Automated data management system on AWS – DynamoDB and EC2

Michael Ohene Aboagye

Cloud Engineer

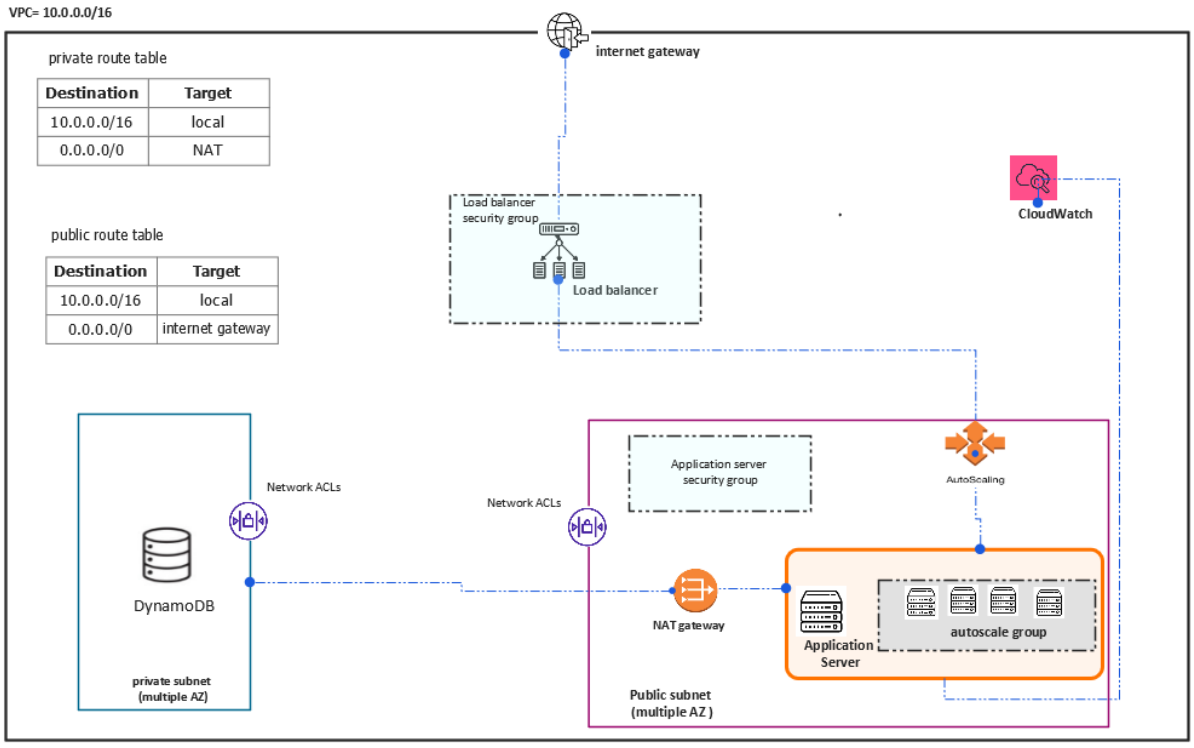
7/7/2025

SUMMARY

This project demonstrates the setup of a cloud solution, specifically a logger database using DynamoDB. Security measures were prioritized by creating a custom Virtual Private Cloud (VPC), setting up subnets for the respective instances, and implementing security features such as Network Access Control Lists (NACLs) and security groups.

To enhance the reliability of the application, load balancing and auto-scaling features were implemented. The load balancer distributes incoming traffic evenly to an auto-scaling group, allowing for the addition of instances to handle increased loads, thus minimizing potential downtime.

Python 3 was used as the scripting tool to automate the logging of data into the DynamoDB database, utilizing Boto3, the Amazon SDK for Python.



fig

fig.1 Automated data management system on AWS Architecture Diagram

1. SETTING UP THE NETWORK ARCHITECTURE

1.1. Create a **VPC** which serves as an isolated space on AWS cloud environment to host application,

Set VPC with CIDR block of 10.0.0.0/16

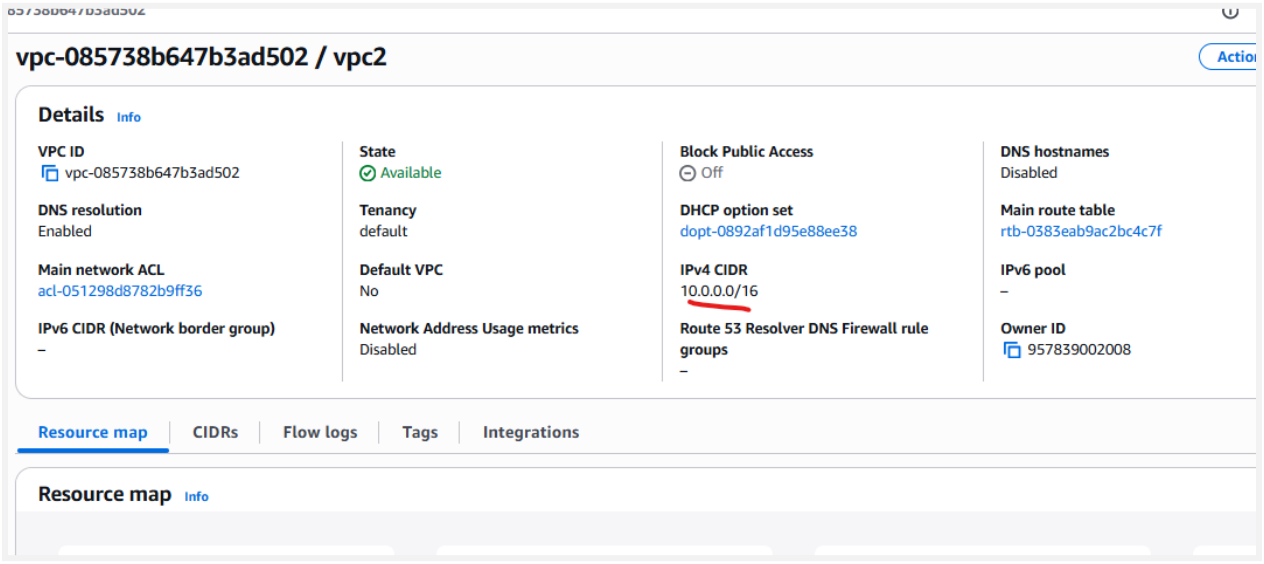


Fig 2. VPC configuration

1.2. Now setup subnets to host the instances and resource.

Public Subnets:
Created in two Availability Zones.
Assigned for Load Balancer and NAT Gateway.
CIDRs: 10.0.1.0/24, 10.0.3.0/24

Private Subnets:
Created in different AZs for EC2 and backend apps.
CIDRs: 10.0.2.0/24, 10.0.4.0/24

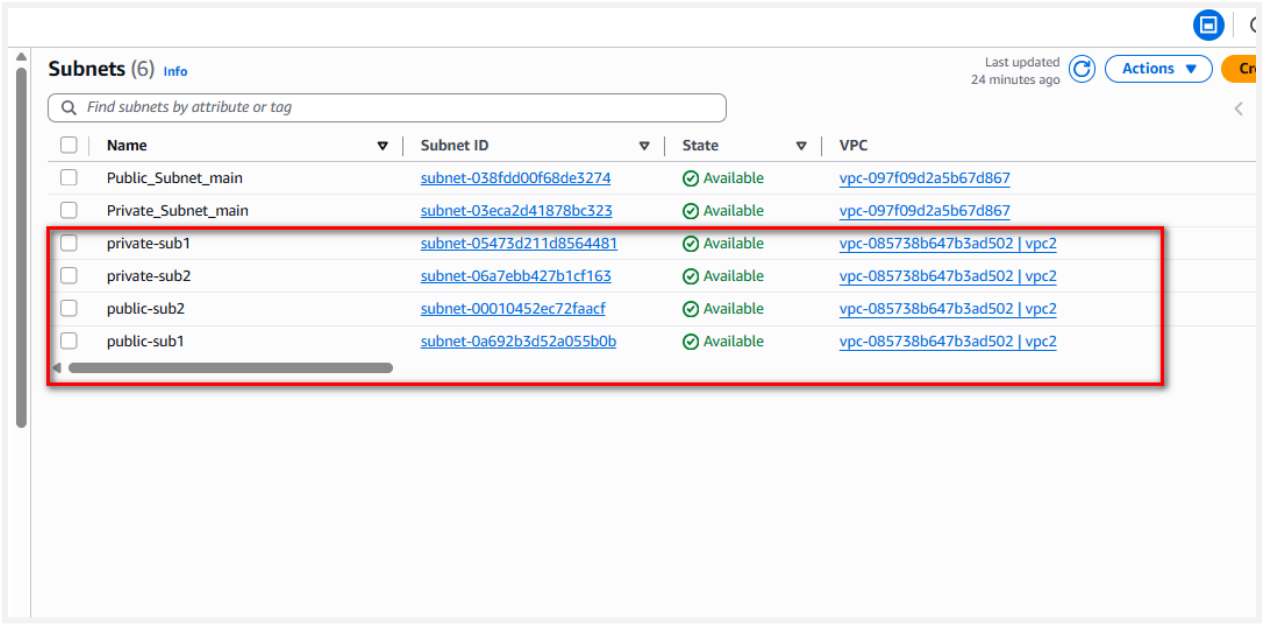


Fig 3. Public and private subnet for the DynamoDB and the application server.

Private and public subnets was created, the application server was placed in the private subnet to allow access to the internet.

The DynamoDB instance was placed in the private subnet to allow internal traffic only.

1.3. set route tables, NAT gateways

In other to control traffic, both internal and external, two route tables was created. Public route to control public traffics and private route table to control the internal traffics.

to control traffic from the public subnet, **public route table** was attached to the public subnets control inbound and outbound traffic. The traffic was restricted to HTTP, HTTPS, TCP and SSH.

Private routable was attached to the **private subnet** to route traffic internally and also to access the internet via the NAT gateway specifically for downloads and updates.

Public Route Table:

Route: 0.0.0.0/0 → Internet Gateway

Associated with Public Subnets

Private Route Table:

Route: 0.0.0.0/0 → NAT Gateway

Associated with Private Subnets

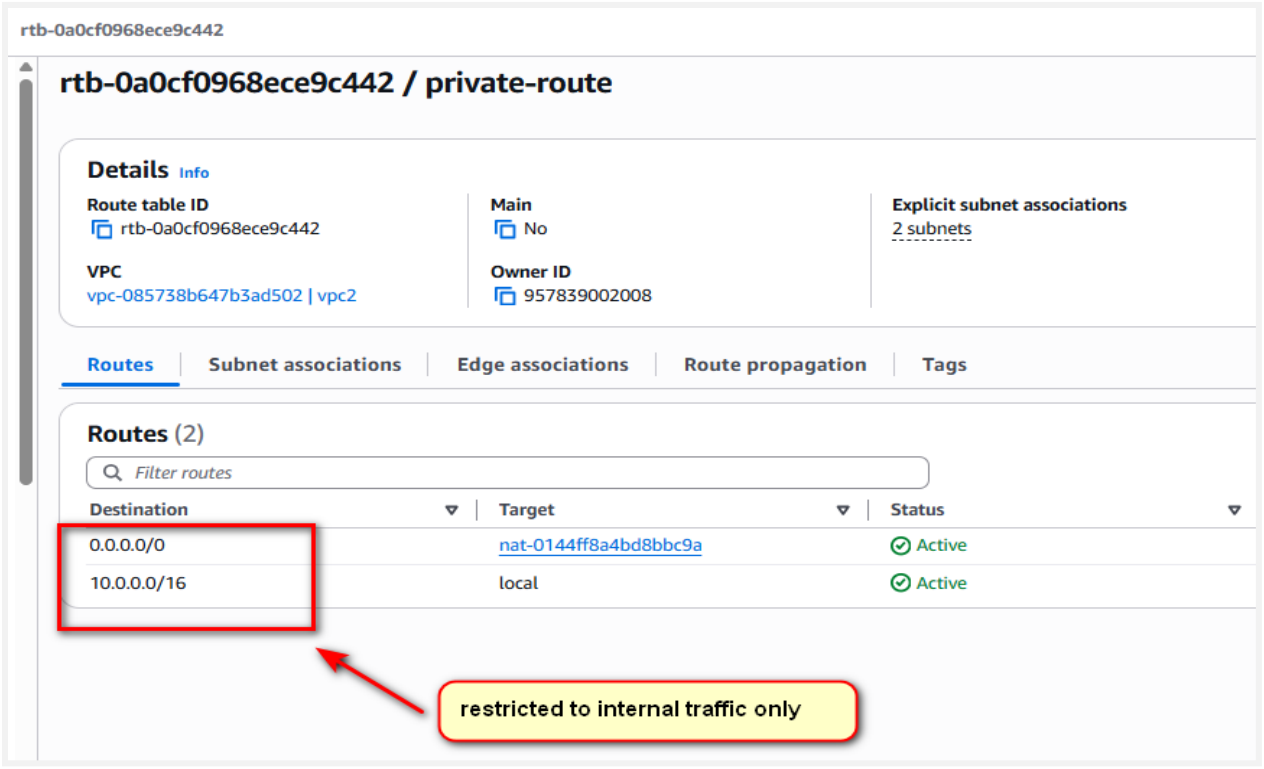


Fig 4. Setting up private route tables and attaching to the private subnets.

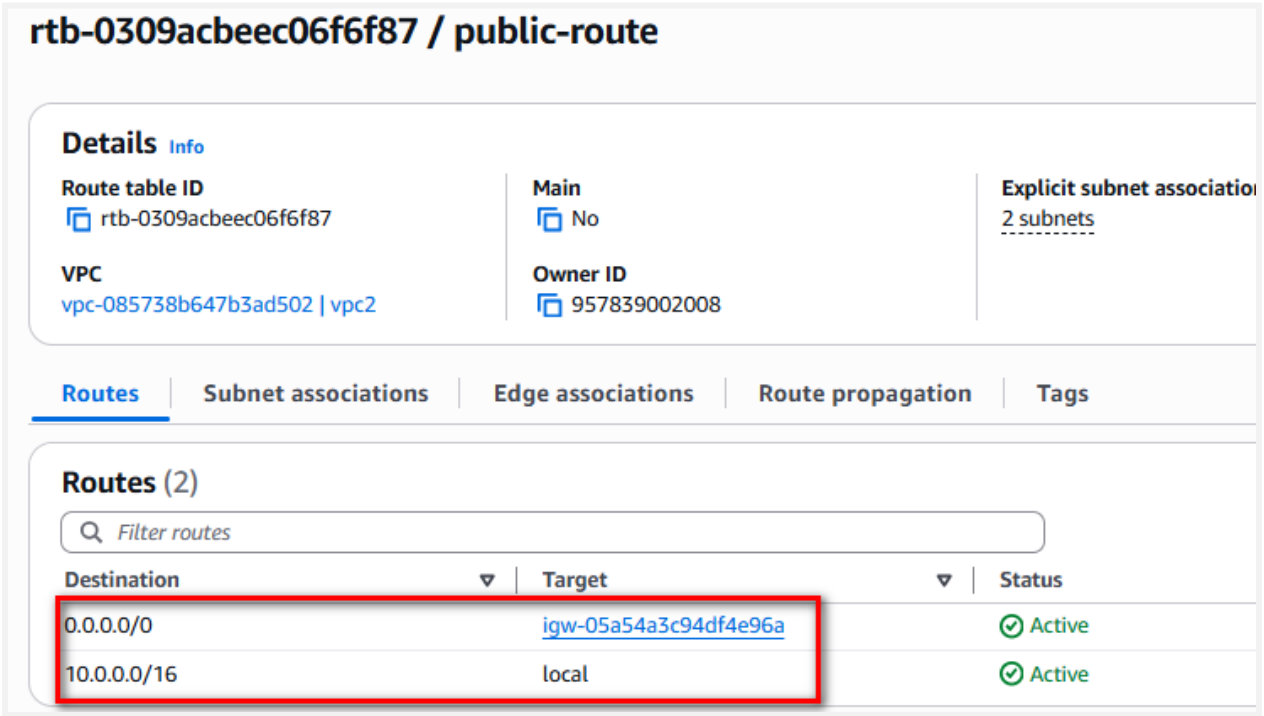


Fig 5. Public route controls traffic outside the VPC. It routes traffic to the internet via the internet gateway.

1.4. setup security groups for the load balancer as "**load-balancer-sg**" and the EC instances/application server(s) as "**app-server-sg**".

First, we create security group for the load balancer to filter out inbound traffic to HTTP, HTTPS, and SSH.

then, we set up security group for the EC2 instance to allow traffic from the load balancer only.

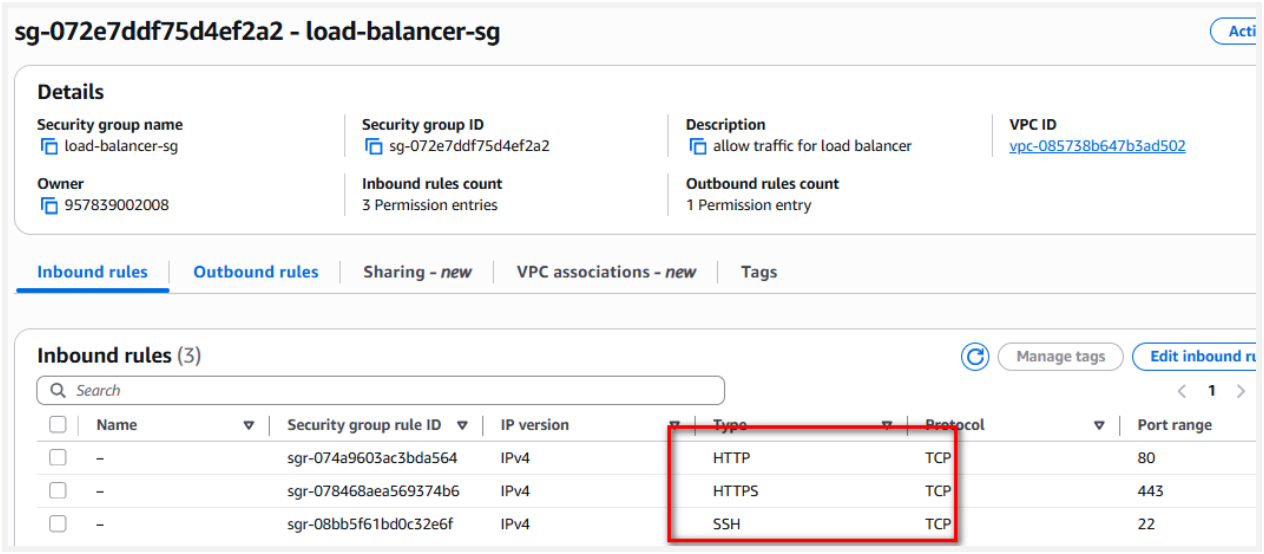


Fig 6. Load balancer security group configuration.

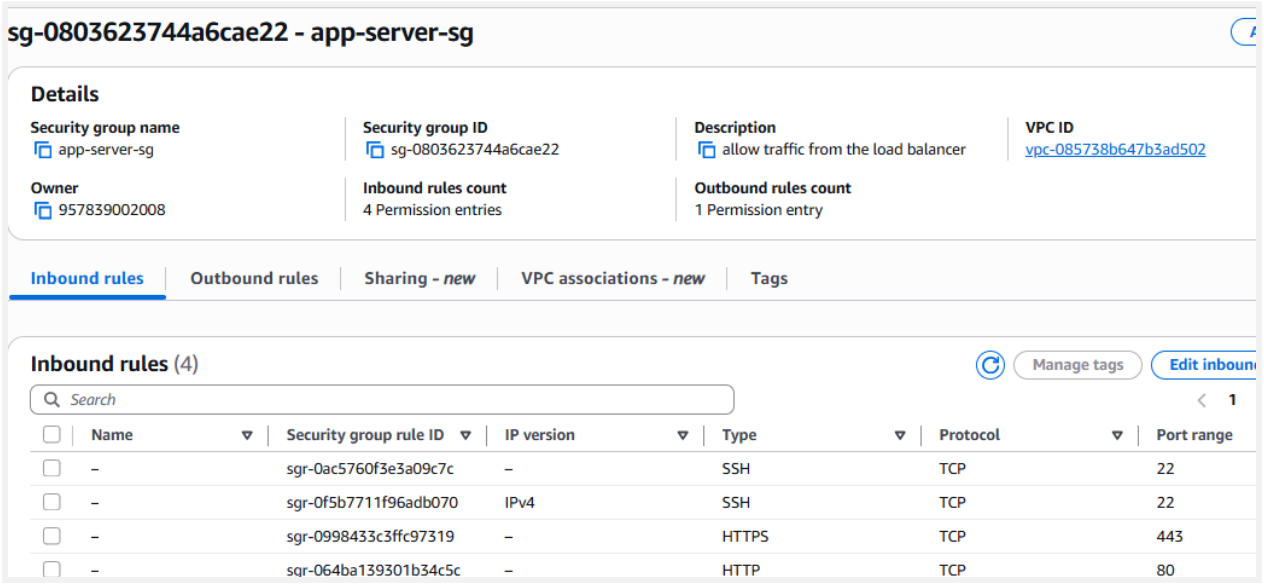


Fig 7. EC2 instance security group configuration.

2. PREPARE THE EC2 INSTANCE.

2.1. Create an EC2 instance with the required configurations and test for connectivity.

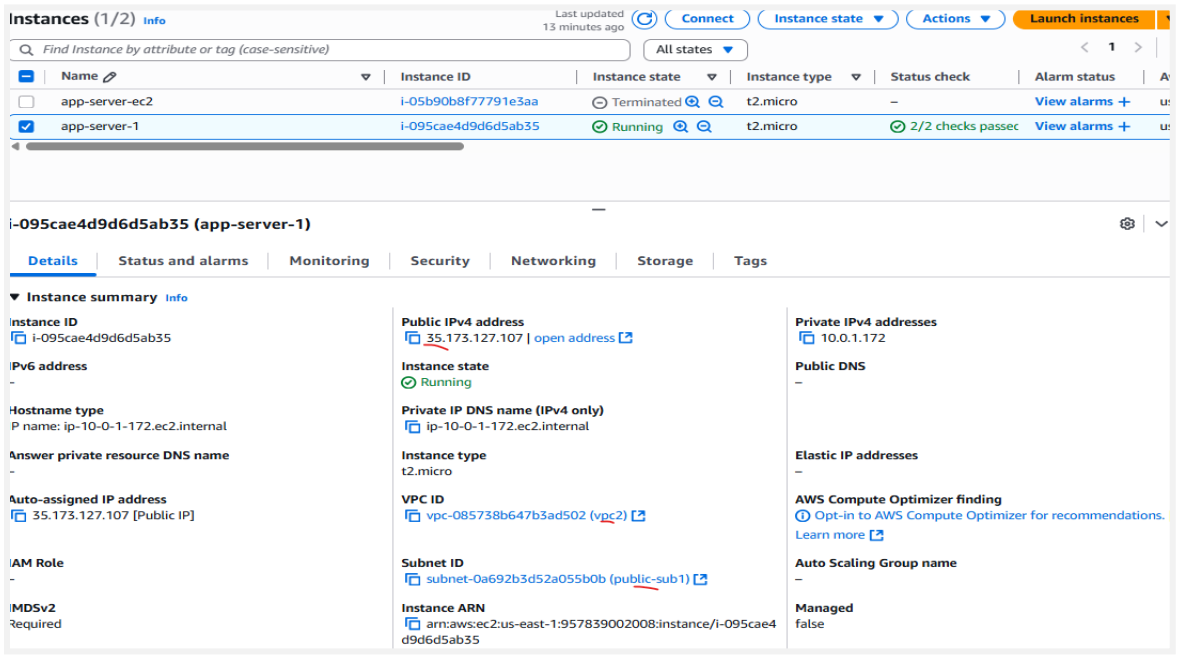


Fig 8. EC2 instance detail configuration.

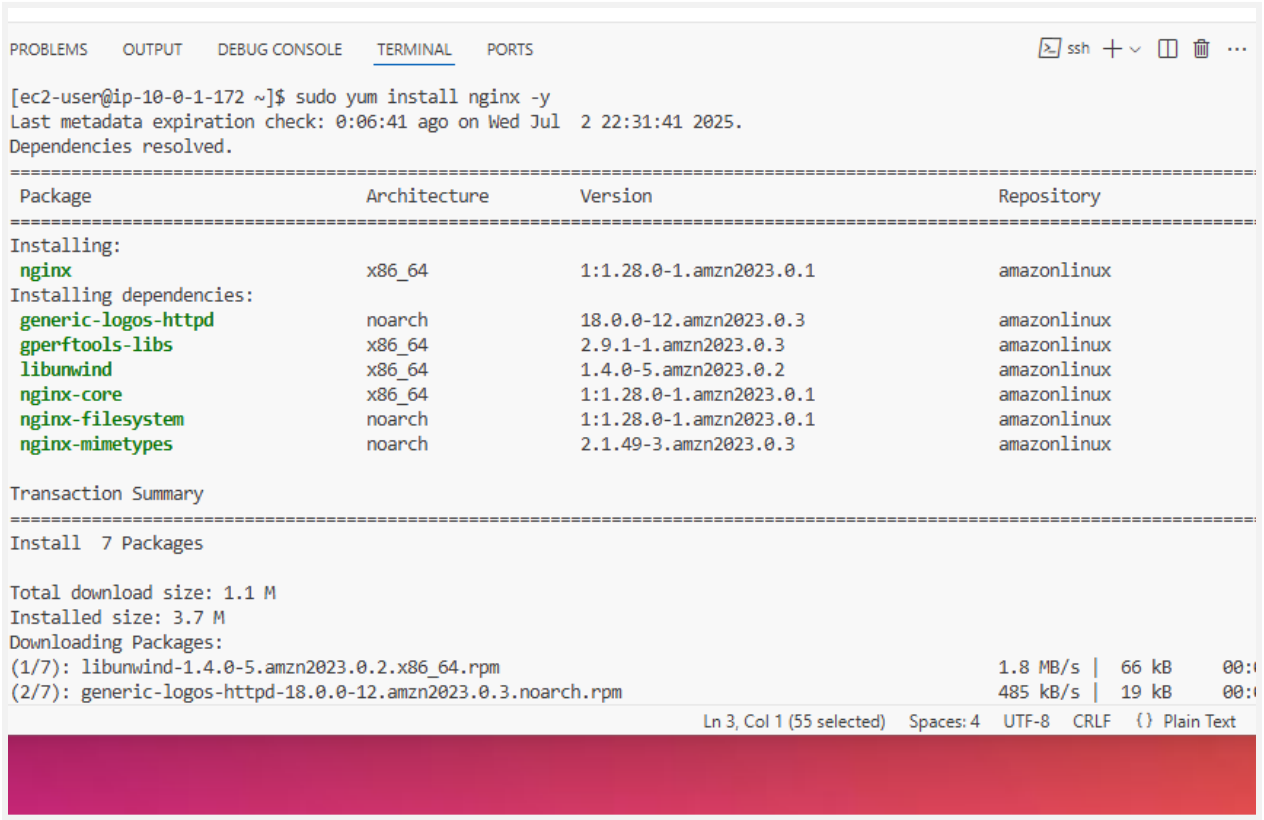
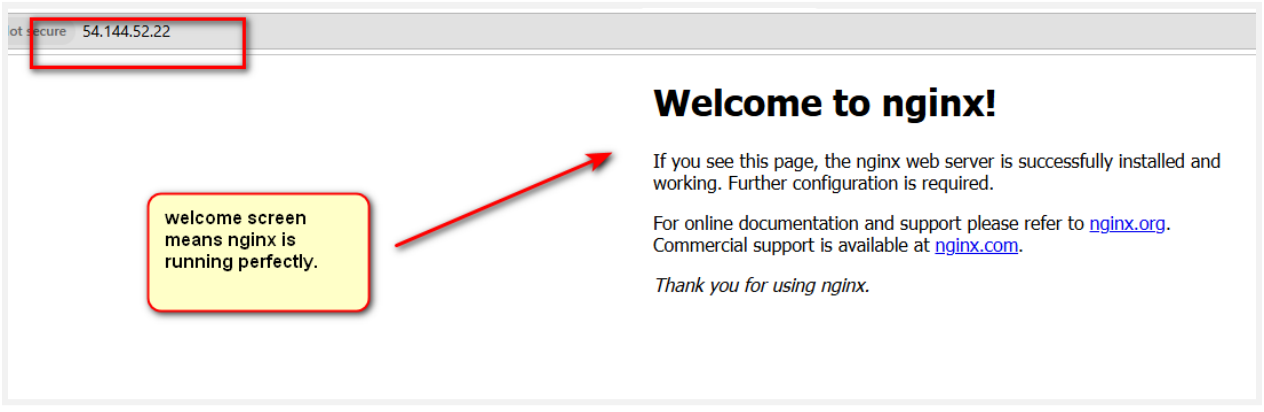


Fig 9. Nginx was installed and run to use as test for internet connectivity.

Run the code below on the terminal to start `nginx` application.

"sudo systemctl start nginx"

nginx welcome screen will be displayed on the web browser by running the public IP address of the EC2 instance.



2.2. Create launch template from the current instance, this will be used by the auto-scale to create preconfigured instances as needed.

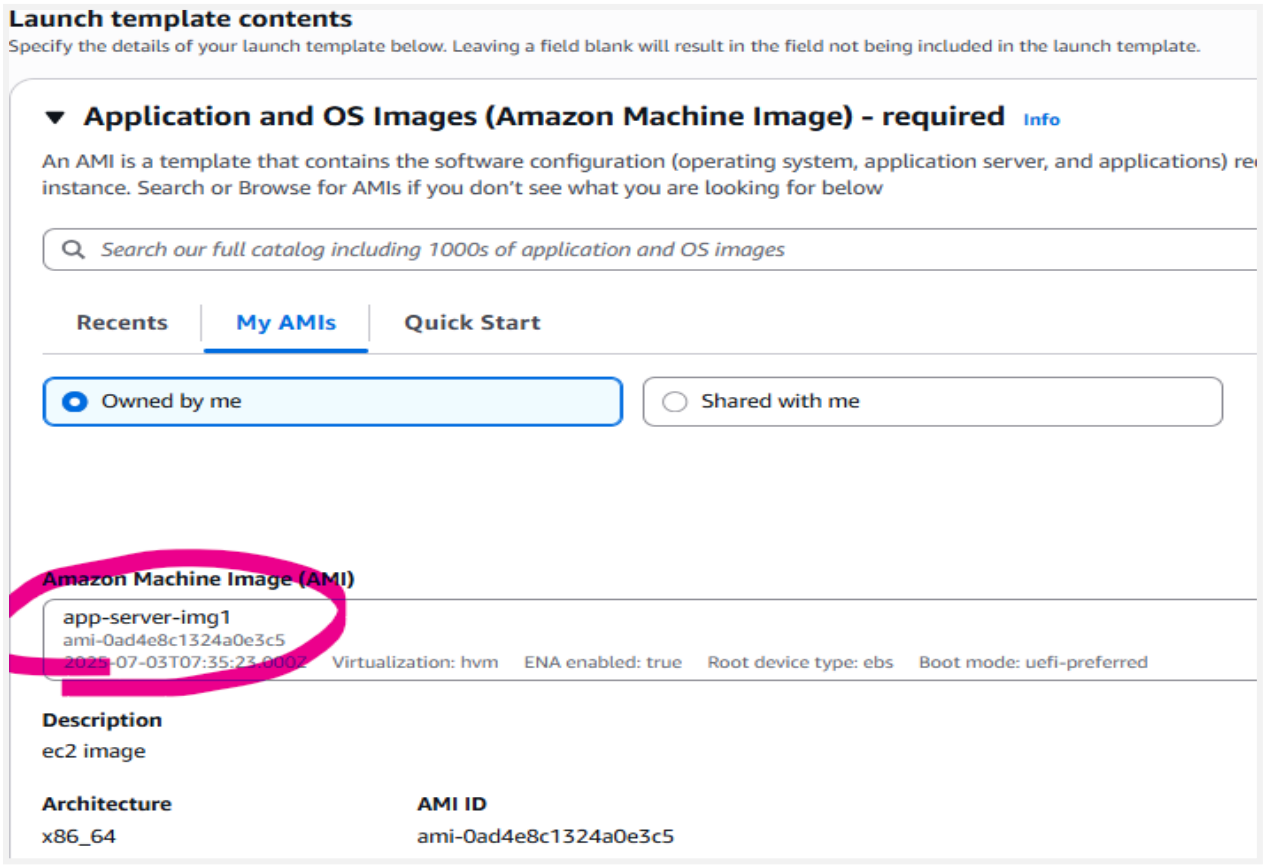
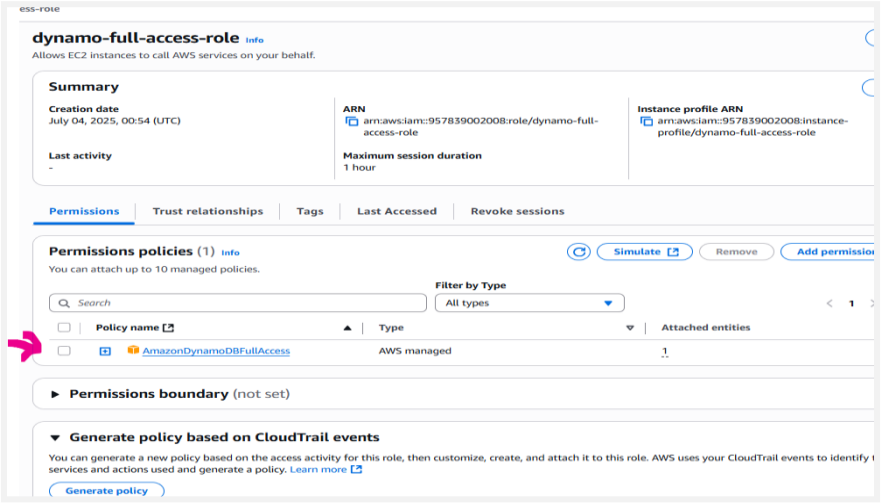


Fig 10. Creating launch template from the preconfigured AMI

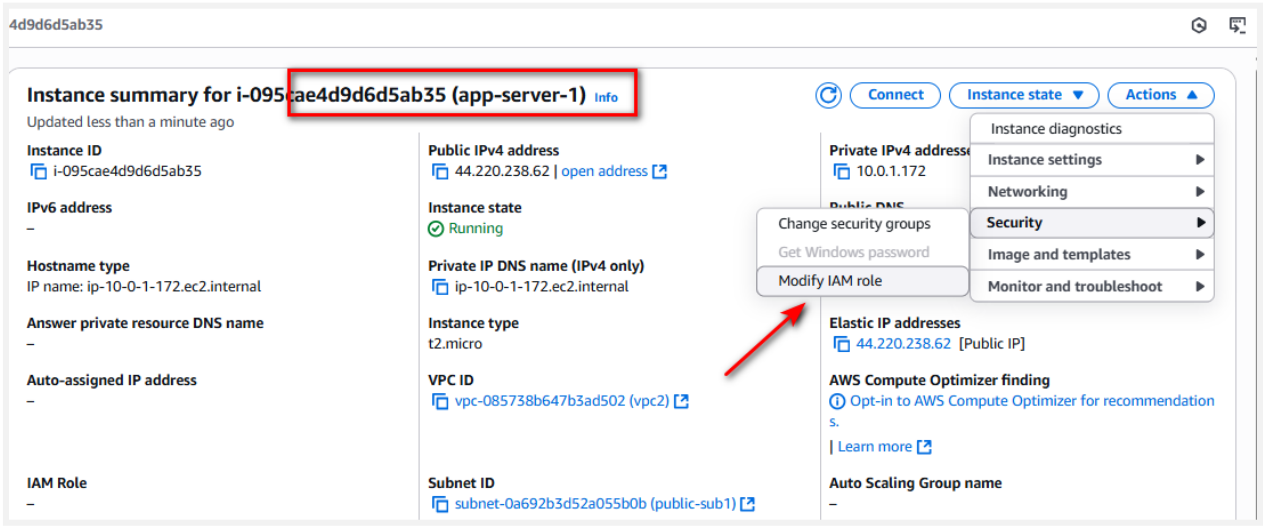
2.3. set up IAM role for the ec2 instance to access DynamoDB resource.

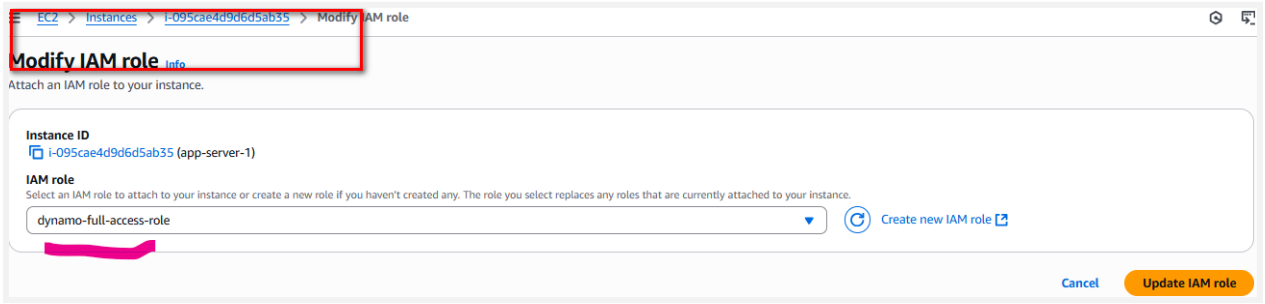
Here, the ec2 instance was allowed administrative access to DynamoDB resource.

i. Create role (AmazonDynamoDBFUllAccess)



ii. Attach role to the instance





3. SETUP LOAD BALANCER AND AUTOSCALING GROUP.

3.1. set up the load balancer, set the listeners to the required target group, then load balancer security group was attached (filters traffic to only HTTP, HTTPS, SSH).

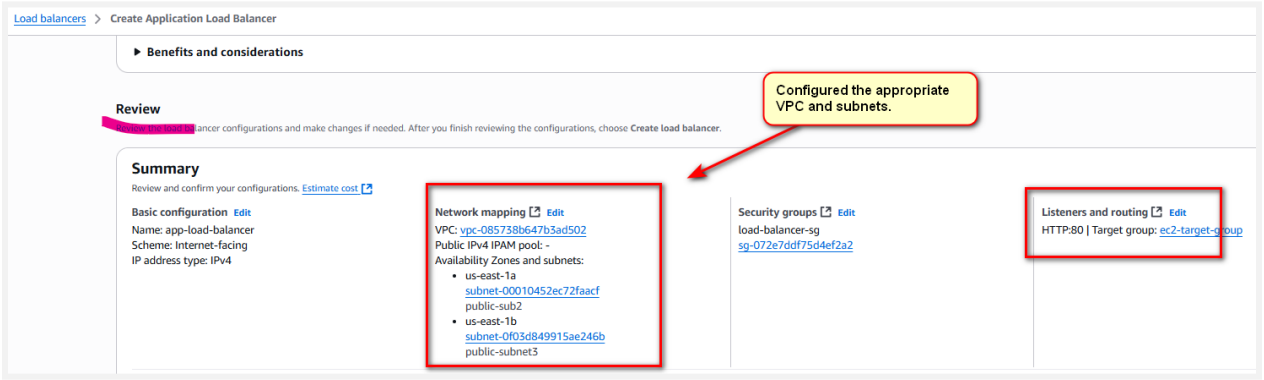


Fig 11. Summary of the load balancer configuration.

3.2. The auto-scale group was configured.

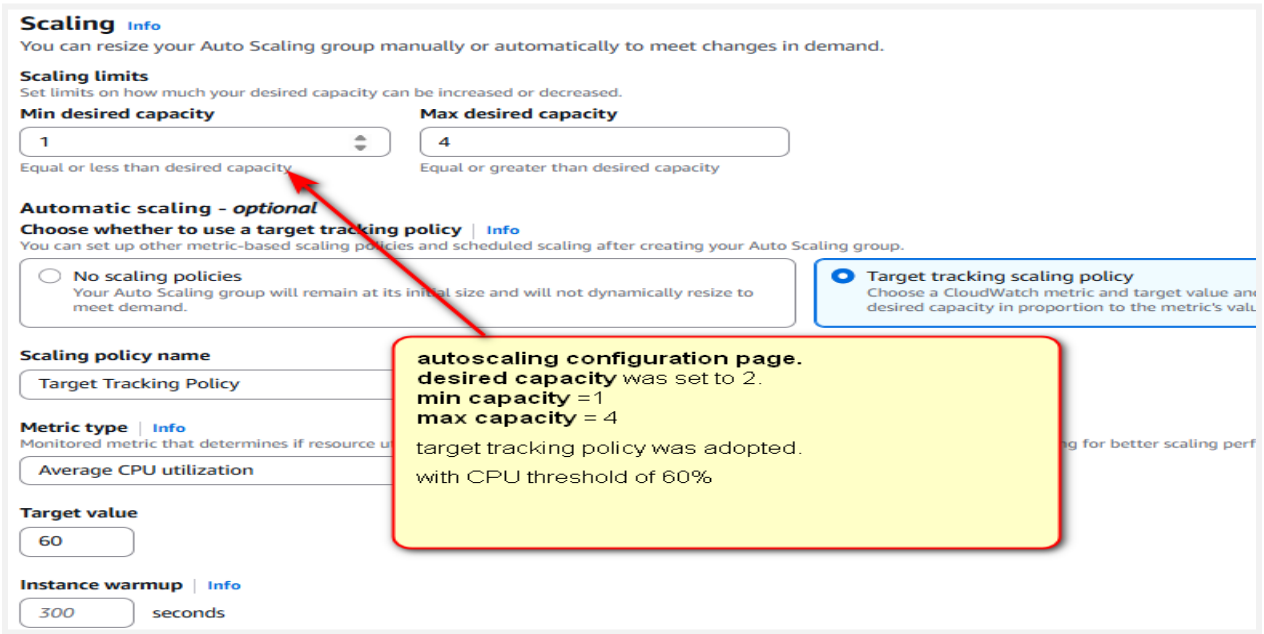


Fig 12. Auto-scale configuration.

We set the scaling limits with desired capacity of 2, minimum capacity of 1 and maximum capacity of 4

This means on normal workload, two instances will be actively running. When there this minimum CPU for a period of time, auto-scale terminates the instances and scale-in to only 1 instance. In a situation where there is high amount of workload, auto-scale scales out the number of instance to 4

3.3. the load balancer and autoscaling configurationn was tested for connectivity and response to incoming workload.

EC2 Instance connectitvity is tested by running the DNS name of the load balancer, while making sure the nginx is still running on the instance.

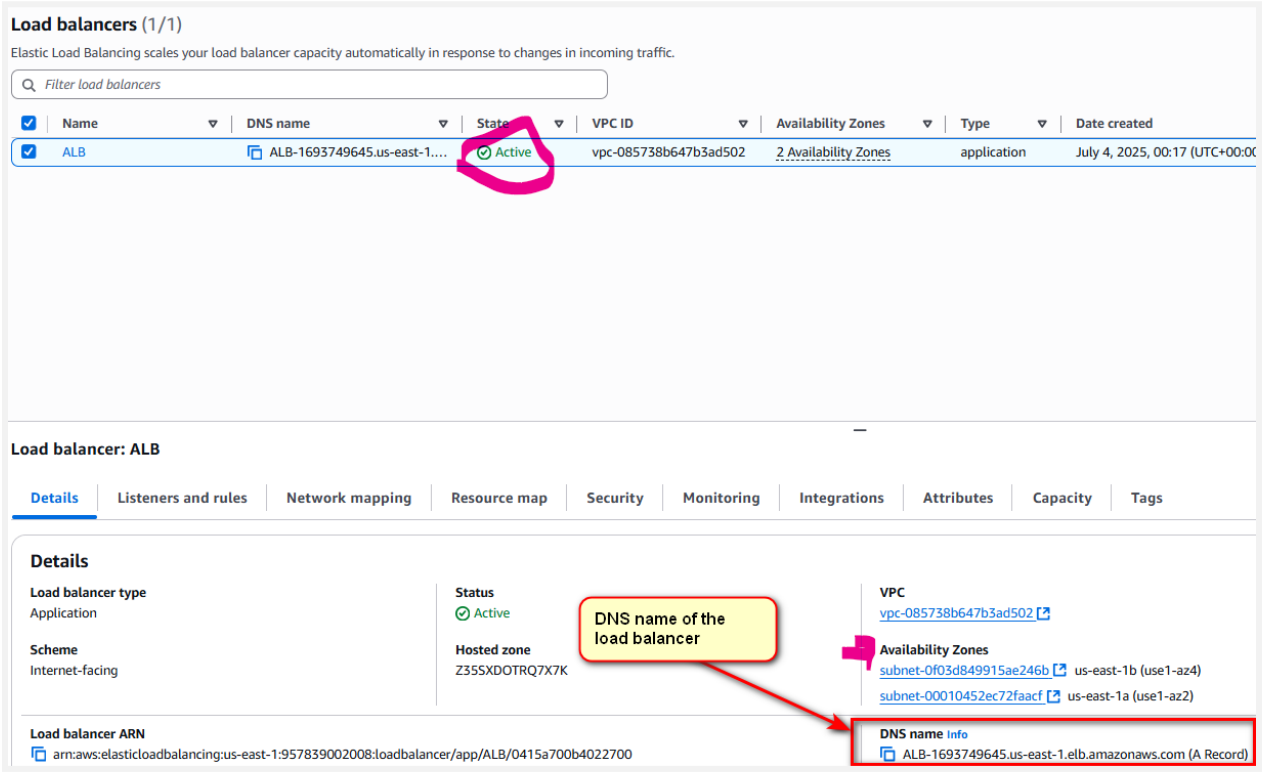


Fig 13. Load balancer with active DNS name.

Fig 14. Load balancer DNS name tested for connectivity.

3.4. we test for the Auto-Scale response of the load balancer on traffic.

i. We install and apply stress test using the code below.

```
sudo yum install stress -y  #For Amazon Linux

stress --cpu 4 --timeout 60 #configures and run stress test.
```

ii. After applying the stress test, the auto-scale group adds new instance when the CPU exceed the **60%** threshold and terminated instance when in goes below the 60% threshold.

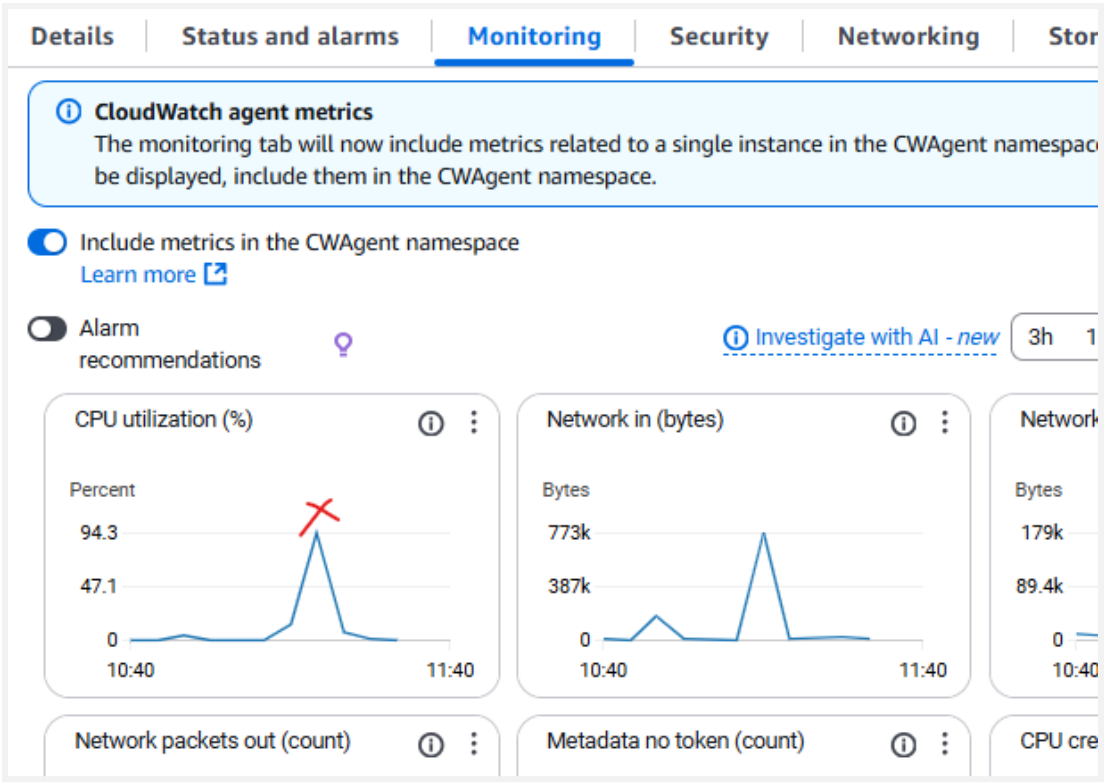


Fig 15. CPU utilization graph exceeding 60% threshold value.

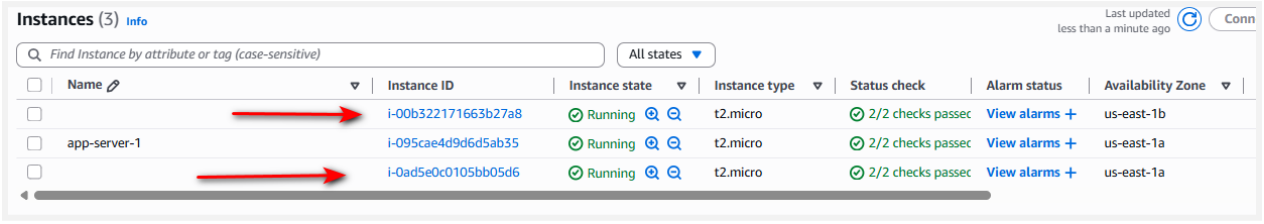


Fig.16 new instance being deployed due to auto-scale configuration.

4. SETUP DYNAMODB TABLE, CONFIGURE PYTHON TO CONNECT TO DYNAMODB

4.1. Python3 and boto3, which is an Amazon SDK for python. The code below was used to configure python and boto3 (assuming python3 and pip3 is already installed).

```
pip install boto3
```

```
[ec2-user@ip-10-0-1-172 ~]$ pip3 install boto3
Defaulting to user installation because normal site-packages is not writeable
Collecting boto3
  Downloading boto3-1.39.3-py3-none-any.whl (139 kB)
     |████████████████████████████████| 139 kB 15.8 MB/s
Collecting botocore<1.40.0,>=1.39.3
  Downloading botocore-1.39.3-py3-none-any.whl (13.8 MB)
     |████████████████████████████████| 13.8 MB 41.7 MB/s
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /usr/lib/python3.9/site
oto3) (0.10.0)
Collecting s3transfer<0.14.0,>=0.13.0
  Downloading s3transfer-0.13.0-py3-none-any.whl (85 kB)
     |████████████████████████████████| 85 kB 7.9 MB/s
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/lib/python3.9
from botocore<1.40.0,>=1.39.3->boto3) (2.8.1)
```

Fig 17. Installing boto3 through the EC2 CLI.

4.2. Staff_logger.py file was created on the EC2 instance using nano text editor, it contained the code to automatically logs staff information to the table.

staff-logger

```python
import boto3# Create a DynamoDB resource using the us-east-1 region

dynamodb= boto3.resource('dynamodb', region_name='us-east-1') # Reference your DynamoDB table

table= dynamodb.Table('staff') # List of staff to add

staff = [ {'StaffID': '001', 'name': 'Alice Appiah', 'department': 'IT'},
          {'StaffID': '002', 'name': 'Rita Okai', 'Department': 'Finance'},
          {'StaffID': '001', 'name': 'Ansong Ampaw', 'Department': 'IT'},
          {'StaffID': '002', 'name': 'Rita Okloo', 'Department': 'Security'},
          {'StaffID': '001', 'name': 'Alice Mensah', 'Department': 'IT'},
          {'StaffID': '002', 'name': 'Rita Okloo', 'Department': 'Finance'},
          {'StaffID': '001', 'name': 'Philip Mensah', 'Department': 'IT'},
          {'StaffID': '002', 'name': 'Rita Yao', 'Department': 'Product'},
          {'StaffID': '001', 'name': 'Alice Appiah-Sarpong', 'Department': 'IT'},
          {'StaffID': '002', 'name': 'Akoto Richard', 'Department': 'Finance'}]

# Loop through and insert each staff

for emp in staff:
    response = table.put_item(Item=emp)
    print(f"Inserted: {emp['name']} | Status: {response['ResponseMetadata']['HTTPStatusCode']}")
```

Fig 18. Python script to logg staff data to the dynamoDB.

4.3. After running the script "`staff-logger.py`" on the EC2 terminal, outputs of data insertion was displayed.

```
[ec2-user@ip-10-0-1-172 ~]$ nano staff-logger.py
[ec2-user@ip-10-0-1-172 ~]$ python3 staff-logger.py
Inserted: Alice Appiah | Status: 200
Inserted: Rita Okai | Status: 200
Inserted: Ansong Ampaw | Status: 200
Inserted: Rita Okloo | Status: 200
Inserted: Alice Mensah | Status: 200
Inserted: Rita Okloo | Status: 200
Inserted: Philip Mensah | Status: 200
Inserted: Rita Yao | Status: 200
Inserted: Alice Appiah-Sarpong | Status: 200
Inserted: Akoto Richard | Status: 200
[ec2-user@ip-10-0-1-172 ~]$
```

Fig 19. Running staff_logger.py on the EC2 instance/ application server.

4.4. By checking the DynamoDB table on the console, the data was automatically populated to the table after running the python script.

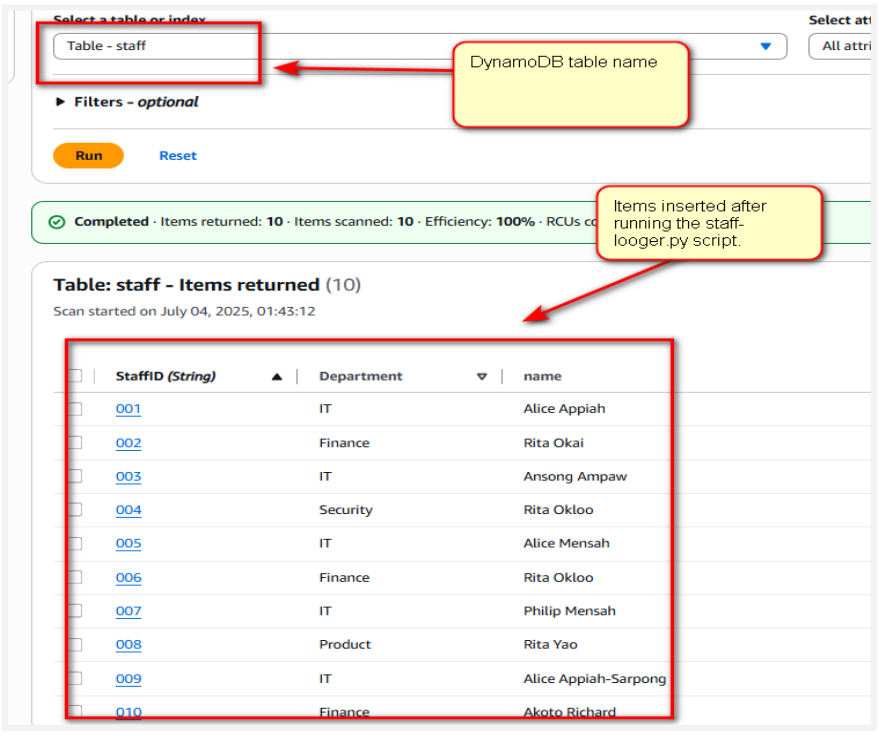


Fig 20. DynamoDB table

CONCLUSION

This project effectively showcased the capabilities of AWS cloud technologies in building a scalable, secure, and highly available architecture. By leveraging Auto Scaling, Elastic Load Balancing (ELB), and DynamoDB, we efficiently managed dynamic workloads and ensured low-latency data access. Security was reinforced through IAM, VPC, and CloudWatch, while adherence to AWS best practices enabled cost-efficiency, resilience, and automation.