL-compatible database engine developed by AWS), MySQL, PostgreSQL, MariaDB, Oracle, and Microsoft SQL Server. This enables you to choose the database engine that best fits your application requirements.
- **Automatic Backups and Point-in-Time Recovery:** Amazon RDS automatically backs up your database and retains backups for a specified retention period. You can also perform point-in-time recovery to restore your database to any specific point within the retention period.
- **High Availability and Fault Tolerance:** Amazon RDS provides high availability and fault tolerance for your databases by offering Multi-AZ (Multi-Availability Zone) deployments. In a Multi-AZ deployment, Amazon RDS automatically replicates your database across multiple Availability Zones within a region to provide redundancy and failover capabilities.
- **Scalability:** Amazon RDS allows you to easily scale your database instance vertically (by adjusting instance size) or horizontally (by adding read replicas). This enables you to handle increasing workloads and traffic demands without downtime.
- **Security**: Amazon RDS integrates with AWS Identity and Access Management (IAM) to control access to your databases. It also supports encryption at rest and in transit to enhance data security.

➢ **Monitoring and Metrics**: Amazon RDS provides built-in monitoring and metrics through Amazon CloudWatch, allowing you to monitor database performance, set alarms, and troubleshoot issues.

1. RDS Setup:

- Navigate to the RDS dashboard and create a new RDS instance (use the free tier, if available).
- Note the name of the instance.

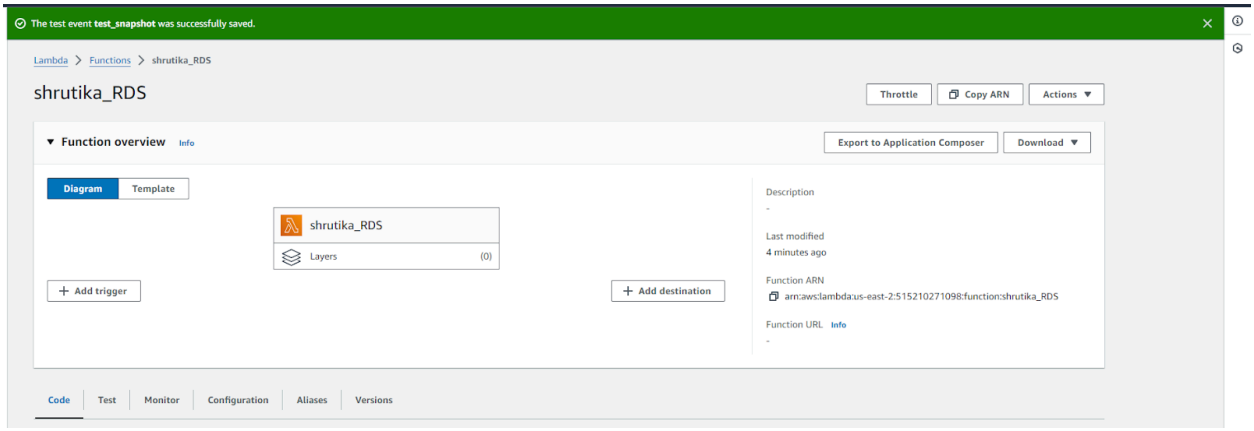


2. Lambda IAM Role:

- In the IAM dashboard, create a new role for Lambda.
- Attach the `AmazonRDSFullAccess` policy to this role. (Note: Always practice the principle of least privilege in real-world scenarios.)

3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.



- Choose Python 3.x as the runtime.
- Assign the IAM role created in the previous step.

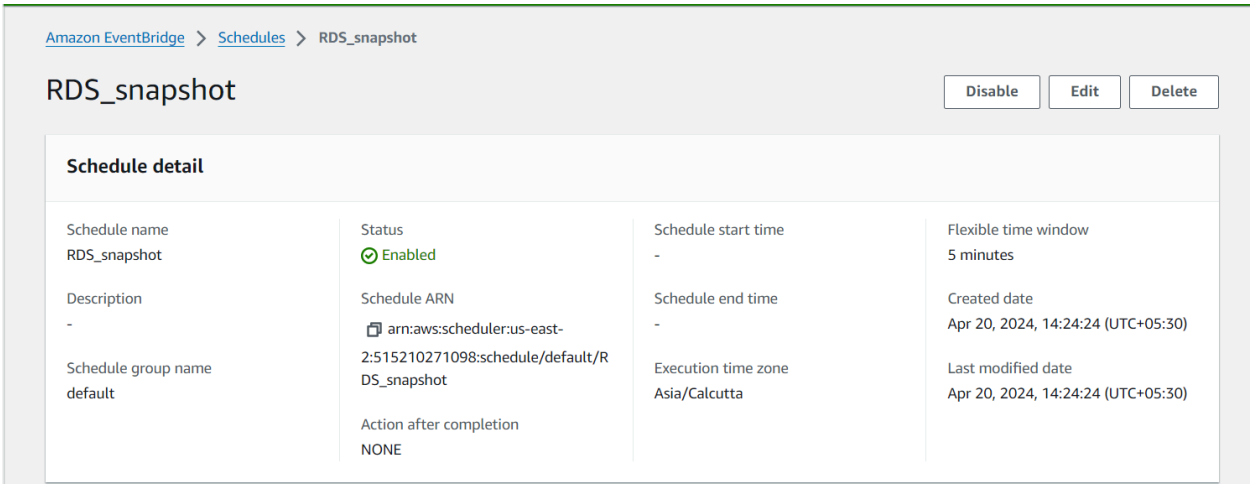
- Write the Boto3 Python script to:

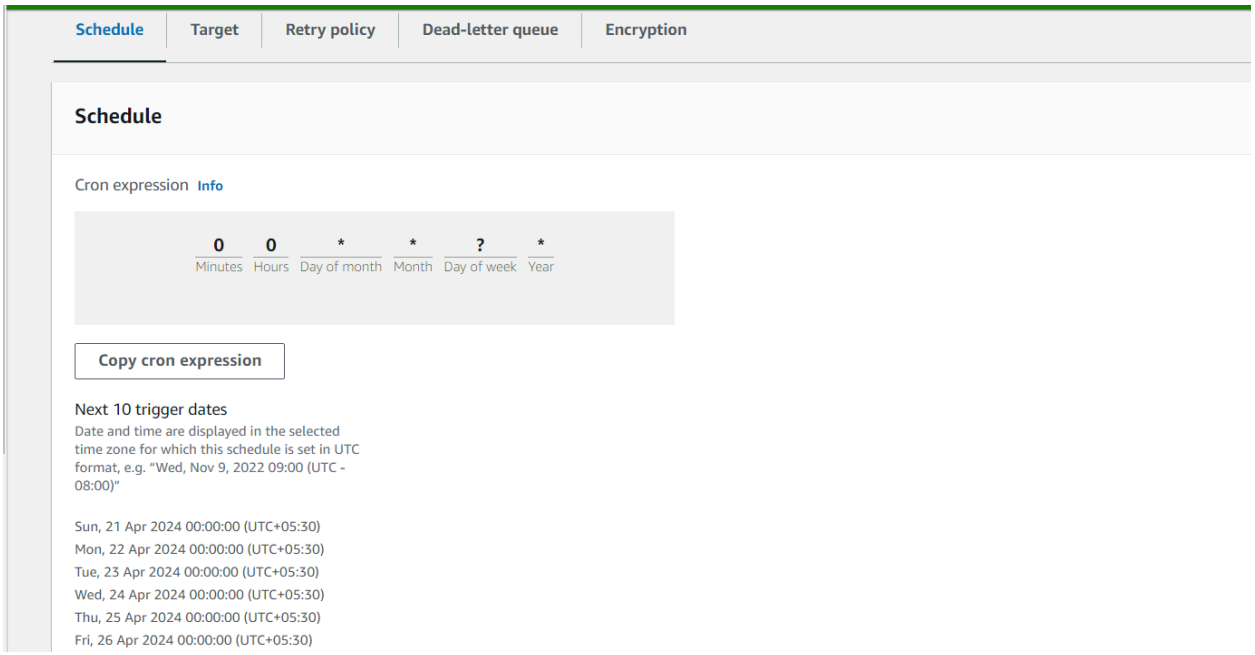
1. Initialize a boto3 RDS client.
2. Take a snapshot of the specified RDS instance.
3. Print the snapshot ID for logging purposes.

```python
import boto3

def take_rds_snapshot(instance_identifier):
    # Initialize Boto3 RDS client
    rds_client = boto3.client('rds')

    # Take a snapshot of the specified RDS instance
    response = rds_client.create_db_snapshot(
        DBSnapshotIdentifier='snapshot-' + instance_identifier.replace('.', '-'),  # Example
        DBInstanceIdentifier=instance_identifier
    )

    # Print the snapshot ID for logging purposes
    print("Snapshot ID:", response['DBSnapshot']['DBSnapshotIdentifier'])

# Example usage: Specify your RDS instance identifier
instance_identifier = 'database-1-instance-1.czyfftv2dgcu.us-east-2.rds.amazonaws.com'
take_rds_snapshot(instance_identifier)
```

4. Event Source (Bonus):

- Attach an event source, like Amazon CloudWatch Events, to trigger the Lambda function every day (or as per your preferred frequency).





5. Manual Invocation:

- After saving your function, manually trigger it (or wait for the scheduled trigger).
- Go to the RDS dashboard and confirm that a snapshot has been taken.

Assignment 4:

Monitor Unencrypted S3 Buckets Using AWS Lambda and Boto3.

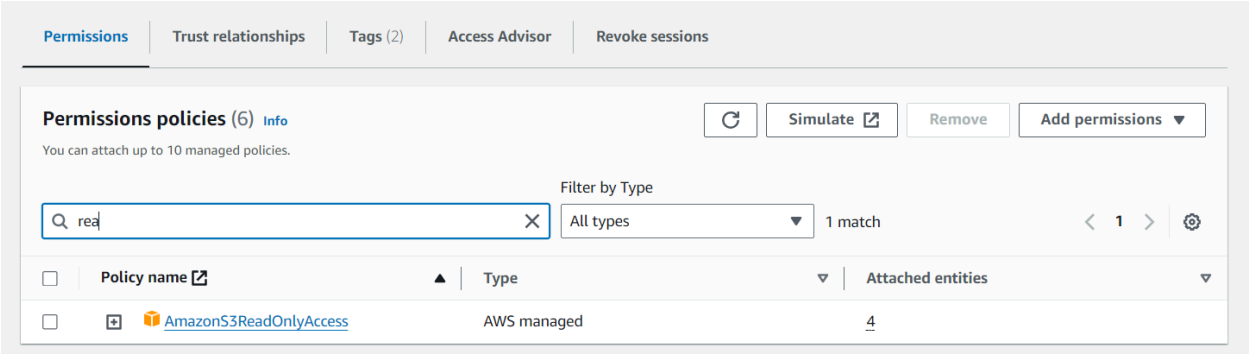
To enhance your AWS security posture by setting up a Lambda function that detects any S3 bucket without server-side encryption.

1. S3 Setup:

- Navigate to the S3 dashboard and create a few buckets. Ensure that a couple of them don't have server-side encryption enabled.

2. Lambda IAM Role:

- In the IAM dashboard, create a new role for Lambda.
- Attach the `AmazonS3ReadOnlyAccess` policy to this role.



3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.
- Choose Python 3.x as the runtime.
- Assign the IAM role created in the previous step.

- Write the Boto3 Python script to:

1. Initialize a boto3 S3 client.
2. List all S3 buckets.
3. Detects buckets without server-side encryption.
4. Print the names of unencrypted buckets for logging purposes.

```python
import boto3

def detect_unencrypted_buckets():
    # Initialize Boto3 S3 client
    s3_client = boto3.client('s3')

    try:
        # List all S3 buckets
        response = s3_client.list_buckets()

        # Detect buckets without server-side encryption
        unencrypted_buckets = []
        for bucket in response['Buckets']:
            bucket_name = bucket['Name']
            bucket_encryption = s3_client.get_bucket_encryption(Bucket=bucket_name)
            if 'ServerSideEncryptionConfiguration' not in bucket_encryption:
                unencrypted_buckets.append(bucket_name)

        # Print the names of unencrypted buckets for logging purposes
        if unencrypted_buckets:
            print("Unencrypted buckets:")
            for bucket_name in unencrypted_buckets:
                print(bucket_name)
        else:
            print("No unencrypted buckets found.")

    except Exception as e:
        print("Error:", str(e))

# Invoke the function to detect unencrypted buckets
detect_unencrypted_buckets()
```

4. Manual Invocation:

- After saving your function, manually trigger it.
- Review the Lambda logs to identify the buckets without server-side encryption.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Shrutika\Desktop\boto3> & C:/Users/Shrutika/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Shrutika/Desktop/boto3/test_boto.py
No unencrypted buckets found.
PS C:\Users\Shrutika\Desktop\boto3>
```

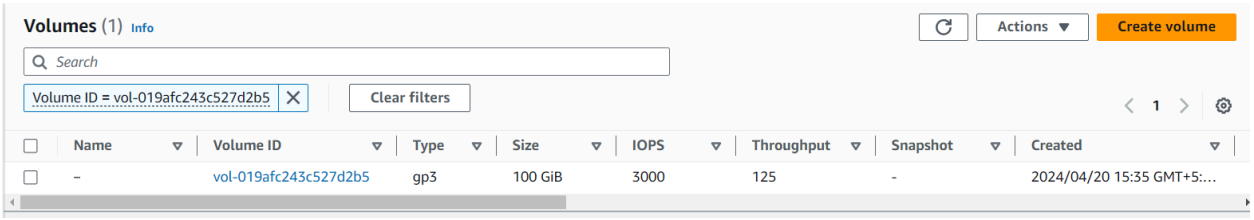
Assignment 5:

Automatic EBS Snapshot and Cleanup Using AWS Lambda and Boto3

To automate the backup process for your EBS volumes and ensure that backups older than a specified retention period are cleaned up to save costs.

1. EBS Setup:

- Navigate to the EC2 dashboard and identify or create an EBS volume you wish to back up.
- Note down the volume ID.



2. Lambda IAM Role:

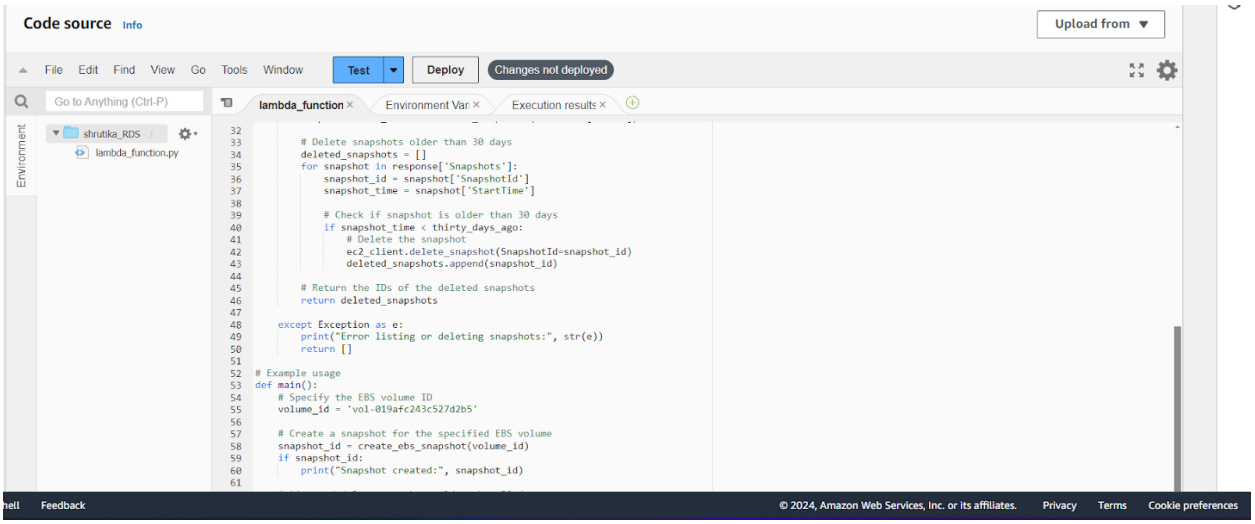
- In the IAM dashboard, create a new role for Lambda.
- Attach policies that allow Lambda to create EBS snapshots and delete them (`AmazonEC2FullAccess` for simplicity, but be more restrictive in real-world scenarios).

3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.
- Choose Python 3.x as the runtime.
- Assign the IAM role created in the previous step.

- Write the Boto3 Python script to:

1. Initialize a boto3 EC2 client.
2. Create a snapshot for the specified EBS volume.
3. List snapshots and delete those older than 30 days.
4. Print the IDs of the created and deleted snapshots for logging purposes.



Python Script :

```python
import boto3
from datetime import datetime, timedelta

def create_ebs_snapshot(volume_id):
    # Initialize Boto3 EC2 client
    ec2_client = boto3.client('ec2')

    try:
        # Create a snapshot for the specified EBS volume
        response = ec2_client.create_snapshot(
            VolumeId=volume_id,
            Description='Snapshot created by backup script'
        )

        # Return the snapshot ID
        return response['SnapshotId']

    except Exception as e:
        print("Error creating snapshot:", str(e))
        return None

def list_and_delete_old_snapshots():
    # Initialize Boto3 EC2 client
    ec2_client = boto3.client('ec2')

    try:
        # Calculate the date 30 days ago
        thirty_days_ago = datetime.now() - timedelta(days=30)

        # List all snapshots
        response = ec2_client.describe_snapshots(OwnerIds=['self'])
```

```python

        # Delete snapshots older than 30 days
        deleted_snapshots = []
        for snapshot in response['Snapshots']:
            snapshot_id = snapshot['SnapshotId']
            snapshot_time = snapshot['StartTime']

            # Check if snapshot is older than 30 days
            if snapshot_time < thirty_days_ago:
                # Delete the snapshot
                ec2_client.delete_snapshot(SnapshotId=snapshot_id)
                deleted_snapshots.append(snapshot_id)

        # Return the IDs of the deleted snapshots
        return deleted_snapshots

    except Exception as e:
        print("Error listing or deleting snapshots:", str(e))
        return []

# Example usage
def main():
    # Specify the EBS volume ID
    volume_id = 'vol-019afc243c527d2b5'

    # Create a snapshot for the specified EBS volume
    snapshot_id = create_ebs_snapshot(volume_id)
    if snapshot_id:
        print("Snapshot created:", snapshot_id)

    # List and delete snapshots older than 30 days
    deleted_snapshots = list_and_delete_old_snapshots()
    if deleted_snapshots:
        print("Snapshots deleted:", deleted_snapshots)

if __name__ == "__main__":
    main()
```

4. Event Source (Bonus):

- Attach an event source, like Amazon CloudWatch Events, to trigger the Lambda function at your desired backup frequency (e.g., every week).

5. Manual Invocation:

- After saving your function, either manually trigger it or wait for the scheduled event.
- Go to the EC2 dashboard and confirm that the snapshot is created and old snapshots are deleted.

```
PS C:\Users\Shrutika\Desktop\boto3> & C:/Users/Shrutika/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Shrutika/Desktop/boto3/test_boto.py
Snapshot created: snap-0e2a5d5462d0010bb
Error listing or deleting snapshots: can't compare offset-naive and offset-aware datetimes
PS C:\Users\Shrutika\Desktop\boto3> & C:/Users/Shrutika/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Shrutika/Desktop/boto3/test_boto.py
```

Assignment 6:
Auto-Tagging EC2 Instances on Launch Using AWS Lambda and Boto3.
Learn to automate the tagging of EC2 instances as soon as they are launched, ensuring better resource tracking and management.

1. EC2 Setup:

- Ensure you have the capability to launch EC2 instances.

2. Lambda IAM Role:

- In the IAM dashboard, create a new role for Lambda.
- Attach the `AmazonEC2FullAccess` policy to this role.

3. Lambda Function:

- Navigate to the Lambda dashboard and create a new function.
- Choose Python 3.x as the runtime.
- Assign the IAM role created in the previous step.

- Write the Boto3 Python script to:

1. Initialize a boto3 EC2 client.
2. Retrieve the instance ID from the event.
3. Tag the new instance with the current date and another tag of your choice.
4. Print a confirmation message for logging purposes.

```python
import boto3
import datetime

def tag_instance(event):
    # Initialize a boto3 EC2 client
    ec2_client = boto3.client('ec2')

    # Retrieve the instance ID from the event
    instance_id = event['instance_id']

    # Tag the new instance with the current date and another tag
    current_date = datetime.datetime.now().strftime('%Y-%m-%d')
    tag_key = 'DateCreated'
    tag_value = current_date
    another_tag_key = 'Test_Tag'
    another_tag_value = 'Test_value'
    tags = [
        {'Key': tag_key, 'Value': tag_value},
        {'Key': another_tag_key, 'Value': another_tag_value}
    ]
    ec2_client.create_tags(Resources=[instance_id], Tags=tags)

    # Print a confirmation message for logging purposes
    print(f"Instance {instance_id} tagged with {tag_key}: {tag_value} and {another_tag_key}: {another_tag_value}")

# Example event containing the instance ID for manual testing
event = {'instance_id': 'i-074cbdef4dbe31b1a'}

# Call the function with the event
tag_instance(event)
```

4. CloudWatch Events:

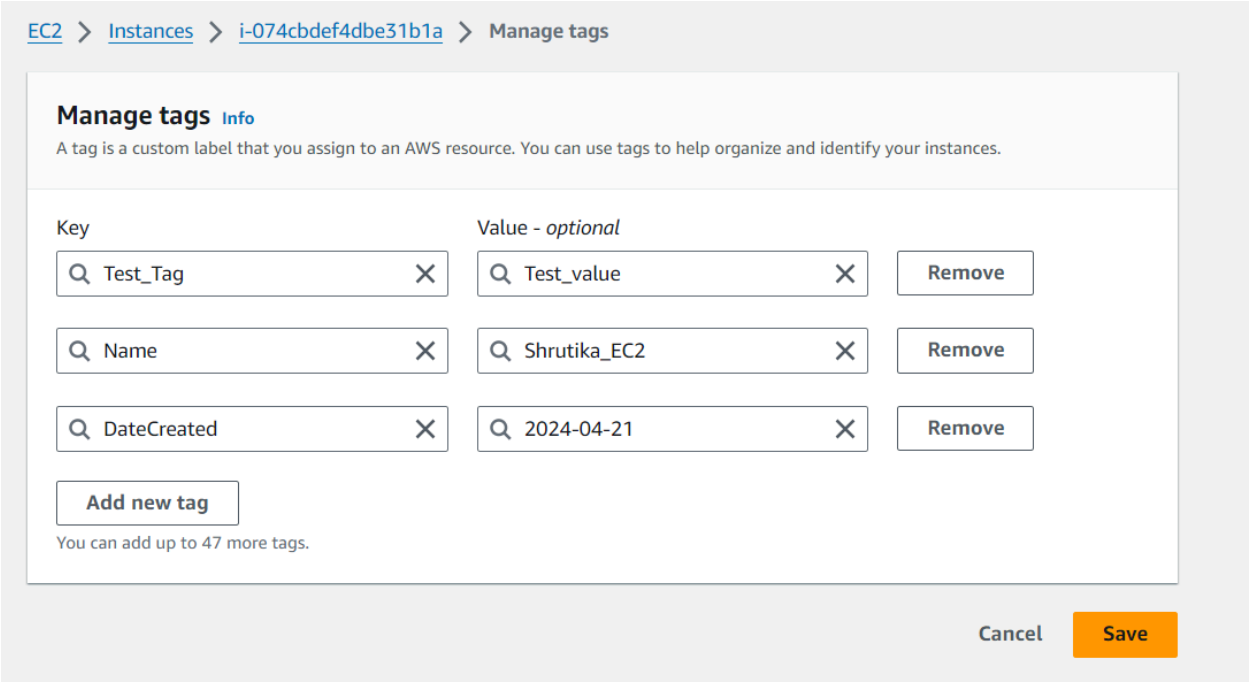
- Set up a CloudWatch Event Rule to trigger the EC2 instance launch event.
- Attach the Lambda function as the target.

5. Testing:

- Launch a new EC2 instance.
- After a short delay, confirm that the instance is automatically tagged as specified.

lambda_function × | Environment Var × | Execution results ×

```python
import boto3
import datetime

def tag_instance(event):
    # Initialize a boto3 EC2 client
    ec2_client = boto3.client('ec2')

    # Retrieve the instance ID from the event
    instance_id = event['instance_id']

    # Tag the new instance with the current date and another tag
    current_date = datetime.datetime.now().strftime('%Y-%m-%d')
    tag_key = 'DateCreated'
    tag_value = current_date
    another_tag_key = 'Assignment'
    another_tag_value = 'Test_EC2'
    tags = [
        {'Key': tag_key, 'Value': tag_value},
        {'Key': another_tag_key, 'Value': another_tag_value}
    ]
    ec2_client.create_tags(Resources=[instance_id], Tags=tags)

    # Print a confirmation message for logging purposes
    print(f"Instance {instance_id} tagged with {tag_key}: {tag_value} and {another_tag_key}: {another_tag_value}")

# Example event containing the instance ID for manual testing
event = {'instance_id': 'i-0eb63e35e0e21c24d'}

# Call the function with the event
tag_instance(event)
```



Manual Testing :

```
PS C:\Users\Shrutika\Desktop\boto3> python .\test_boto.py
Instance i-074cbdef4dbe31b1a tagged with DateCreated: 2024-04-21 and Test_Tag: Test_value
PS C:\Users\Shrutika\Desktop\boto3>
```

Assignment 9:
Analyze Sentiment of User Reviews Using AWS Lambda, Boto3, and Amazon Comprehend. Automatically analyze and categorize the sentiment of user reviews using Amazon Comprehend.

Task: Set up a Lambda function to receive user reviews, analyze their sentiment using Amazon Comprehend, and log the results.

Instructions:

1. Lambda IAM Role:

   - In the IAM dashboard, create a new role for Lambda.
   - Attach policies that allow Lambda to use Amazon Comprehend.

2. Lambda Function:

   - Navigate to the Lambda dashboard and create a new function.
   - Choose Python 3.x as the runtime.
   - Assign the IAM role created previously.

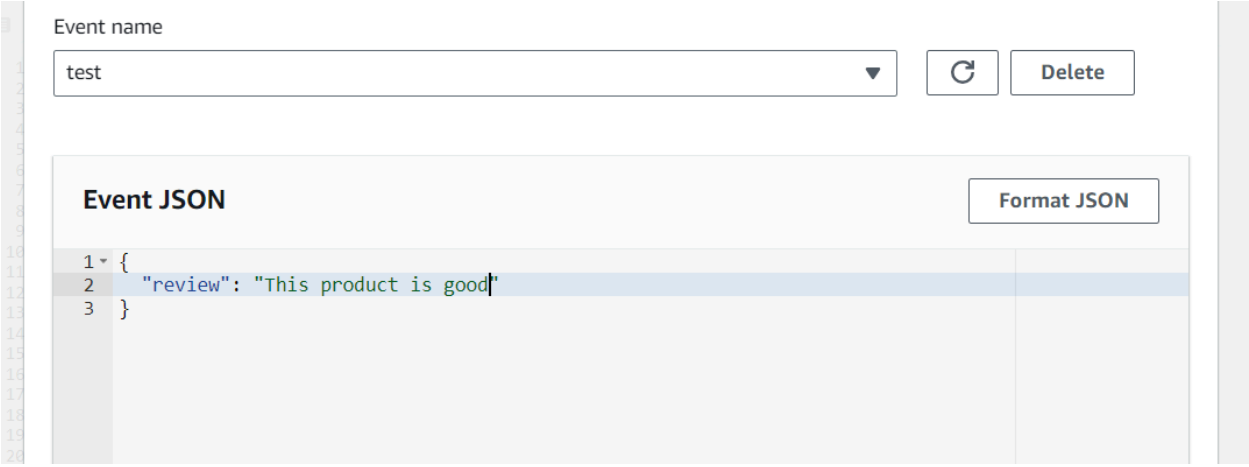
   - Write the Boto3 Python script to:

     1. Extract the user review from an event.
     2. Use Amazon Comprehend to analyze the sentiment of the review.
     3. Log the sentiment result.

```python
import boto3

# Initialize Boto3 clients
comprehend = boto3.client('comprehend')
cloudwatch = boto3.client('cloudwatch')

def extract_review(event):
    # Assuming event contains the review text
    review = event['review']
    return review

def analyze_sentiment(review):
    # Use Amazon Comprehend to analyze sentiment
    response = comprehend.detect_sentiment(Text=review, LanguageCode='en')
    sentiment = response['Sentiment']
    return sentiment

def log_sentiment(sentiment):
    # Log sentiment result to CloudWatch
    response = cloudwatch.put_metric_data(
        Namespace='SentimentAnalysis',
        MetricData=[
            {
                'MetricName': 'Sentiment',
                'Value': 1,  # You can assign a numeric value to each sentiment if needed
                'Unit': 'Count',
                'Dimensions': [
                    {
                        'Name': 'SentimentType',
                        'Value': sentiment
                    }
                ]
            }
        ]
    )

def main(event):
    review = extract_review(event)
    sentiment = analyze_sentiment(review)
    log_sentiment(sentiment)

# Example usage
if __name__ == "__main__":
    event = {'review': 'This product is amazing!'}
    main(event)
```

3. Testing:

- Manually trigger the Lambda function with sample reviews.



- Confirm the sentiment analysis results in the Lambda logs.

Assignment 11:
Notify When ELB 5xx Errors Spike Using AWS Lambda, Boto3, and SNS
To automatically receive notifications when your Elastic Load Balancer (ELB) encounters an unusually high number of 5xx errors.
Set up a Lambda function that checks the ELB's 5xx error metrics and sends an SNS notification if errors exceed a certain threshold.

Instructions:

1. SNS Setup:

- In the SNS dashboard, create a new topic.
- Subscribe your email or phone to this topic.

2. Lambda IAM Role:

- Create a new role for Lambda with permissions to read CloudWatch metrics and send SNS notifications.

3. Lambda Function:

- Create a function and assign the above IAM role.

- Use Boto3 to:

1. Fetch the 5xx error count metric from CloudWatch for your ELB.
2. If the count exceeds a threshold (e.g., 10 in the last 5 minutes), trigger an SNS notification.

```python
import boto3
import datetime

def fetch_cloudwatch_metric(elb_name, duration_minutes):
    # Create CloudWatch client
    cloudwatch = boto3.client('cloudwatch')

    # Define time range for the metric query
    end_time = datetime.datetime.now()
    start_time = end_time - datetime.timedelta(minutes=duration_minutes)

    # Fetch the metric data
    response = cloudwatch.get_metric_statistics(
        Namespace='AWS/ELB',
        MetricName='HTTPCode_Backend_5XX',
        Dimensions=[
            {
                'Name': 'LoadBalancerName',
                'Value': elb_name
            },
        ],
        StartTime=start_time,
        EndTime=end_time,
        Period=60,  # 1-minute granularity
        Statistics=['Sum']
    )

    # Calculate the total count of 5xx errors
    error_count = 0
    for datapoint in response['Datapoints']:
        error_count += datapoint['Sum']

    return error_count
```

```python
def trigger_sns_notification(topic_arn, count_threshold, duration_minutes):
    # Fetch the 5xx error count metric for your ELB
    elb_name = 'LBassignmentSP'
    error_count = fetch_cloudwatch_metric(elb_name, duration_minutes)

    # If the count exceeds the threshold, trigger an SNS notification
    if error_count > count_threshold:
        sns = boto3.client('sns')
        message = f"5xx error count for ELB '{elb_name}' exceeded threshold ({count_threshold}) in the last {duration_minutes} minutes. Count: {error_count}"
        sns.publish(
            TopicArn=topic_arn,
            Message=message,
            Subject=f"5xx Error Count Exceeded Threshold for ELB '{elb_name}'"
        )

def main():
    # Specify your SNS topic ARN and threshold values
    topic_arn = 'arn:aws:sns:us-east-2:515210271098:EC2_State_Change_Notifications'
    count_threshold = 10
    duration_minutes = 5

    # Trigger SNS notification if error count exceeds threshold
    trigger_sns_notification(topic_arn, count_threshold, duration_minutes)

if __name__ == "__main__":
    main()
```

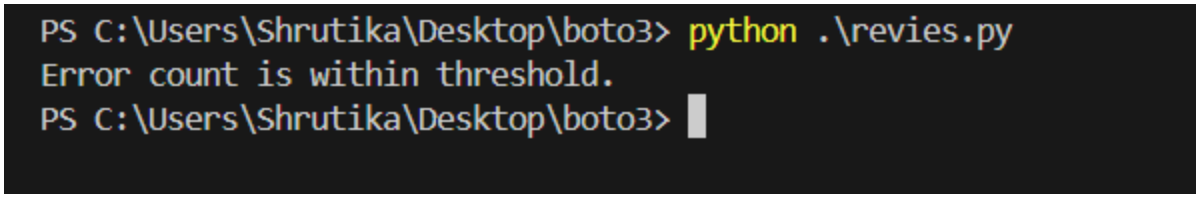
4. CloudWatch Events:

   - Schedule your Lambda function to run every 5 minutes.

5. Testing:

   - Simulate or wait for a spike in 5xx errors.

Manual on local :



- Confirm receipt of the SNS notification.

Assignment 14:
Audit S3 Bucket Permissions and Notify for Public Buckets
Automatically audit S3 bucket permissions and send notifications if any buckets have public read or write permissions.
Set up a Lambda function to regularly audit S3 bucket permissions and send SNS notifications for any buckets that are publicly accessible.

Instructions:

1. SNS Setup:

- Create a new SNS topic.
- Subscribe to the topic with your email.

2. Lambda IAM Role:

- Create a role with permissions to list S3 bucket permissions and send SNS notifications.

3. Lambda Function:

- Create a function with the above IAM role.

- Use Boto3 to:

1. List all S3 buckets.
2. For each bucket, check its permission settings.
3. If a bucket has public read or write permissions, send a notification with its name via SNS.

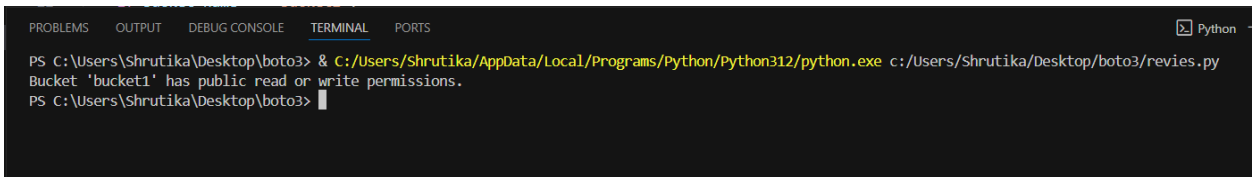
```python
import boto3
from botocore.exceptions import ClientError

def list_s3_buckets():
    # Mocked list of bucket names for testing
    return ['bucket1', 'bucket2', 'bucket3']

def check_bucket_permissions(bucket_name):
    # Simulate checking permissions for each bucket
    # For testing purposes, assume 'bucket1' has public write permission
    if bucket_name == 'bucket1':
        return True
    return False

def send_notification(bucket_name):
    # Print notification message for testing
    print(f"Bucket '{bucket_name}' has public read or write permissions.")

def main():
    # List all S3 buckets
    bucket_names = list_s3_buckets()

    # Check permissions for each bucket
    for bucket_name in bucket_names:
        if check_bucket_permissions(bucket_name):
            # If public access is detected, print notification
            send_notification(bucket_name)

if __name__ == "__main__":
    main()

```

4. CloudWatch Events:

- Schedule your Lambda function to run daily.

5. Testing:

- Make one or two of your S3 buckets public.
- Run the Lambda function and ensure you receive appropriate SNS notifications.

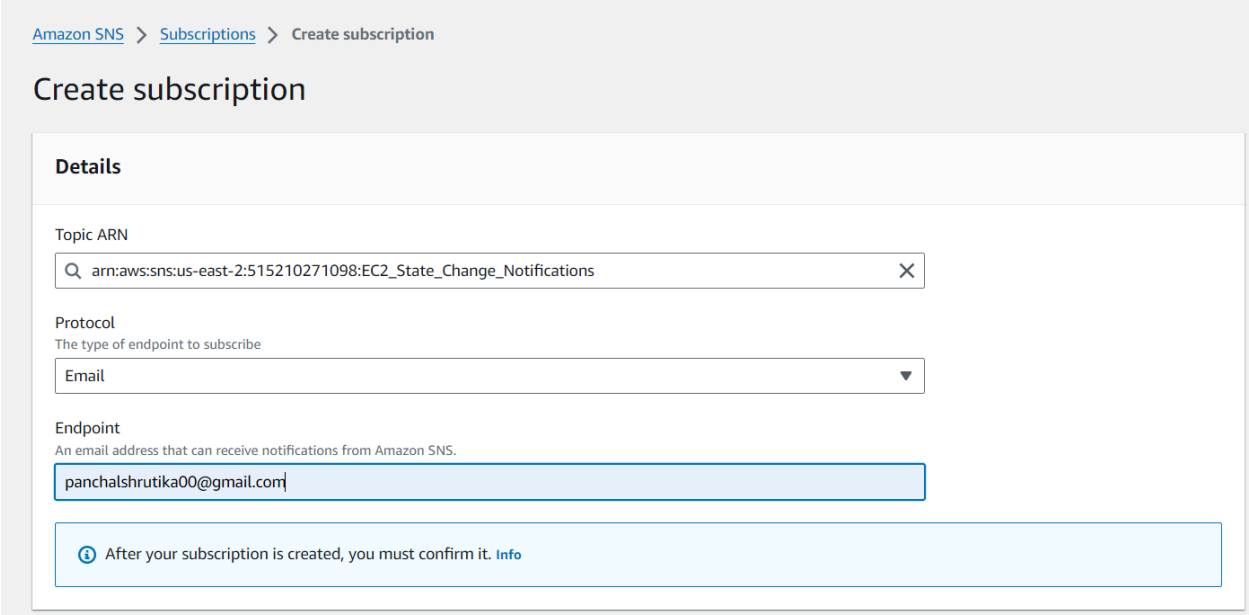


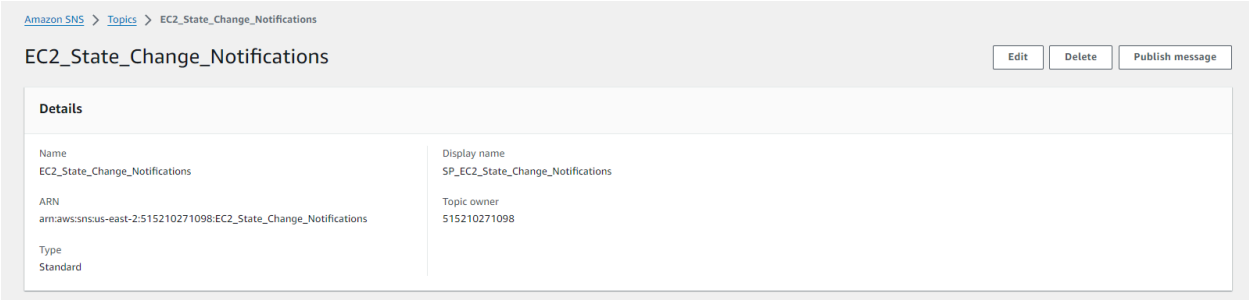
Assignment 15:
Monitor EC2 Instance State Changes Using AWS Lambda, Boto3, and SNS
Automatically monitor changes in EC2 instance states and send notifications whenever an instance is started or stopped.
Set up a Lambda function that listens to EC2 state change events and sends SNS notifications detailing the state changes.

Instructions:

1. SNS Setup:

- Navigate to the SNS dashboard and create a new topic.
- Subscribe to this topic with your email.





2. Lambda IAM Role:

- Create a role with permissions to read EC2 instance states and send SNS notifications.

3. Lambda Function:

- Create a function and assign the above IAM role.

- Use Boto3 to:

1. Extract details from the event regarding the EC2 state change.

2. Send an SNS notification with details about which EC2 instance changed state and the new state (e.g., started, stopped).

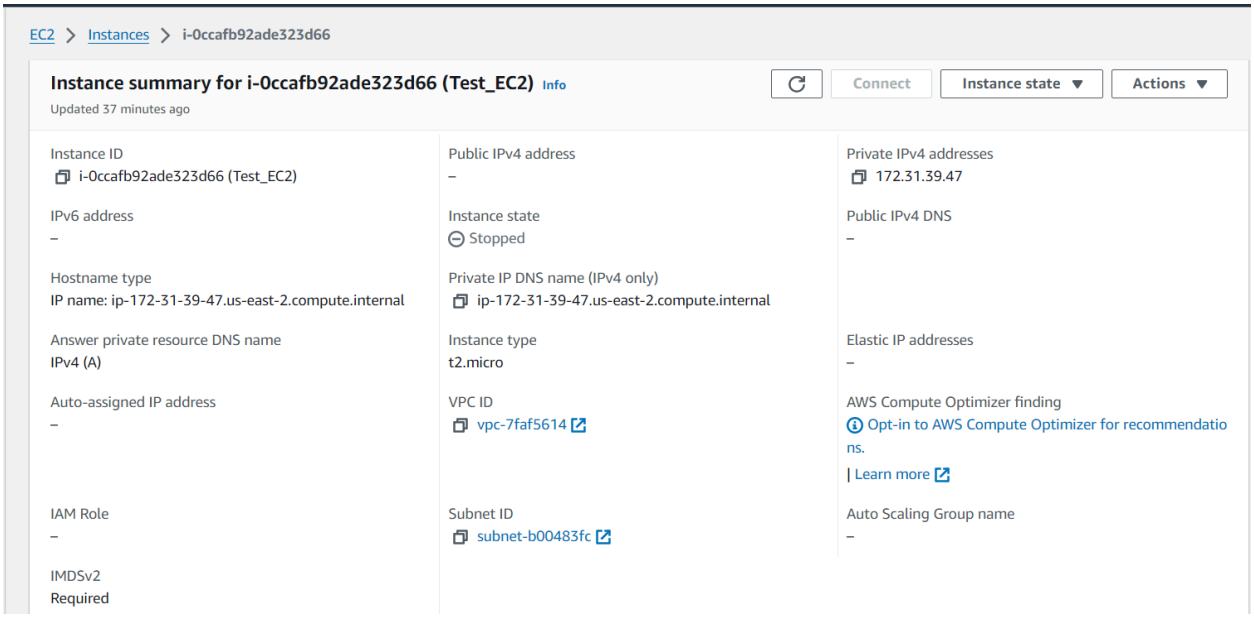
```python
import boto3

def extract_ec2_state_change_details(event):
    # Extract relevant details from the event regarding the EC2 state change
    instance_id = event['detail']['instance-id']
    state = event['detail']['state']

    return instance_id, state

def send_sns_notification(topic_arn, instance_id, state):
    # Create SNS client
    sns = boto3.client('sns')

    # Compose message for the SNS notification
    message = f"The state of EC2 instance {instance_id} has changed to {state}."

    # Send SNS notification
    sns.publish(
        TopicArn=topic_arn,
        Message=message,
        Subject=f"EC2 Instance State Change: {instance_id}"
    )

def main(event):
    # Specify your SNS topic ARN
    topic_arn = 'arn:aws:sns:us-east-2:515210271098:EC2_State_Change_Notifications'

    # Extract details from the event regarding the EC2 state change
    instance_id, state = extract_ec2_state_change_details(event)

    # Send an SNS notification with details about the EC2 instance state change
    send_sns_notification(topic_arn, instance_id, state)
```

4. EC2 Event Bridge (formerly CloudWatch Events):

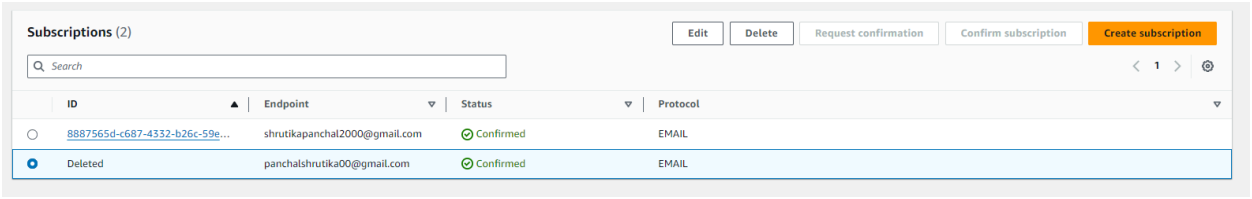
- Set up an Event Bridge rule to trigger your Lambda function whenever an EC2 instance state changes.

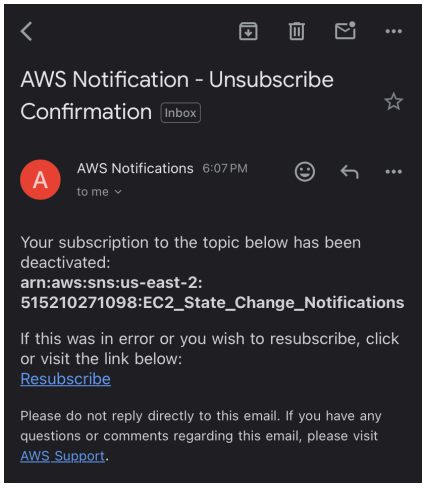
5. Testing:

- Start or stop one of your EC2 instances.



- Confirm you receive an SNS notification about the state change.





Assignment 20: Load Balancer Health Checker
Design a Lambda function that checks the health of registered instances behind an Elastic Load Balancer (ELB) and notifies via SNS if any instances are unhealthy.

1. Create a Lambda function.

2. With Boto3, configure the function to:

- Check the health of registered instances behind a given ELB.
- If any instances are found to be unhealthy, publish a detailed message to an SNS topic.

```python
import boto3

def check_instance_health(elb_name):
    # Create ELB client
    elb = boto3.client('elb')

    # Describe instance health for the ELB
    response = elb.describe_instance_health(
        LoadBalancerName=elb_name
    )

    return response['InstanceStates']

def publish_sns_message(topic_arn, message):
    # Create SNS client
    sns = boto3.client('sns')

    # Publish message to SNS topic
    sns.publish(
        TopicArn=topic_arn,
        Message=message,
        Subject="Unhealthy Instances Detected"
    )

def main(elb_name, sns_topic_arn):
    # Check the health of registered instances behind the ELB
    instance_states = check_instance_health(elb_name)

    # Check if any instances are unhealthy
    unhealthy_instances = [instance for instance in instance_states if instance['State'] != 'InService']
```

```python
    # If unhealthy instances are found, publish a detailed message to the SNS topic
    if unhealthy_instances:
        message = f"Unhealthy instances detected behind ELB '{elb_name}':\n"
        for instance in unhealthy_instances:
            message += f"Instance ID: {instance['InstanceId']}, State: {instance['State']}\n"
        publish_sns_message(sns_topic_arn, message)
        print("Detailed message published to SNS topic.")
    else:
        print("All instances are healthy.")

if __name__ == "__main__":
    # Specify your ELB name and SNS topic ARN
    elb_name = 'Nikitasloadbalancer'
    sns_topic_arn = 'arn:aws:sns:us-east-2:515210271098:EC2_State_Change_Notifications'

    main(elb_name, sns_topic_arn)
```

3. Set up a CloudWatch event to trigger this Lambda function every 10 minutes.