**Description**

**Overview:**

**Develop a system that automatically manages the lifecycle of a web application hosted on EC2 instances, monitors its health, and reacts to changes in traffic by scaling resources.  Furthermore, administrators receive notifications regarding the infrastructure's health and scaling events.**

**Detailed Breakdown:**

**1. Web Application Deployment:**

 - Use `boto3` to:

 - Create an S3 bucket to store your web application's static files.

 - Launch an EC2 instance and configure it as a web server (e.g., Apache, Nginx).  - Deploy the web application onto the EC2 instance.

**2. Load Balancing with ELB:**

 - Deploy an Application Load Balancer (ALB) using `boto3`.

 - Register the EC2 instance(s) with the ALB.

**3. Auto Scaling Group (ASG) Configuration:**

- Using `boto3`, create an ASG with the deployed EC2 instance as a template.

 - Configure scaling policies to scale in/out based on metrics like CPU utilization or network traffic.

**4. SNS Notifications:**

 - Set up different SNS topics for different alerts (e.g., health issues, scaling events, high traffic).

 - Integrate SNS with Lambda so that administrators receive SMS or email notifications.

**5. Infrastructure Automation:**

 - Create a single script using `boto3` that:

 - Deploys the entire infrastructure.

 - Updates any component as required.

 - Tears down everything when the application is no longer needed.

**6. Optional Enhancement – Dynamic Content Handling:**

- Store user-generated content or uploads on S3.

 - When a user uploads content to the web application, it gets temporarily stored on the  EC2 instance. A background process (or another Lambda function) can move this to the S3  bucket and update the application's database to point to the content's new location on S3.

**Objectives:**

- Gain a comprehensive understanding of key AWS services and their integration. - Understand the lifecycle of a dynamic web application and its infrastructure.

- Learn how to automate infrastructure deployment and management tasks using boto3. - Experience with real-time monitoring and alerting systems

**Github Url:** [**https://github.com/Ram495-ctrl/Monitoring-Scaling-and-Automation.git**](https://github.com/Ram495-ctrl/Monitoring-Scaling-and-Automation.git)

**Create boto3 code:**

from optparse import Values

import boto3

import time

import json

import yaml

AWS\_REGION = 'us-west-2'

# Create a session using 'default' profile

session = boto3.Session(profile\_name="default", region\_name=AWS\_REGION)

# Initialize all clients

s3 = session.client('s3')

ec2 = session.client('ec2')

elb = session.client('elbv2')

autoscaling = session.client('autoscaling')

sns = session.client('sns')

# Step 1: Providing all values by using yaml file

def read\_values\_from\_file(file\_path):

    with open(file\_path, 'r') as file:

        values = yaml.safe\_load(file)

    return values

# Step 2: Create S3 Bucket before checking if it exists

def create\_s3\_bucket(bucket\_name):

    try:

        # Check if the bucket exists

        response = s3.head\_bucket(Bucket=bucket\_name)

        print(f"S3 bucket already exists: {bucket\_name}")

        return bucket\_name

    except s3.exceptions.ClientError as e:

        error\_code = e.response['Error']['Code']

        if error\_code == '403':

            print(f"Access denied to check bucket {bucket\_name}. Ensure you have correct permissions.")

            return None

        elif error\_code == '404':

            # Bucket doesn't exist, proceed to create it

            try:

                response = s3.create\_bucket(

                    Bucket=bucket\_name,

                    CreateBucketConfiguration={

                        'LocationConstraint': session.region\_name

                    }

                )

                print(f"S3 bucket created: {bucket\_name}")

                return bucket\_name

            except s3.exceptions.BucketAlreadyExists:

                print(f"Bucket name '{bucket\_name}' already exists globally. Please choose a different name.")

                return None

            except Exception as e:

                print(f"Error creating bucket: {e}")

                return None

        else:

            print(f"Error checking bucket: {e}")

            return None

# Step 3: Create Security Group or use existing

def create\_security\_group(description, security\_group\_name):

    try:

        # Check if the security group already exists

        response = ec2.describe\_security\_groups(

            Filters=[{'Name': 'group-name', 'Values': [security\_group\_name]}]

        )

        if response['SecurityGroups']:

            security\_group\_id = response['SecurityGroups'][0]['GroupId']

            print(f"Security group already exists: {security\_group\_id}")

            return security\_group\_id

        else:

            # If the security group doesn't exist, create a new one

            response = ec2.create\_security\_group(

                GroupName=security\_group\_name,

                Description=description

            )

            security\_group\_id = response['GroupId']

            print(f"Security group created: {security\_group\_id}")

            ec2.create\_tags(

                Resources=[security\_group\_id],

                Tags=[{'Key': 'Name', 'Value': security\_group\_name}]

            )

            print(f"Tags added to security group: {security\_group\_name}")

            # Set inbound rules

            ec2.authorize\_security\_group\_ingress(

                GroupId=security\_group\_id,

                IpPermissions=[

                    {

                        'IpProtocol': 'tcp',

                        'FromPort': 80,

                        'ToPort': 80,

                        'IpRanges': [{'CidrIp': '0.0.0.0/0'}]

                    },

                    {

                        'IpProtocol': 'tcp',

                        'FromPort': 22,

                        'ToPort': 22,

                        'IpRanges': [{'CidrIp': '0.0.0.0/0'}]

                    }

                ]

            )

            print("Inbound traffic rules applied to security group")

            return security\_group\_id

    except Exception as e:

        print(f"Error creating or checking security group: {e}")

# Step 4: Launch an EC2 instance or use existing

def launch\_ec2\_instance(security\_group\_id, ami\_id, instance\_type, ec2\_name):

    try:

        # Check if an instance with the tag 'WebServer' already exists

        response = ec2.describe\_instances(

            Filters=[

                {'Name': 'tag:Name', 'Values': [ec2\_name]},

                {'Name': 'instance-state-name', 'Values': ['running', 'pending']}

            ]

        )

        if response['Reservations']:

            instance\_id = response['Reservations'][0]['Instances'][0]['InstanceId']

            print(f"EC2 instance already exists: {instance\_id}")

            return instance\_id

        else:

            # Launch new instance if not found

            response = ec2.run\_instances(

                ImageId=ami\_id,

                InstanceType=instance\_type,

                MinCount=1,

                MaxCount=1,

                UserData='''

                    #!/bin/bash

                    sudo apt update

                    sudo apt install -y nginx

                    sudo systemctl start nginx

                    sudo systemctl enable nginx

                    echo "<html><body><h1>Welcome to My Web App</h1></body></html>" > /var/www/html/index.html

                ''',

                TagSpecifications=[{

                    'ResourceType': 'instance',

                    'Tags': [{'Key': 'Name', 'Value': ec2\_name}]

                }],

                SecurityGroupIds=[security\_group\_id]

            )

            instance\_id = response['Instances'][0]['InstanceId']

            print(f"EC2 Instance launched: {instance\_id}")

            return instance\_id

    except Exception as e:

        print(f"Error launching EC2 instance: {e}")

# Step 5: Create Load Balancer and Target Group or use existing

def create\_load\_balancer(instance\_id, security\_group\_id, subnet\_id, vpc\_id, loadbalancer\_name, lb\_targetgroup):

    try:

        # Check if the ALB exists

        response = elb.describe\_load\_balancers(

            Names=[loadbalancer\_name]

        )

        load\_balancer\_arn = response['LoadBalancers'][0]['LoadBalancerArn']

        print(f"Load Balancer already exists: {load\_balancer\_arn}")

        # Check if the target group exists

        tg\_response = elb.describe\_target\_groups(

            Names=[lb\_targetgroup]

        )

        target\_group\_arn = tg\_response['TargetGroups'][0]['TargetGroupArn']

        print(f"Target Group already exists: {target\_group\_arn}")

    except elb.exceptions.LoadBalancerNotFoundException:

        # If load balancer doesn't exist, create a new one

        response = elb.create\_load\_balancer(

            Name=loadbalancer\_name,

            Subnets=subnet\_id,

            SecurityGroups=[security\_group\_id],

            Scheme='internet-facing',

            Type='application',

            Tags=[{'Key': 'Name', 'Value': loadbalancer\_name}]

        )

        load\_balancer\_arn = response['LoadBalancers'][0]['LoadBalancerArn']

        print(f"Load Balancer created: {load\_balancer\_arn}")

        # Create target group

        tg\_response = elb.create\_target\_group(

            Name=lb\_targetgroup,

            Protocol='HTTP',

            Port=80,

            VpcId=vpc\_id,

            TargetType='instance',

            HealthCheckProtocol='HTTP',

            HealthCheckPort='80',

            HealthCheckPath='/',

            Matcher={'HttpCode': '200'}

        )

        target\_group\_arn = tg\_response['TargetGroups'][0]['TargetGroupArn']

        print(f"Target Group created: {target\_group\_arn}")

    return load\_balancer\_arn, target\_group\_arn

# Step 6: Register EC2 Instances with Target Group

def register\_instances\_with\_target\_group(target\_group\_arn, instance\_id):

    try:

        response = elb.register\_targets(

            TargetGroupArn=target\_group\_arn,

            Targets=[

                {

                    'Id': instance\_id,

                    'Port': 80

                }

            ]

        )

        print(f"EC2 instance {instance\_id} registered with target group {target\_group\_arn}")

    except Exception as e:

        print(f"Error registering instance with target group: {e}")

# Step 7: Create Listener for ALB

def create\_listener(load\_balancer\_arn, target\_group\_arn):

    try:

        response = elb.create\_listener(

            LoadBalancerArn=load\_balancer\_arn,

            Protocol='HTTP',

            Port=80,

            DefaultActions=[{

                'Type': 'forward',

                'TargetGroupArn': target\_group\_arn

            }]

        )

        listener\_arn = response['Listeners'][0]['ListenerArn']

        print(f"Listener created: {listener\_arn}")

        #  # Create HTTPS listener (port 443)

        # https\_response = elb.create\_listener(

        #     LoadBalancerArn=load\_balancer\_arn,

        #     Protocol='HTTPS',

        #     Port=443,

        #     Certificates=[{

        #         'CertificateArn': certificate\_arn  # provided certificate ARN

        #     }],

        #     DefaultActions=[{

        #         'Type': 'forward',

        #         'TargetGroupArn': target\_group\_arn

        #     }]

        # )

        # https\_listener\_arn = https\_response['Listeners'][0]['ListenerArn']

        # print(f"HTTPS Listener created: {https\_listener\_arn}")

    except Exception as e:

        print(f"Error creating listener: {e}")

# Step 8: Create Launch-templete or use existing

def create\_launch\_template(instance\_id, launch\_template\_name):

    try:

        # Check if Launch Template already exists

        response = ec2.describe\_launch\_templates(

            LaunchTemplateNames=[launch\_template\_name]

        )

        if response['LaunchTemplates']:

            print(f"Launch Template {launch\_template\_name} already exists.")

            return response['LaunchTemplates'][0]['LaunchTemplateId']

    except ec2.exceptions.ClientError as e:

        error\_code = e.response['Error']['Code']

        if error\_code == 'InvalidLaunchTemplateName.NotFoundException':

            print(f"Launch Template {launch\_template\_name} not found. Proceeding to create it.")

        else:

            print(f"Error checking for launch template: {e}")

            return None

    try:

        # Fetch instance details

        response = ec2.describe\_instances(InstanceIds=[instance\_id])

        instance = response['Reservations'][0]['Instances'][0]

        # Create the Launch Template

        launch\_template\_data = {

            'ImageId': instance['ImageId'],

            'InstanceType': instance['InstanceType'],

            'SecurityGroupIds': [sg['GroupId'] for sg in instance['SecurityGroups']],

            'TagSpecifications': [{

                'ResourceType': 'instance',

                'Tags': instance['Tags']

            }]

        }

        # Only add KeyName if it's not None

        if instance.get('KeyName'):

            launch\_template\_data['KeyName'] = instance['KeyName']

        response = ec2.create\_launch\_template(

            LaunchTemplateName=launch\_template\_name,

            LaunchTemplateData=launch\_template\_data

        )

        launch\_template\_id = response['LaunchTemplate']['LaunchTemplateId']

        print(f"Launch Template created: {launch\_template\_id}")

        return launch\_template\_id

    except Exception as e:

        print(f"Error creating launch template: {e}")

        return None

# Step 9: Creating Auto Scaling Group

      # Check if Auto Scaling Group exists and create if it doesn't

def auto\_scaling\_group(launch\_template\_id, target\_group\_arn, subnet\_id, auto\_scaling\_group\_name, desired\_capacity, min\_size, max\_size ):

    try:

        # Check if the Auto Scaling Group already exists

        response = autoscaling.describe\_auto\_scaling\_groups(

            AutoScalingGroupNames=[auto\_scaling\_group\_name]

        )

        if response['AutoScalingGroups']:

            print(f"Auto Scaling Group {auto\_scaling\_group\_name} already exists.")

            return  # ASG exists, no need to create

    except Exception as e:

        print(f"Auto Scaling Group not found or error occurred: {e}")

    # Create an Auto Scaling Group if it doesn't exist

    try:

        autoscaling.create\_auto\_scaling\_group(

            AutoScalingGroupName=auto\_scaling\_group\_name,

            LaunchTemplate={

                'LaunchTemplateId': launch\_template\_id,

                'Version': '$Latest'

            },

            MinSize=min\_size,

            MaxSize=max\_size,

            DesiredCapacity=desired\_capacity,

            VPCZoneIdentifier=','.join(subnet\_id),

            TargetGroupARNs=[target\_group\_arn],

            HealthCheckType='EC2',

            HealthCheckGracePeriod=300,

            Tags=[{

                'Key': 'Name',

                'Value': auto\_scaling\_group\_name,

                'PropagateAtLaunch': True

            }]

        )

        print("Auto Scaling Group created successfully.")

    except Exception as e:

        print(f"Error creating Auto Scaling Group: {e}")

# Step 10: Creating scaling policies to scale in/out

def manage\_scaling\_policy(auto\_scaling\_group\_name):

    try:

        # Check if the target tracking policy already exists

        response = autoscaling.describe\_policies(

            AutoScalingGroupName=auto\_scaling\_group\_name,

            PolicyTypes=['TargetTrackingScaling']

        )

        existing\_policies = {policy['PolicyName'] for policy in response['ScalingPolicies']}

        # Create or update a single target tracking policy if it doesn't exist

        if 'cpu-target-tracking-policy' not in existing\_policies:

            target\_tracking\_policy = autoscaling.put\_scaling\_policy(

                AutoScalingGroupName=auto\_scaling\_group\_name,

                PolicyName='cpu-target-tracking-policy',

                PolicyType='TargetTrackingScaling',

                TargetTrackingConfiguration={

                    'PredefinedMetricSpecification': {

                        'PredefinedMetricType': 'ASGAverageCPUUtilization'

                    },

                    'TargetValue': 70.0,

                    'DisableScaleIn': False,

                }

            )

            print(f"Target Tracking Policy ARN: {target\_tracking\_policy['PolicyARN']}")

        else:

            print("Target tracking policy already exists.")

    except Exception as e:

        print(f"Error managing scaling policy: {e}")

# Step 11: Create SNS Topic for Alerts

def create\_sns\_topic(sns\_topic\_name):

    try:

        response = sns.create\_topic(Name=sns\_topic\_name)

        topic\_arn = response['TopicArn']

        print(f"SNS topic created: {topic\_arn}")

        return topic\_arn

    except Exception as e:

        print(f"Error creating SNS topic: {e}")

        return None

# Step 12: Subscribe to SNS Topic

def subscribe\_to\_topic(topic\_arn, protocol, endpoint):

    try:

        response = sns.subscribe(

            TopicArn=topic\_arn,

            Protocol=protocol,

            Endpoint=endpoint

        )

        subscription\_arn = response['SubscriptionArn']

        print(f"Subscription created: {subscription\_arn}")

    except Exception as e:

        print(f"Error subscribing to SNS topic: {e}")

# Step 13: Publish Message to SNS Topic

def publish\_sns\_message(topic\_arn, subject, message):

    try:

        response = sns.publish(

            TopicArn=topic\_arn,

            Subject=subject,

            Message=message

        )

        print(f"Message published to topic {topic\_arn}: {response['MessageId']}")

    except Exception as e:

        print(f"Error publishing message to SNS topic: {e}")

# Full Deployment Script

def deploy\_web\_application():

    file\_path = 'values.yaml'

    values = read\_values\_from\_file(file\_path)

    # Step 1: Create S3 Bucket

    bucket\_name = create\_s3\_bucket(values['bucket\_name'])

    # Step 2: Create Security Group

    security\_group\_id = create\_security\_group('Security group for Web App', values['security\_group\_name'])

    # Step 3: Launch EC2 Instance

    instance\_id = launch\_ec2\_instance(security\_group\_id, values['image\_id'], values['instance\_type'], values['ec2\_name'])

    waiter = ec2.get\_waiter('instance\_running')

    waiter.wait(InstanceIds=[instance\_id])

    # Step 4: Create Load Balancer and Target group

    load\_balancer\_arn, target\_group\_arn = create\_load\_balancer(instance\_id, security\_group\_id, values['subnet\_id'], values['vpc\_id'], values['loadbalancer\_name'], values['lb\_targetgroup'])

    # Step 5: Register EC2 Instances with Target Group

    register\_instances\_with\_target\_group(target\_group\_arn, instance\_id)

    # Step 6: Create Listener for Load Balancer

    create\_listener(load\_balancer\_arn, target\_group\_arn)

    # Step 7: Check/Create Launch Template

    launch\_template\_id = create\_launch\_template(instance\_id, values['launch\_template\_name'])

    # Step 8: Create Auto Scaling Group

    auto\_scaling\_group\_name = auto\_scaling\_group(launch\_template\_id, target\_group\_arn, values['subnet\_id'], values['auto\_scaling\_group\_name'], values['desired\_capacity'], values['min\_size'], values['max\_size'])

    # Step 9: Creating Scaling policy for ASG

    manage\_scaling\_policy(values['auto\_scaling\_group\_name'])

    # Step 10: Creating SNS topics for alerts

    health\_alert\_topic = create\_sns\_topic(values['sns\_topic\_name'])

    subscribe\_to\_topic(health\_alert\_topic, values['protocol'], values['email\_id'])

    publish\_sns\_message(health\_alert\_topic, values['subject'], values['message'])

    # Store resource IDs in a JSON file

    resources = {

        'bucket\_name': bucket\_name,

        'security\_group\_id': security\_group\_id,

        'instance\_id': instance\_id,

        'load\_balancer\_arn': load\_balancer\_arn,

        'target\_group\_arn': target\_group\_arn,

        'launch\_template\_id': launch\_template\_id,

        'auto\_scaling\_group\_name': values['auto\_scaling\_group\_name'],

        'sns\_topic': health\_alert\_topic

    }

    with open('resources.json', 'w') as f:

        json.dump(resources, f, indent=4)

# Run the deployment

deploy\_web\_application()

enter required in values.yaml

## ASG Values

auto\_scaling\_group\_name: Ram\_ASG

launch\_configuration\_name: my-lc1

launch\_template\_name: mylc

min\_size: 1

max\_size: 2

desired\_capacity: 1

image\_id: ami-05134c8ef96964280

instance\_type: t2.micro

subnet\_id: ['subnet-0f30c30418def6379', 'subnet-03ca36de9a927fe8e']

security\_group\_name: Ram\_WebAppSecurityGroup1

bucket\_name: ram1234345678

vpc\_id: vpc-0321f38a7b594180d

loadbalancer\_name: Ramwebapp-lb1

lb\_targetgroup: Ramwebapp-tg1

ec2\_name: Ram\_webserver1

sns\_topic\_name: Ram-sns

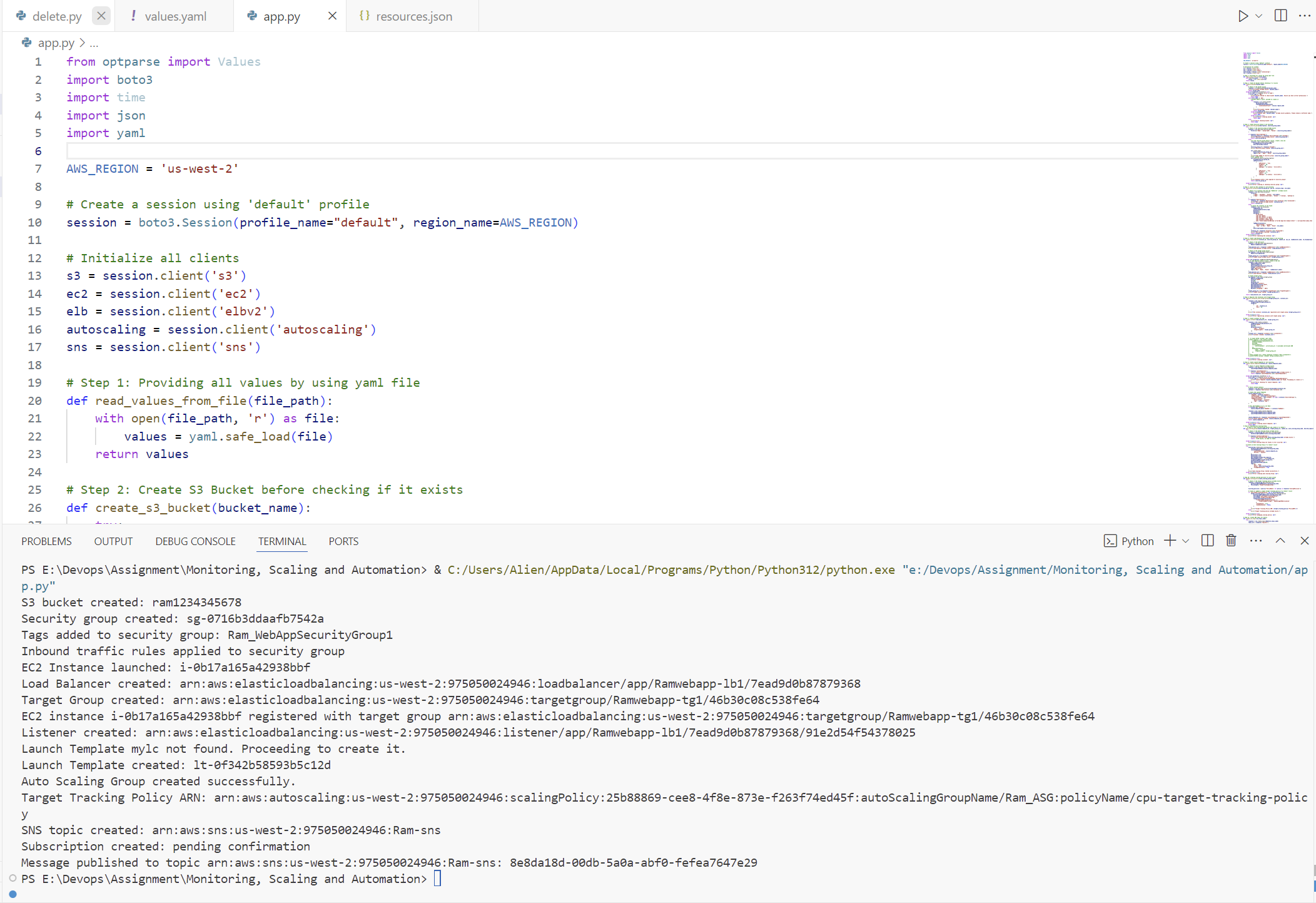
protocol: email

email\_id: [ramasrinivas495@gmail.com](mailto:ramasrinivas495@gmail.com)

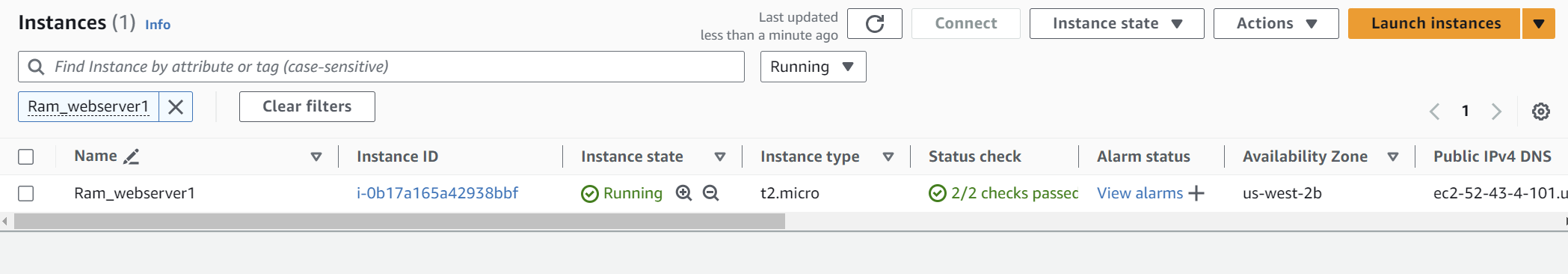
subject: 'Test Alert'

message: 'This is a test alert for health issues'

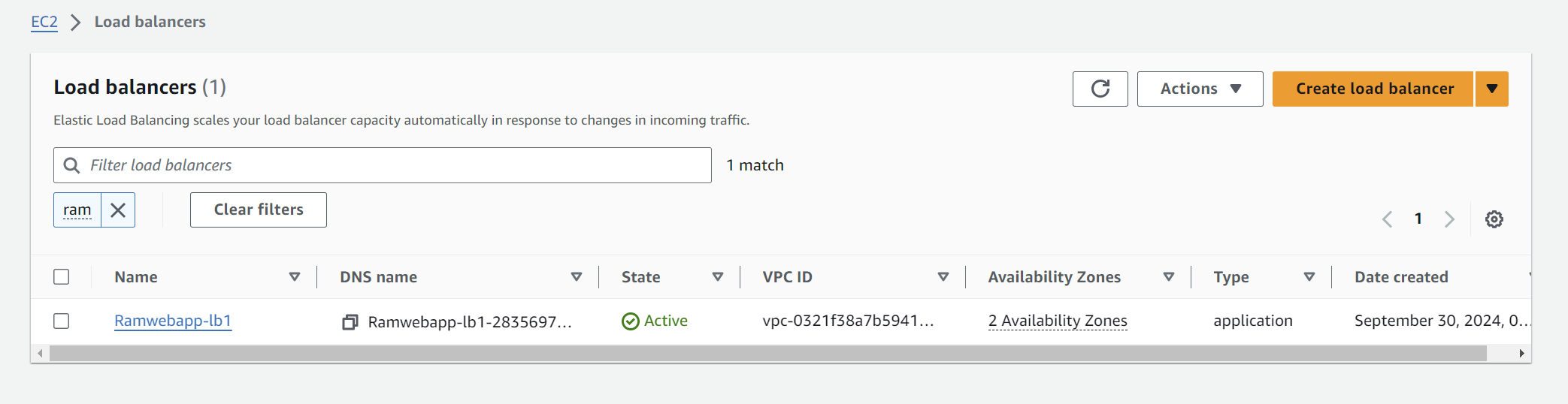
**Execute the app.py file**

****

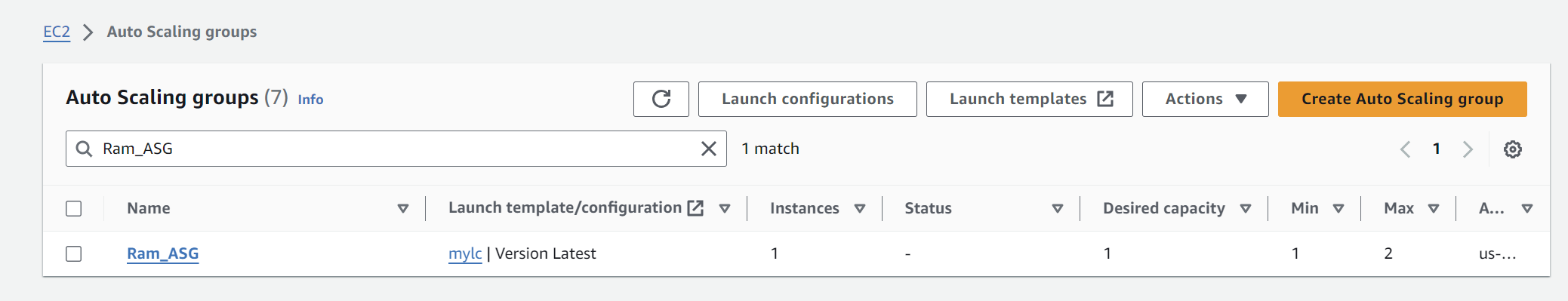
EC2 created- Ram\_webserver1



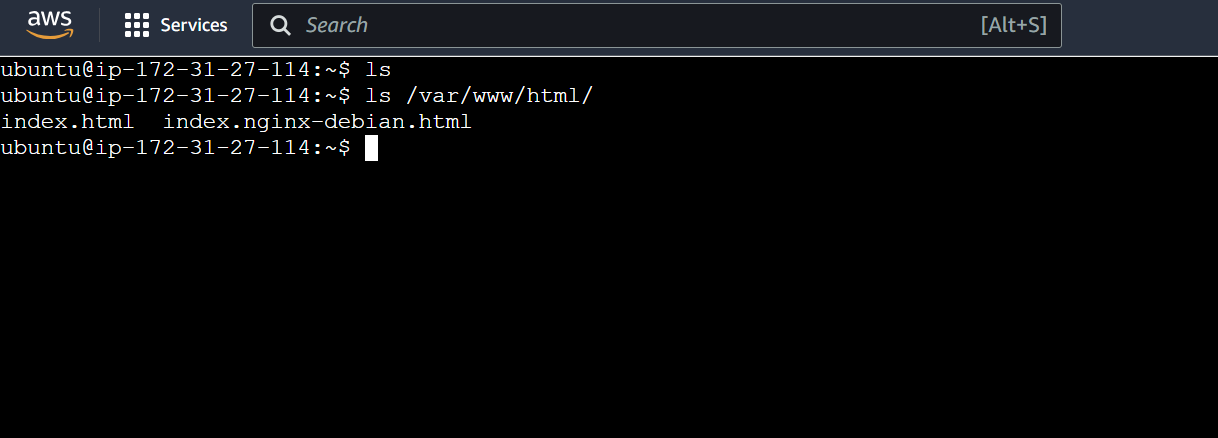
2. Load balancer-Ramwebapp-lb1



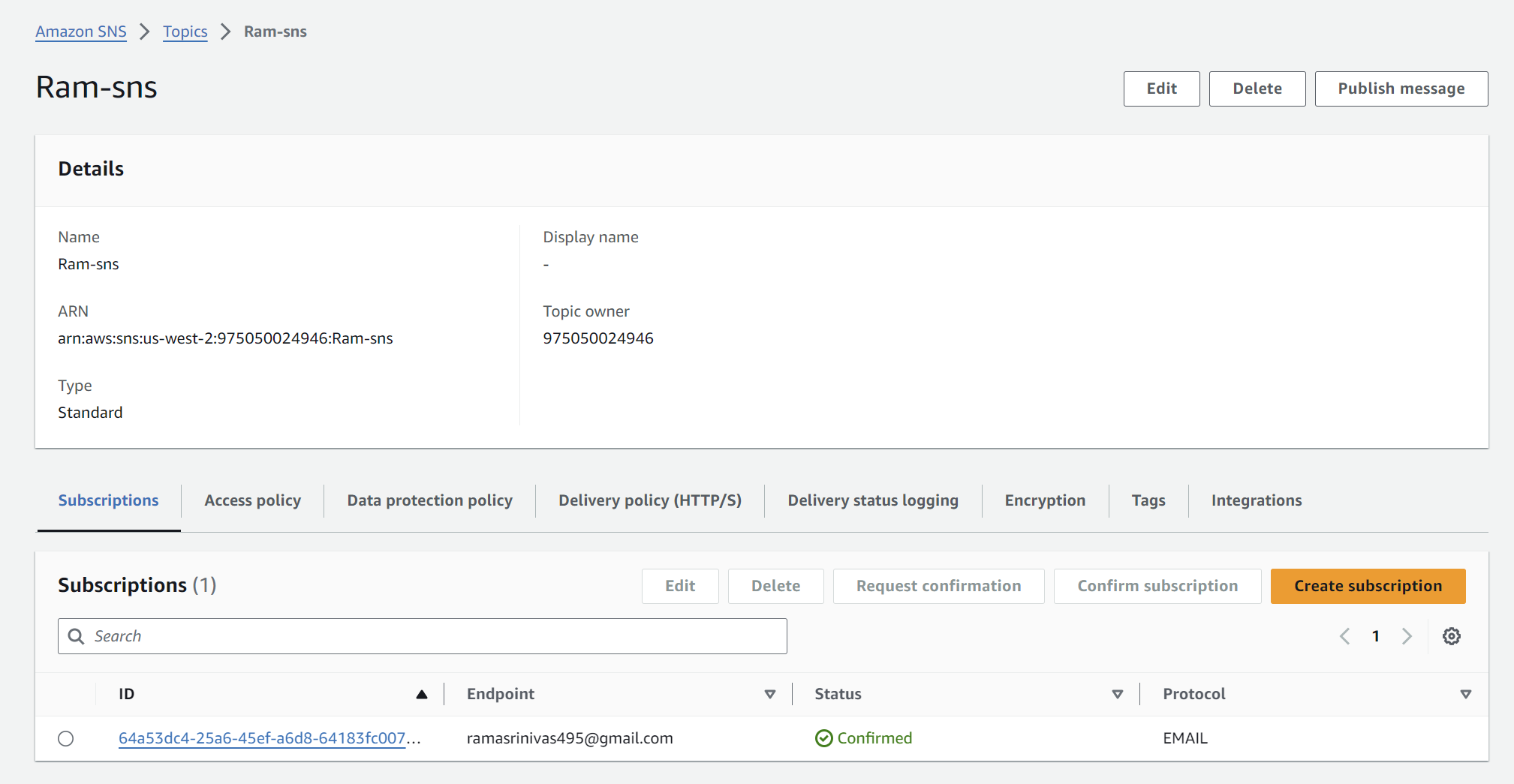
3. ASG created – Ram\_ASG

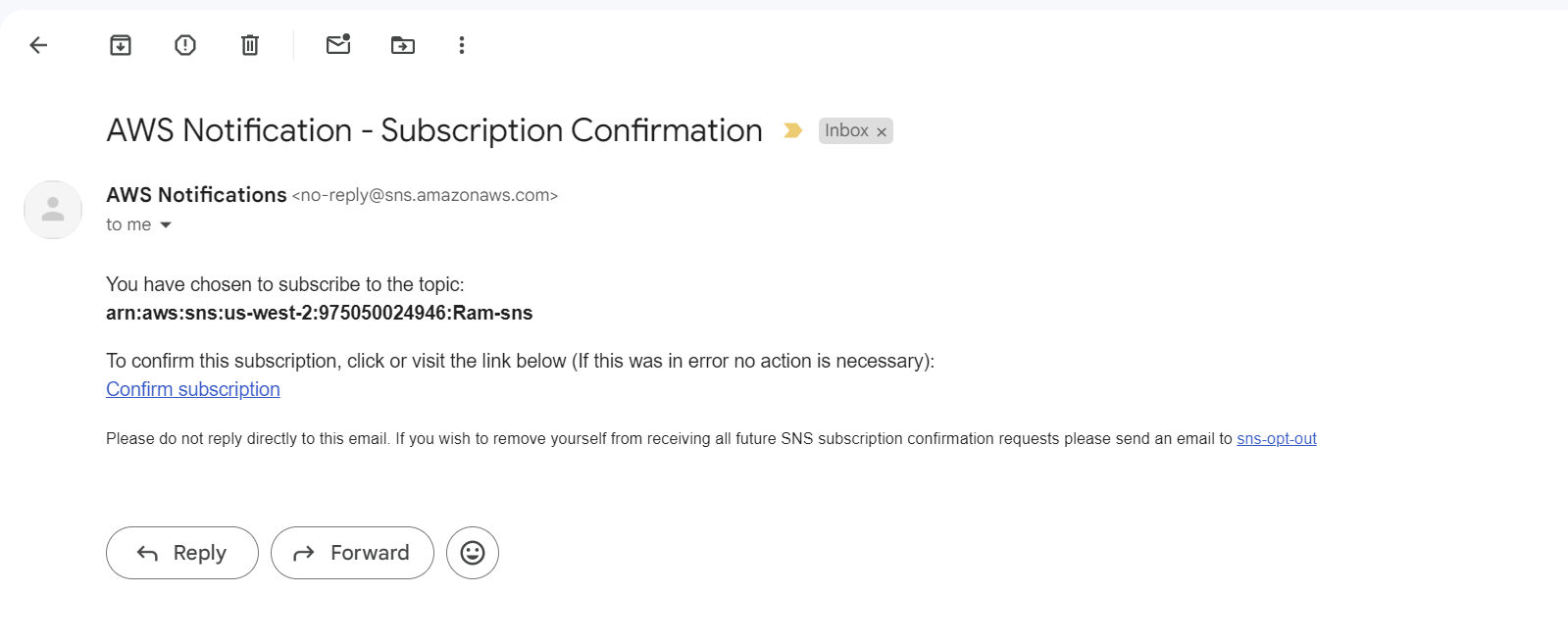


Index file create in /var/www/html/



4. SNS Notifications:





5. Infrastructure Automation:

Create python for deletion and execute

import boto3

import json

import time

AWS\_REGION = 'us-west-2'

# Create a session using 'herovired' profile

session = boto3.Session(profile\_name="default", region\_name=AWS\_REGION)

# Initialize all clients

s3 = session.client('s3')

ec2 = session.client('ec2')

elb = session.client('elbv2')

autoscaling = session.client('autoscaling')

sns = session.client('sns')

def delete\_resources():

    try:

        with open('resources.json', 'r') as f:

            resources = json.load(f)

        # Deleting S3 bucket

        try:

            print(f"Deleting S3 bucket: {resources['bucket\_name']}...")

            s3.delete\_bucket(Bucket=resources['bucket\_name'])

            print("S3 bucket deleted successfully.")

        except s3.exceptions.ClientError as e:

            print(f"Error deleting S3 bucket: {e}")

        # Deleting Auto Scaling Group

        try:

            print(f"Deleting Auto Scaling Group: {resources['auto\_scaling\_group\_name']}...")

            autoscaling.delete\_auto\_scaling\_group(AutoScalingGroupName=resources['auto\_scaling\_group\_name'], ForceDelete=True)

            print("Auto Scaling Group deleted successfully.")

        except Exception as e:

            print(f"Error deleting Auto Scaling Group: {e}")

        # Deleting Launch Templete

        try:

            print(f"Deleting Launch Template: {resources['launch\_template\_id']}...")

            ec2.delete\_launch\_template(LaunchTemplateId=resources['launch\_template\_id'])

            print("Launch Template deleted successfully.")

        except Exception as e:

            print(f"Error deleting Launch Template: {e}")

        # Deleting Load Balancer

        try:

            print(f"Deleting Load Balancer: {resources['load\_balancer\_arn']}...")

            elb.delete\_load\_balancer(LoadBalancerArn=resources['load\_balancer\_arn'])

            print("Load Balancer deleted successfully.")

        except elb.exceptions.LoadBalancerNotFoundException:

            print("Load Balancer not found. It may have already been deleted.")

        except Exception as e:

            print(f"Error deleting Load Balancer: {e}")

        # Deleting EC2 instance

        try:

            print(f"Terminating EC2 Instance: {resources['instance\_id']}...")

            ec2.terminate\_instances(InstanceIds=[resources['instance\_id']])

            waiter = ec2.get\_waiter('instance\_terminated')

            print("Waiting for instance to terminate...")

            waiter.wait(InstanceIds=[resources['instance\_id']])

            print("EC2 Instance terminated successfully.")

        except Exception as e:

            print(f"Error terminating EC2 Instance: {e}")

        # Deleting Target Group

        try:

            print(f"Deleting Target Group: {resources['target\_group\_arn']}...")

            elb.delete\_target\_group(TargetGroupArn=resources['target\_group\_arn'])

            time.sleep(60)

            print("Target Group deleted successfully.")

        except elb.exceptions.TargetGroupNotFoundException:

            print("Target Group not found. It may have already been deleted.")

        except Exception as e:

            print(f"Error deleting Target Group: {e}")

        # Deleting Security Group

        try:

            print(f"Deleting Security Group: {resources['security\_group\_id']}...")

            ec2.delete\_security\_group(GroupId=resources['security\_group\_id'])

            print("Security Group deleted successfully.")

        except ec2.exceptions.ClientError as e:

            print(f"Error deleting Security Group: {e}")

        # Deleting SNS Topic

        try:

            print(f"Deleting SNS Topic: {resources['sns\_topic']}...")

            sns.delete\_topic(TopicArn=resources['sns\_topic'])

            print("SNS Topic deleted successfully.")

        except sns.exceptions.ClientError as e:

            print(f"Error deleting SNS Topic: {e}")

        print("Resource deletion process completed.")

    except Exception as e:

        print(f"An error occurred during resource deletion: {e}")

# Call the delete\_resources function

delete\_resources()

Successfully executed:

Tears down everything when the application is no longer needed. Run the delete.py to delete the application

