Task-6.

Cost Optimization

Use free-tier eligible instances

Minimal resources

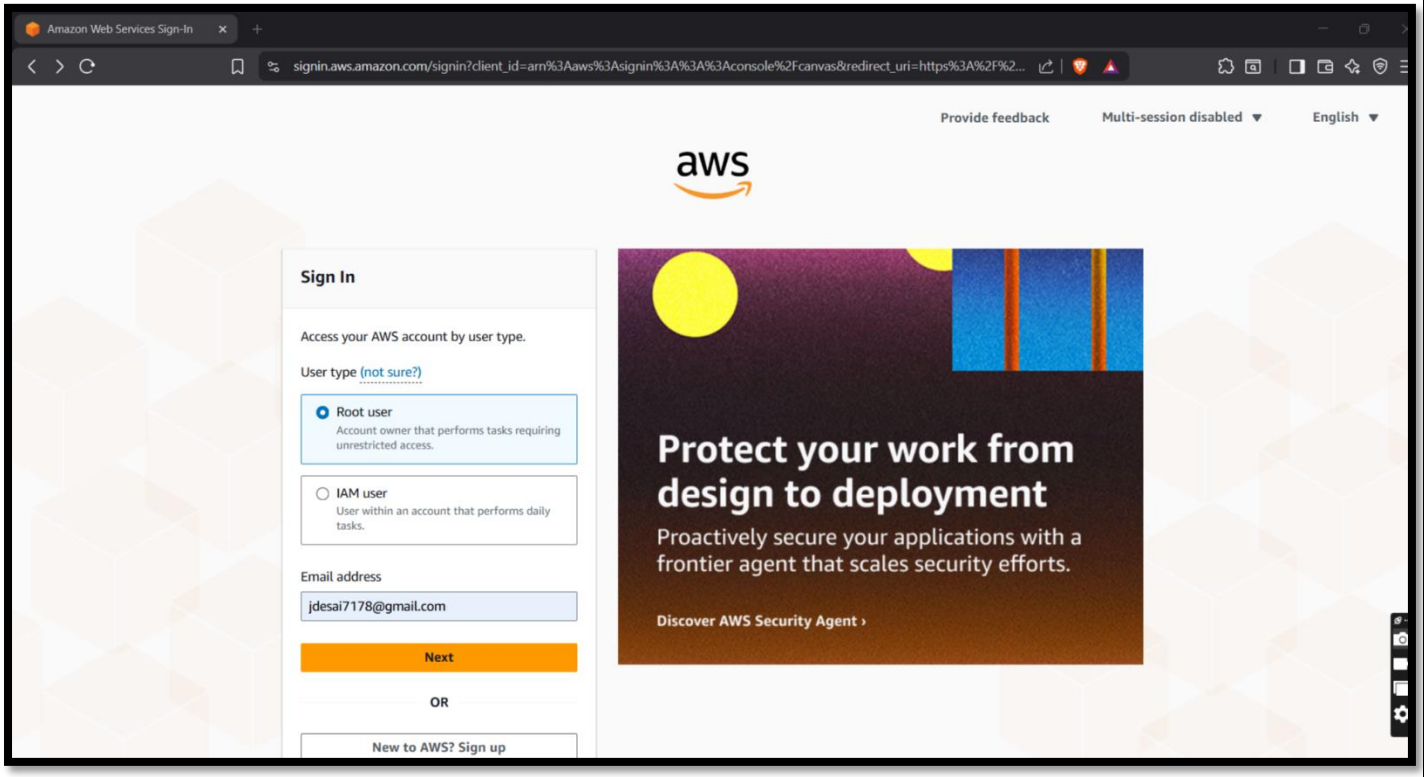
Auto Scaling to avoid over-provisioning

Steps:

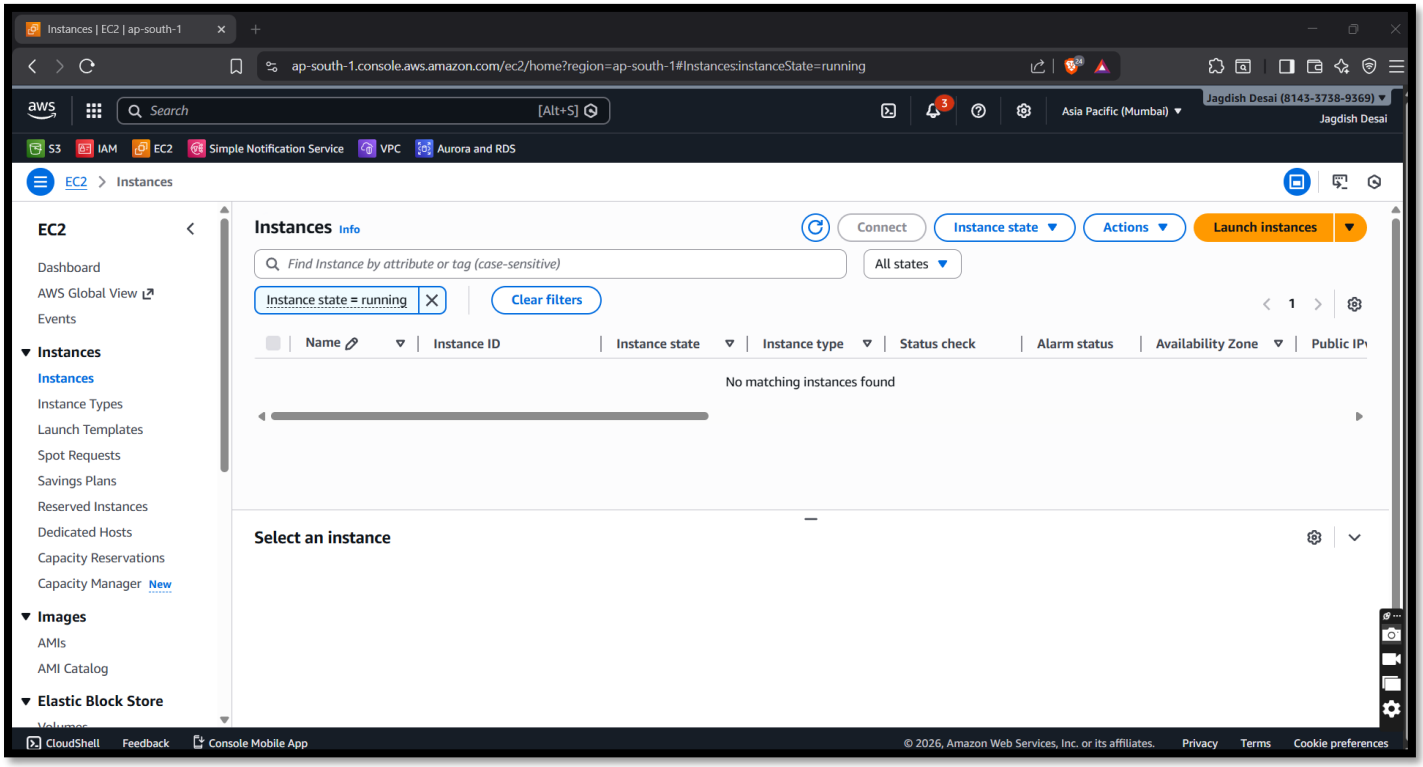
Step 1: Cost Optimization – Implementation Steps

Using Free-Tier Eligible Instances

1. Login to AWS Console

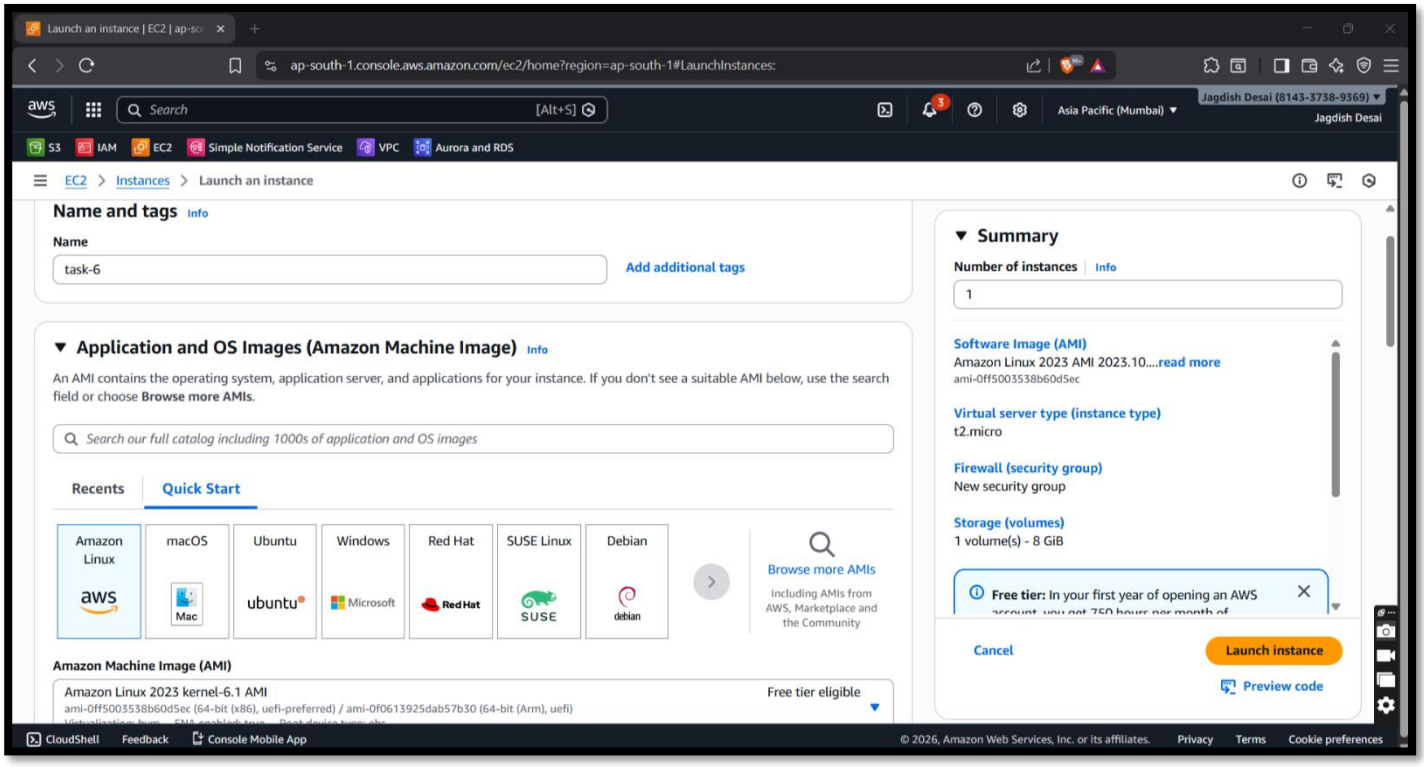


Step 2: Going to EC2 → Launch Instance

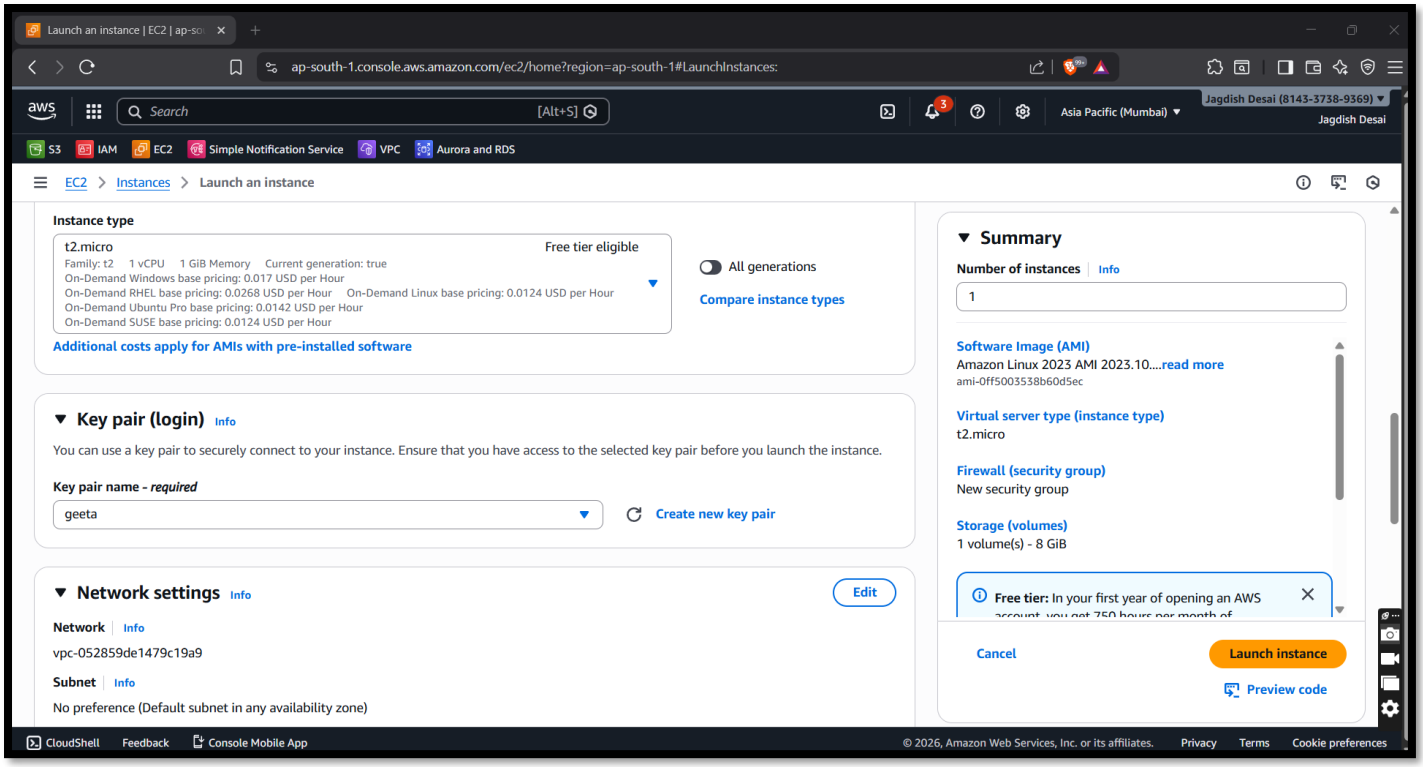


Step 3: Choose Amazon Linux or Ubuntu (because it is free-tier eligible)

and select free tier eligible ami. These AMIs are marked as “Free tier eligible”, AWS provides these OS images at no additional cost, Optimized for AWS environment



Step 4: Select instance type t2.micro / t3.micro [free tier eligible]



1.In Instance Type, choose:

t2.micro OR

t3.micro

2.Instance shows Free tier eligible

- 1 vCPU
- 1 GB RAM
- Suitable for small applications and testing
- Covered under 750 hours/month

3. I Choose Existing Key-Pair

Choose existing key pair OR create new

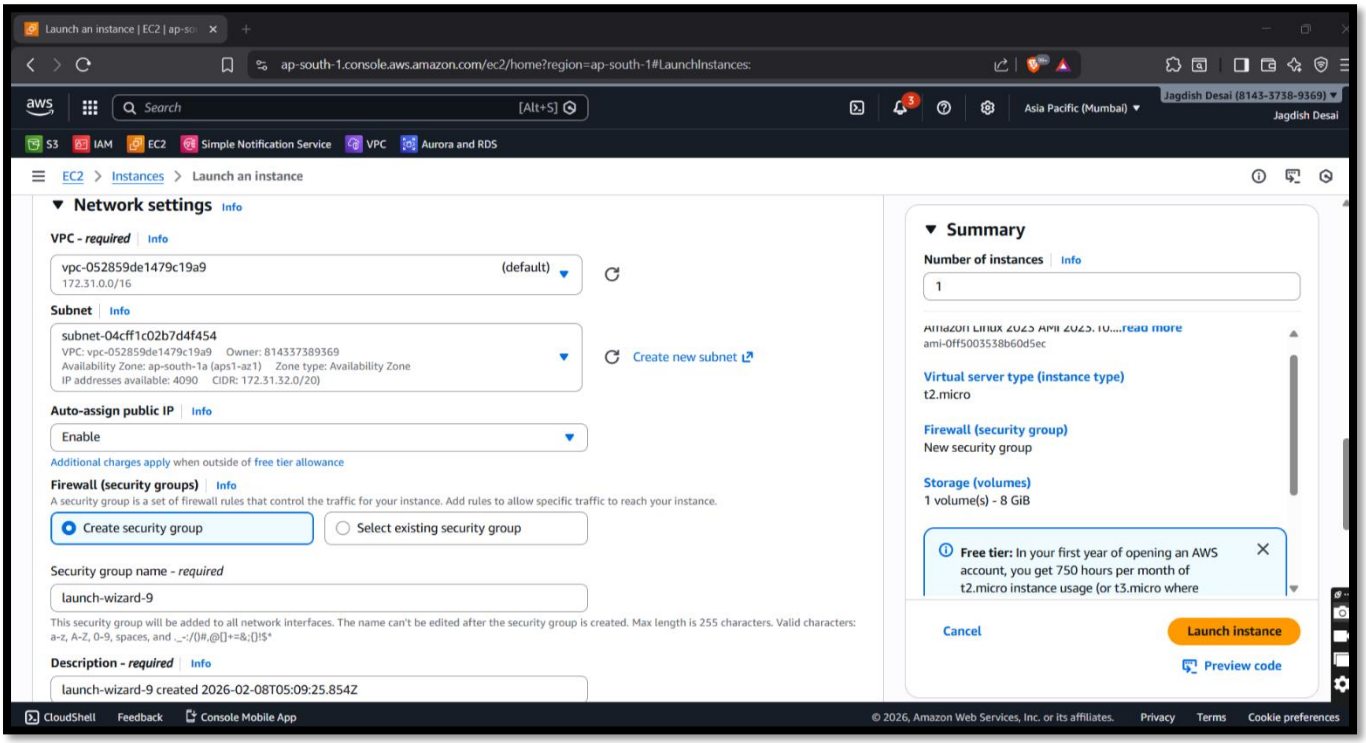
Required for SSH login

No key pair → no server access

Step 5: Configuring Instance Settings

1. Leave default settings:
    - Network: Default VPC
    - Subnet: Auto-assign
2. Enable **Auto-assign Public IP** because

- No extra networking cost
- Easy access using public IP

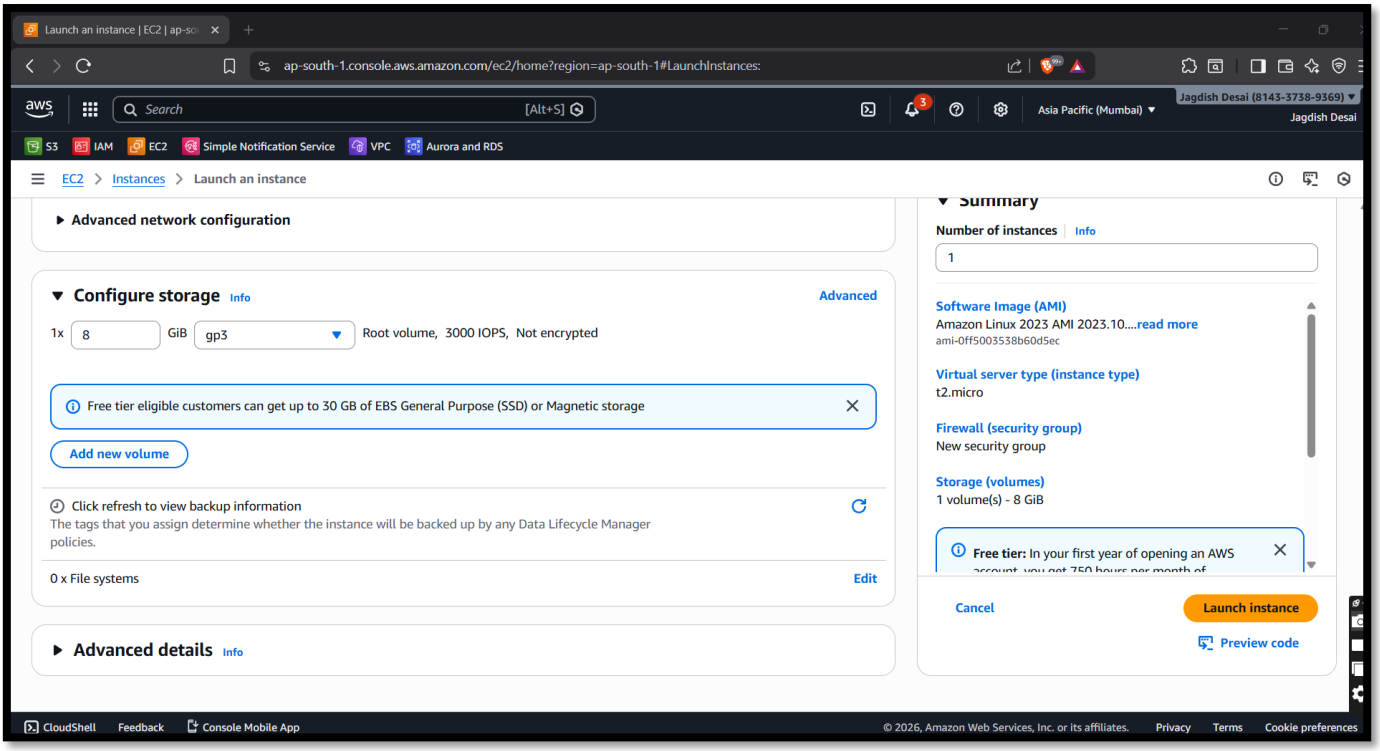


Step 6: Configuring Storage

1. Root volume size: **8 GB (default)**
2. Volume type: **gp2 / gp3**

Because:

- Free tier allows **30 GB** total
- Keeping minimal storage avoids extra charges

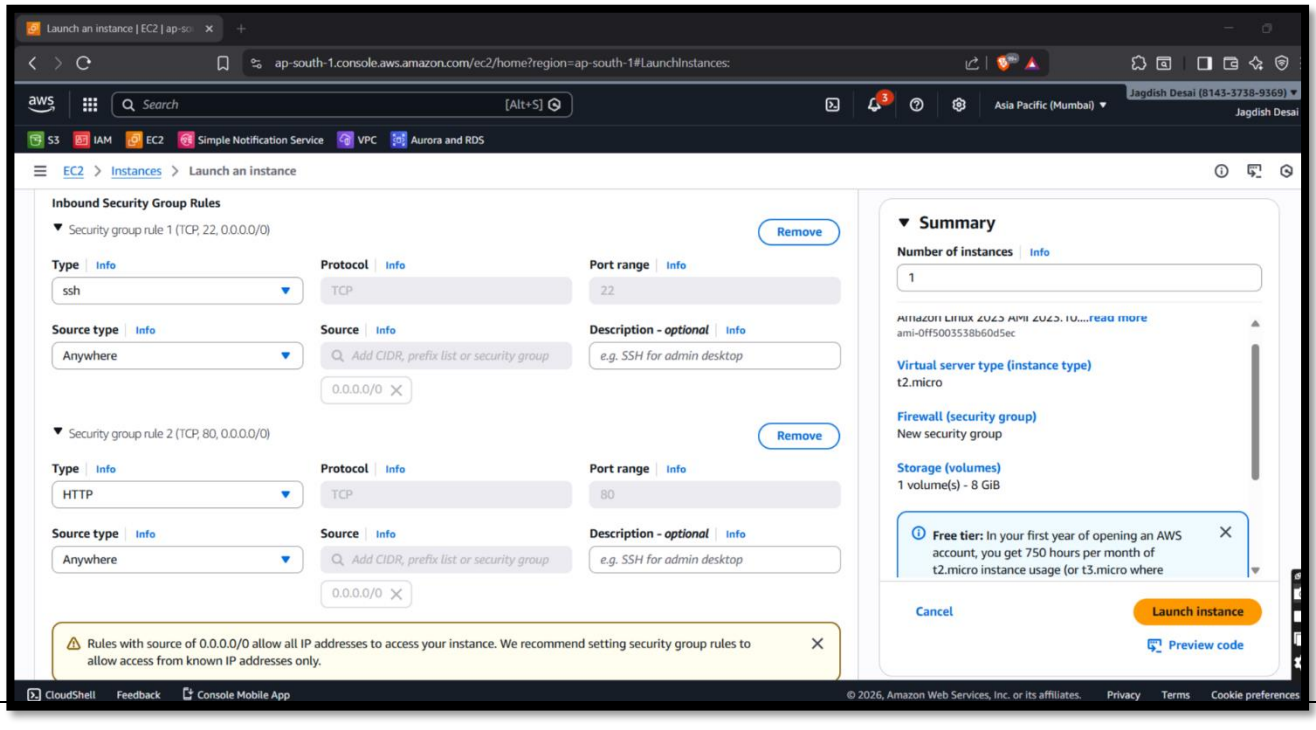


Step 7: Configure Security Group

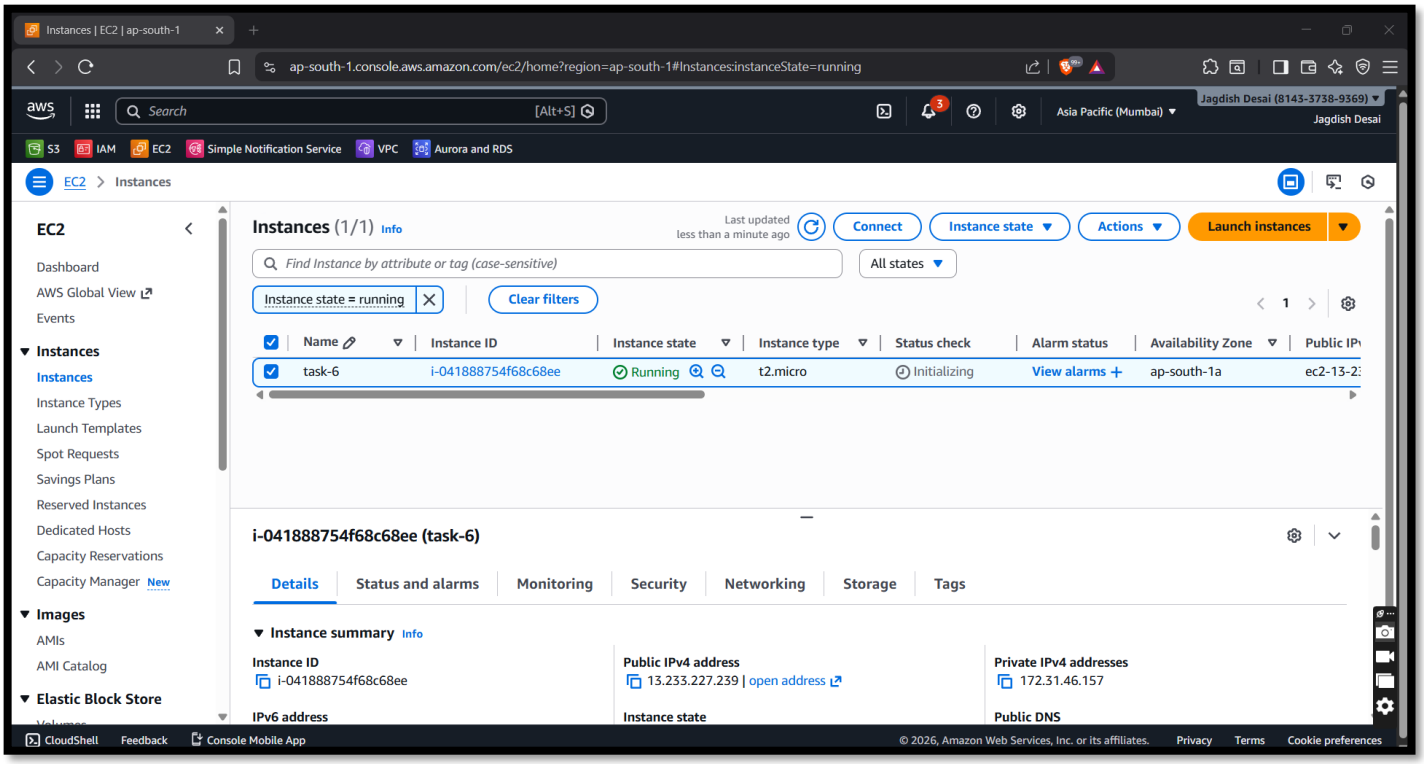
1. Create a new security group
2. Allow:
    - SSH → Port 22 (My IP)
    - HTTP → Port 80 (Anywhere)

Because We give

- Secure access
- Only required ports open (cost & security optimized)



Step 8: Launch Instance



Our Task-6 Instance Successfully Launched with minimal Resourses.

Benefits of Using Minimal Resources

Lower cost

we pay only for what we use, so AWS bills stay low.

Free-tier safe

Helps you stay within free-tier limits and avoid charges.

Less waste

No unused EC2, storage, or IPs running without purpose.

Easy management

Fewer resources = easier monitoring and troubleshooting.

Better scaling

Auto Scaling works efficiently when starting small.

Improved performance control

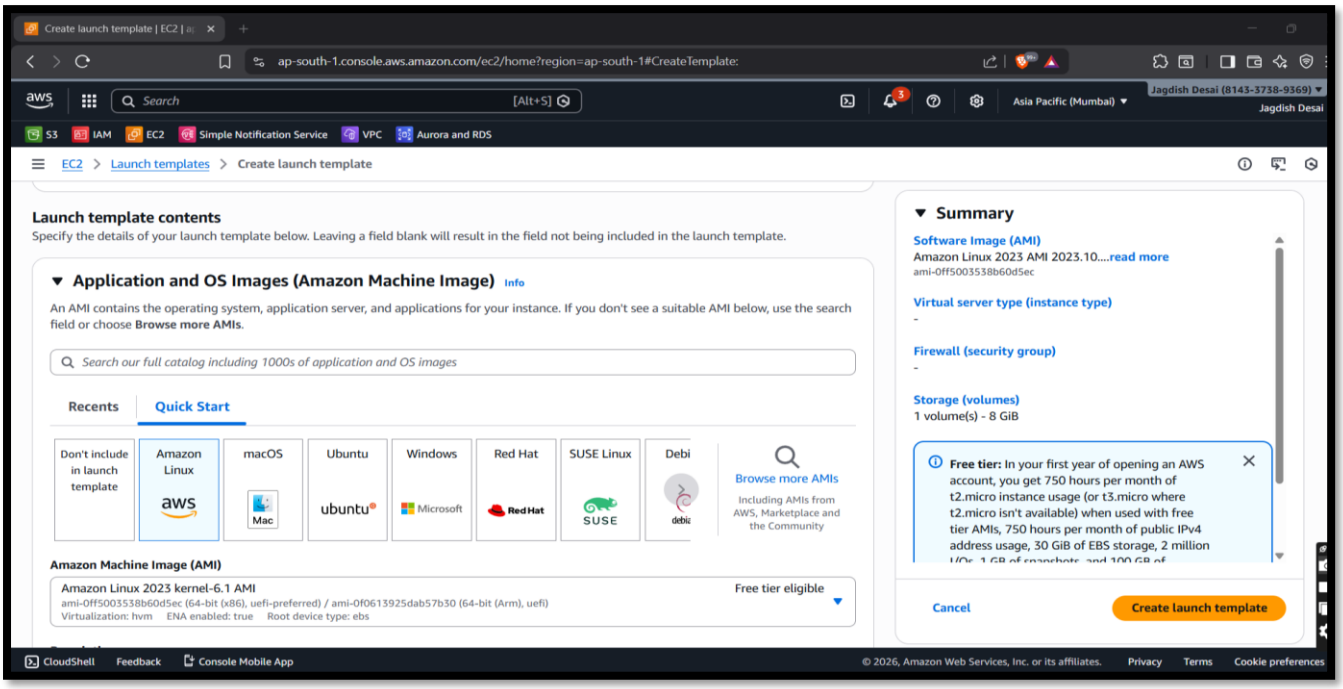
we can clearly see when to scale up or down.

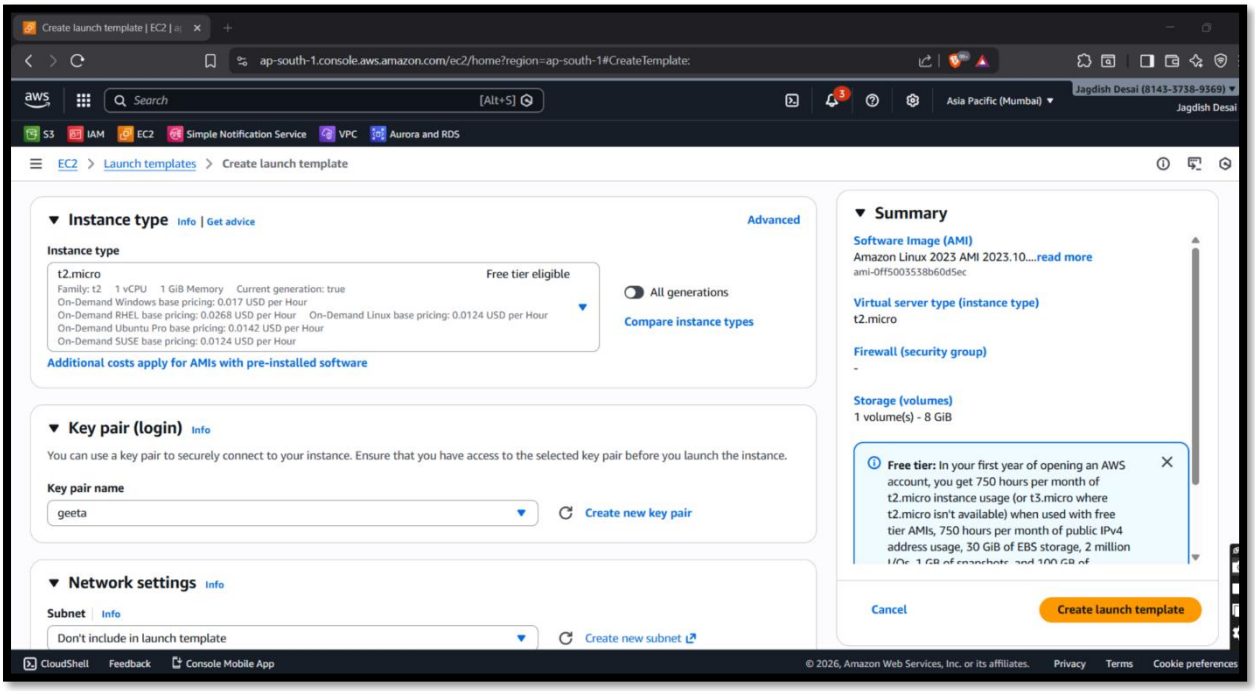
Auto Scaling to avoid over-provisioning

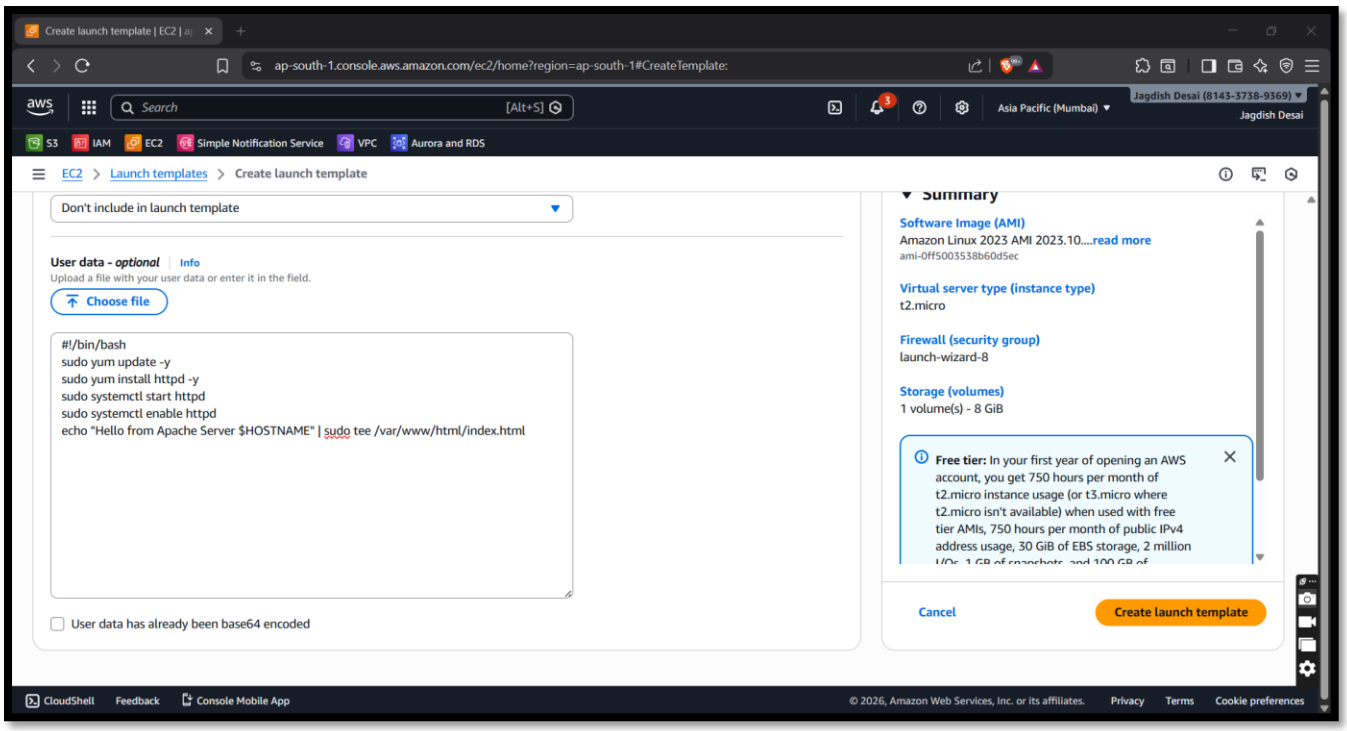
Step 1: Configure Auto Scaling (Avoid Over-Provisioning)

Create Launch Template

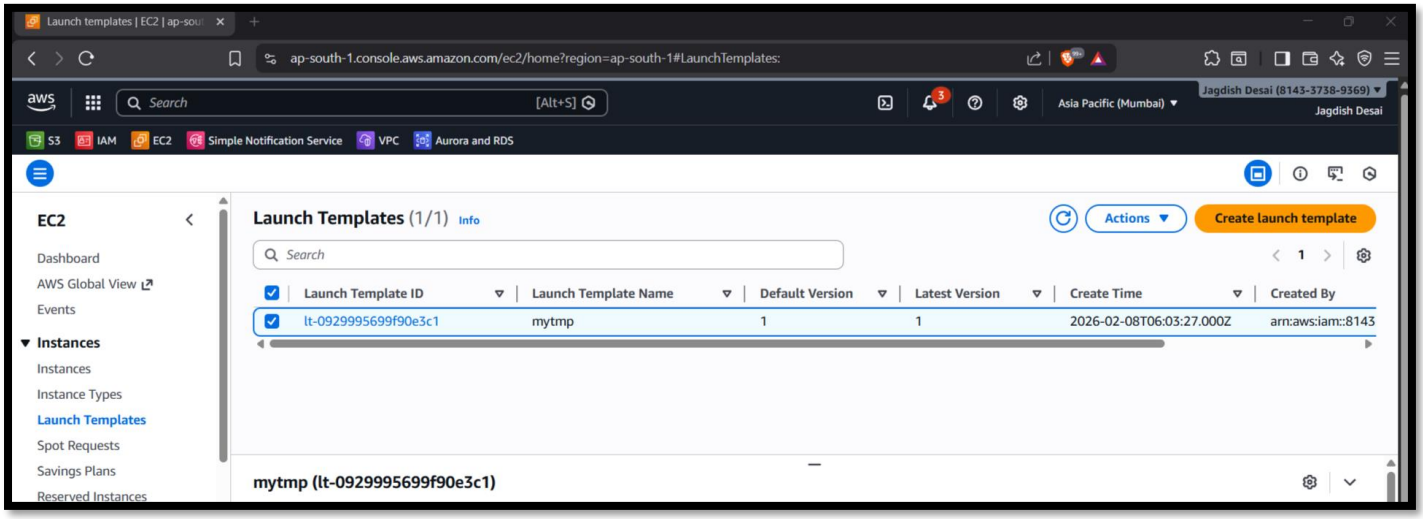
1. Go to EC2 → Launch Templates
2. Create template using:
    - Free-tier instance (t2.micro)
    - AMI with your app installed
3. Save template



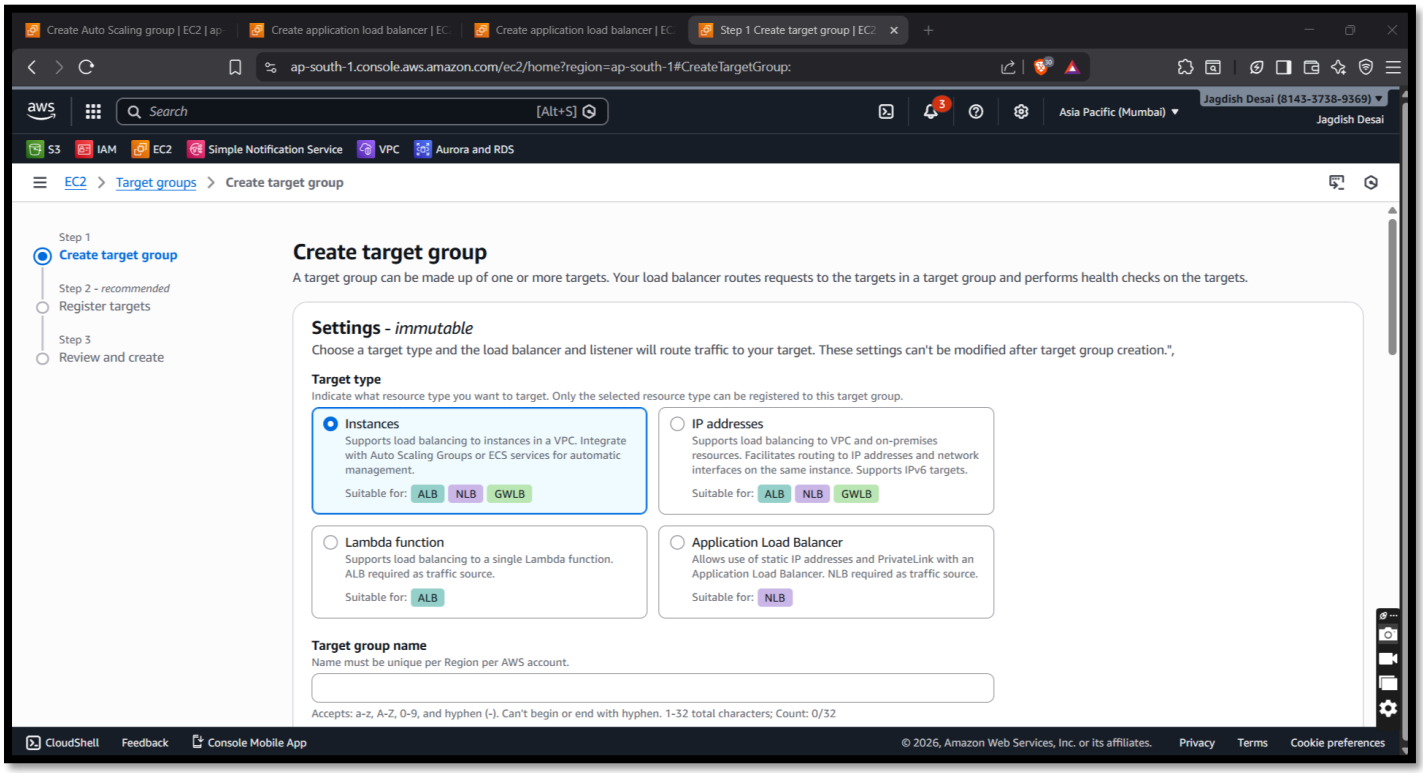




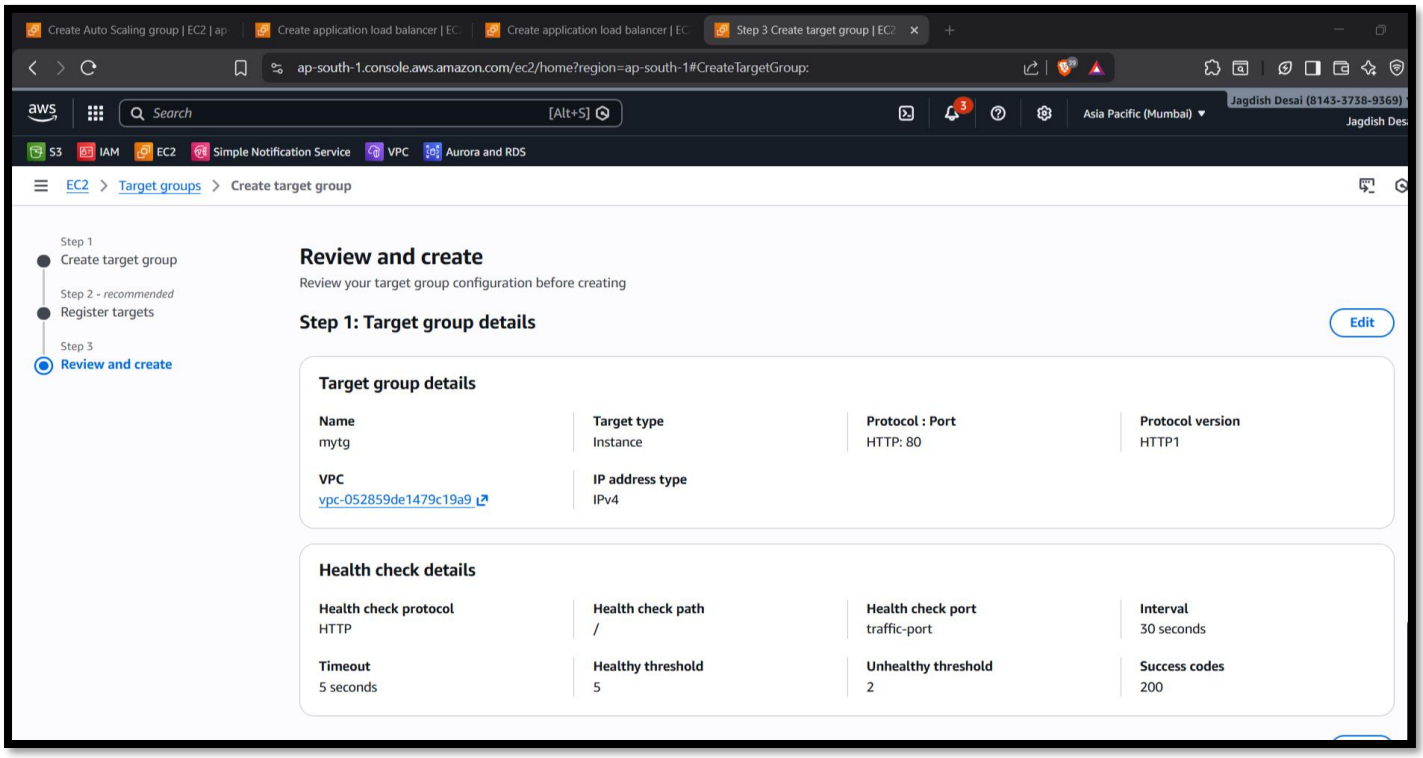
Mytmp Template Successfully Launched:



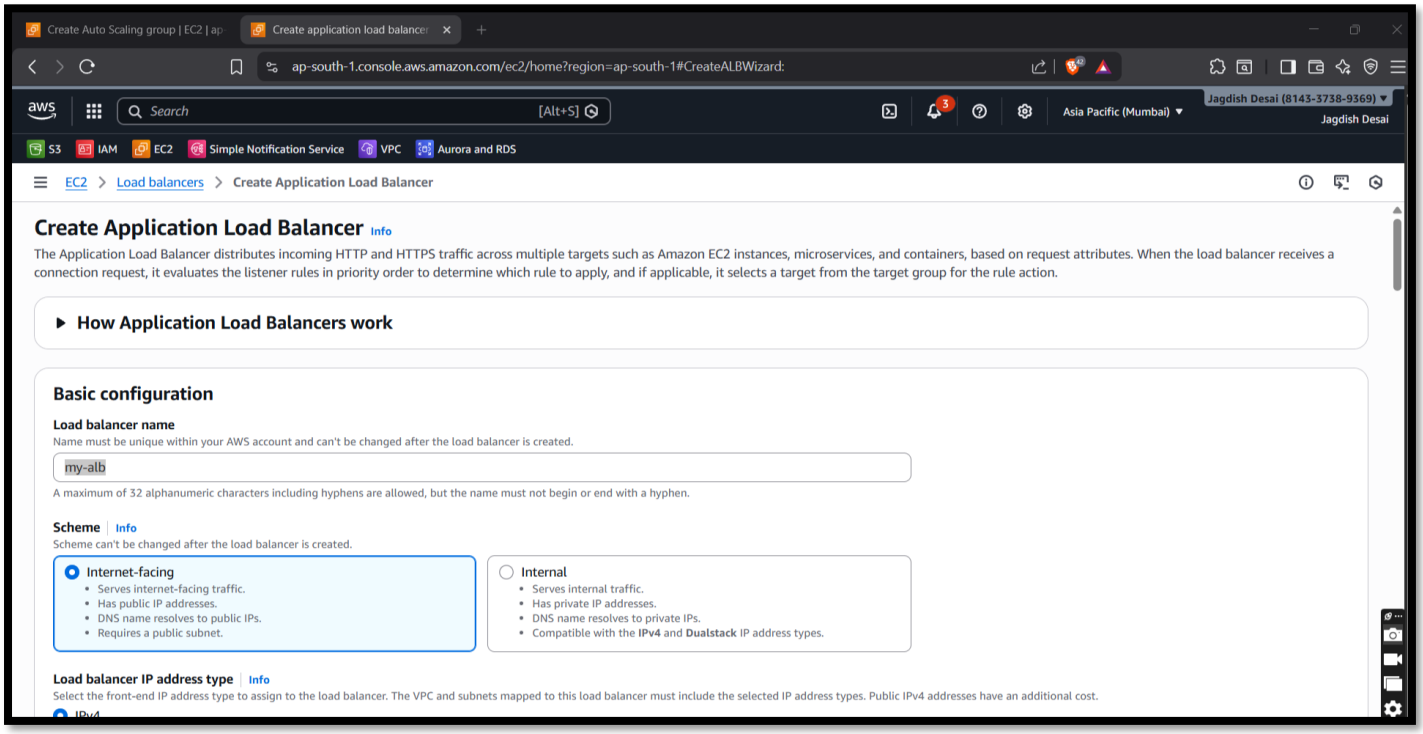
Step 3: Create Target Groups



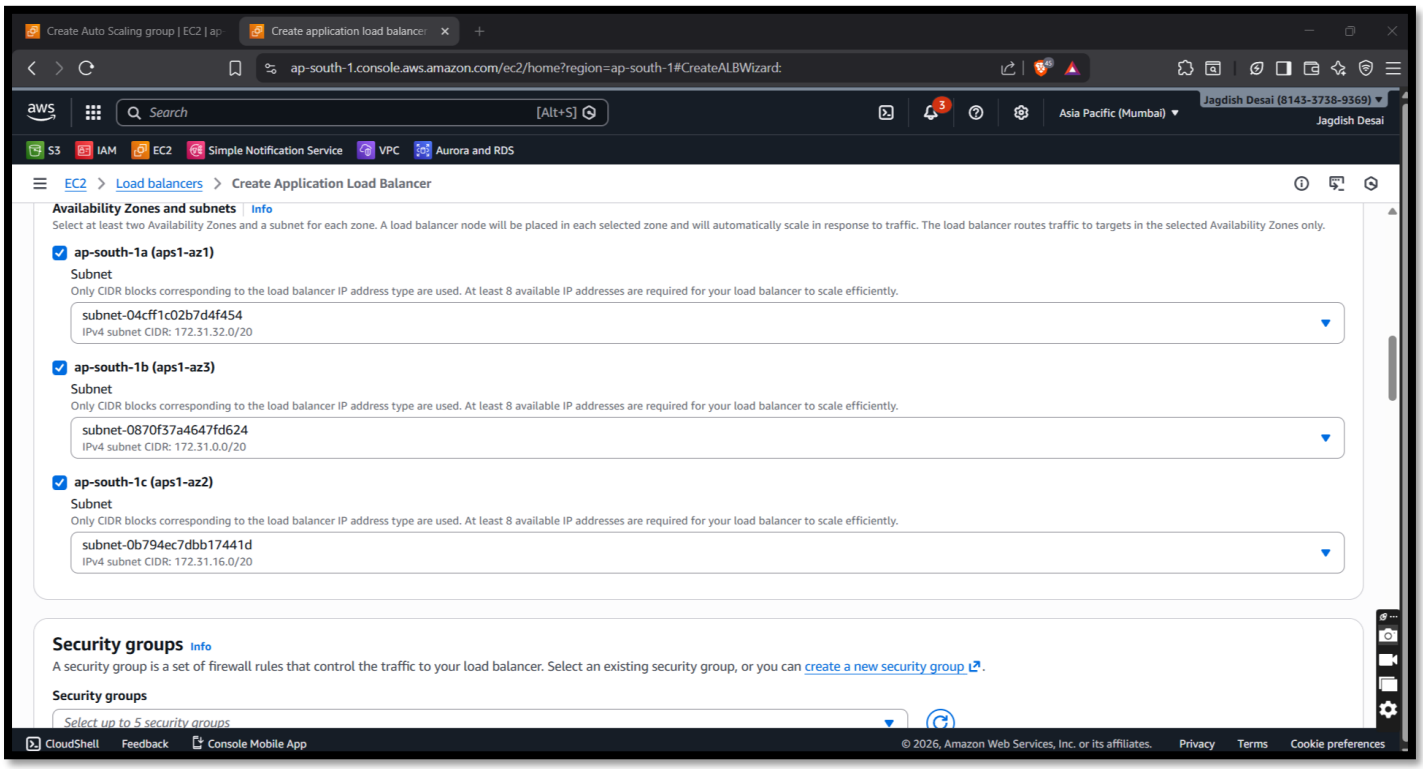
Review Create Target group:



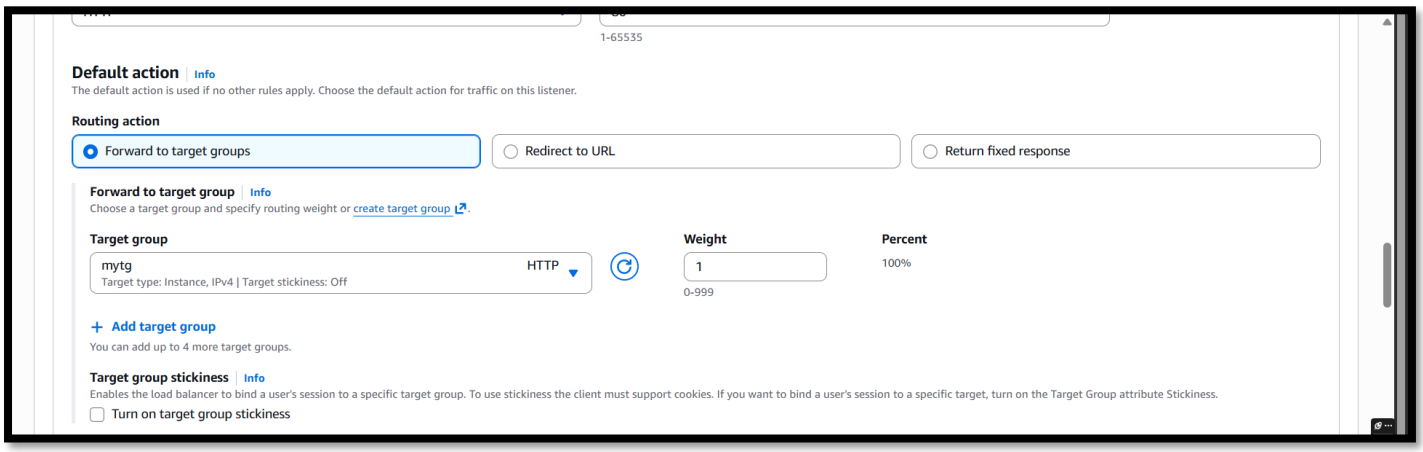
Step 4: Create Load Balancer



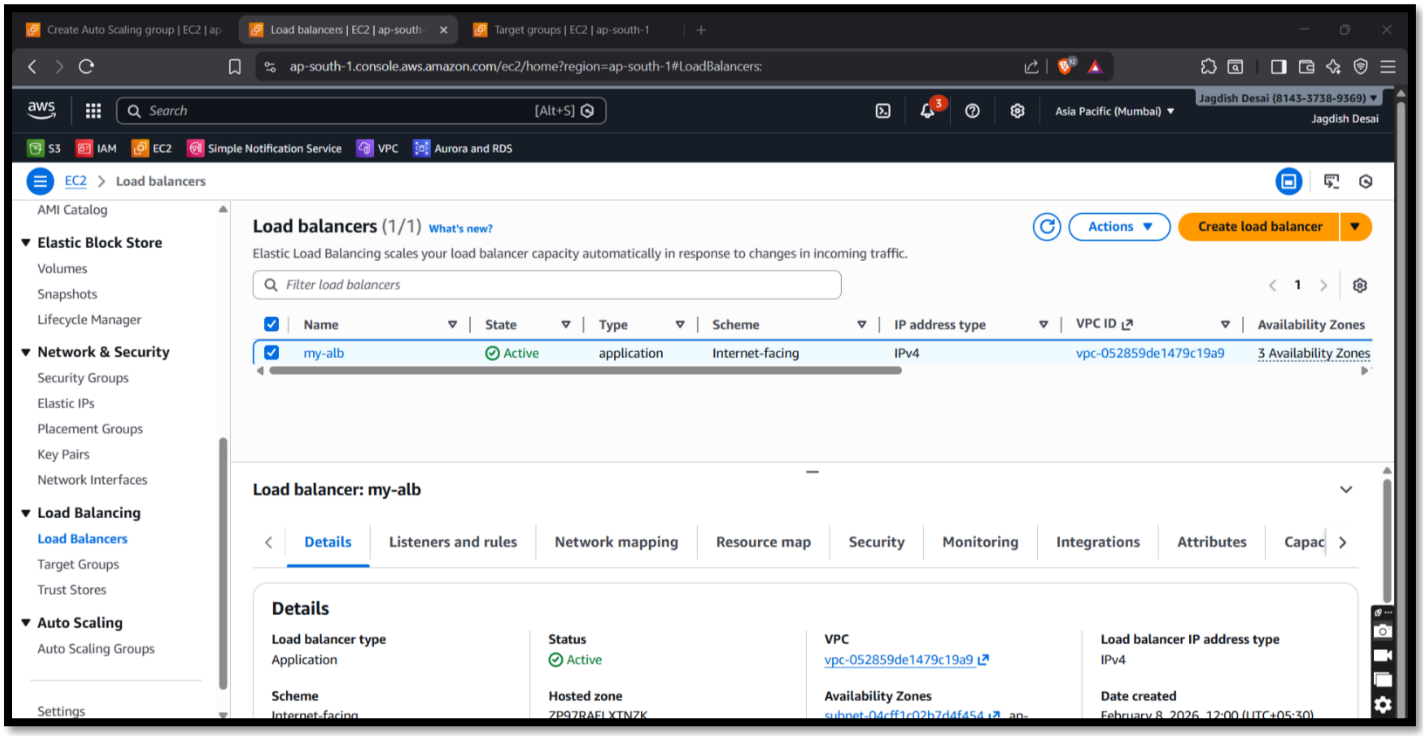
Select atleast 2 availability zones



Attach Target Group:

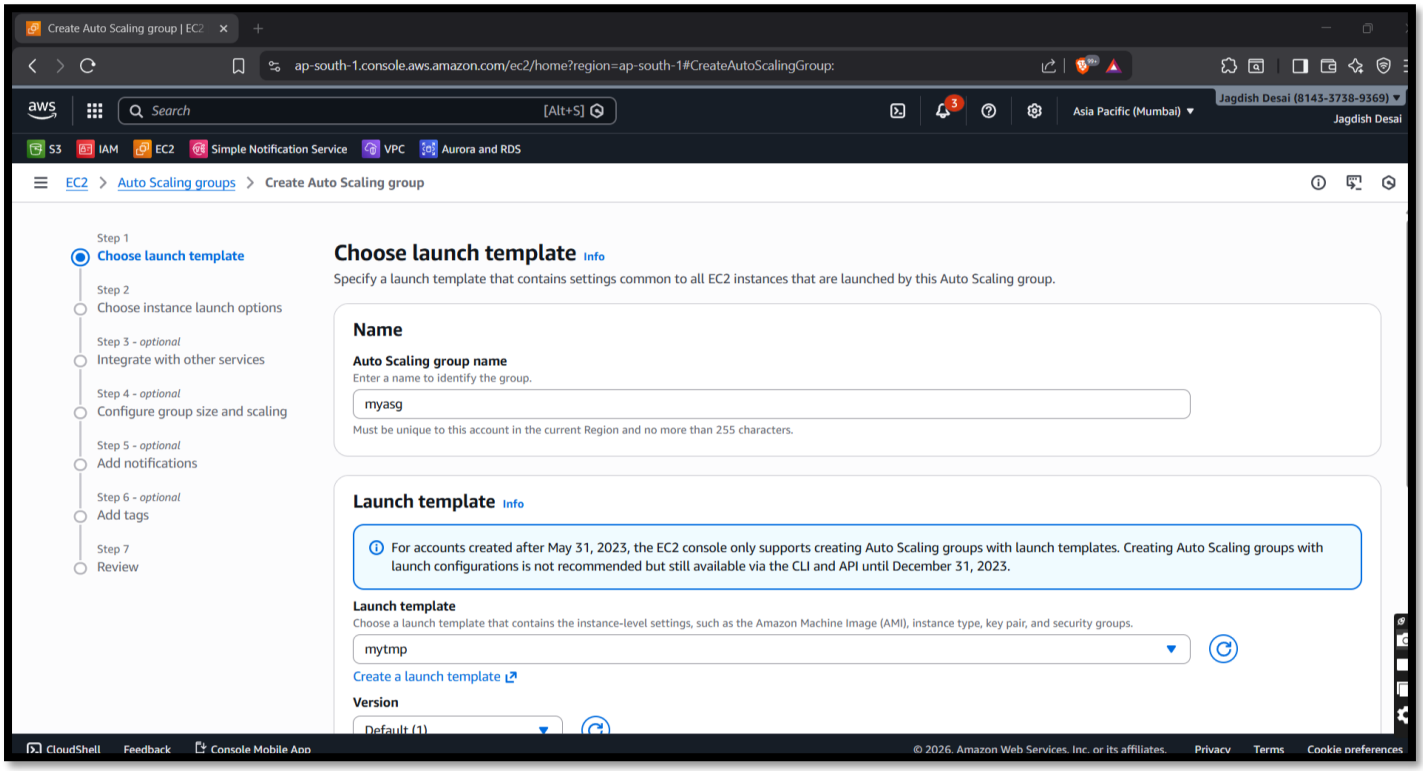


Successfully created load balancer

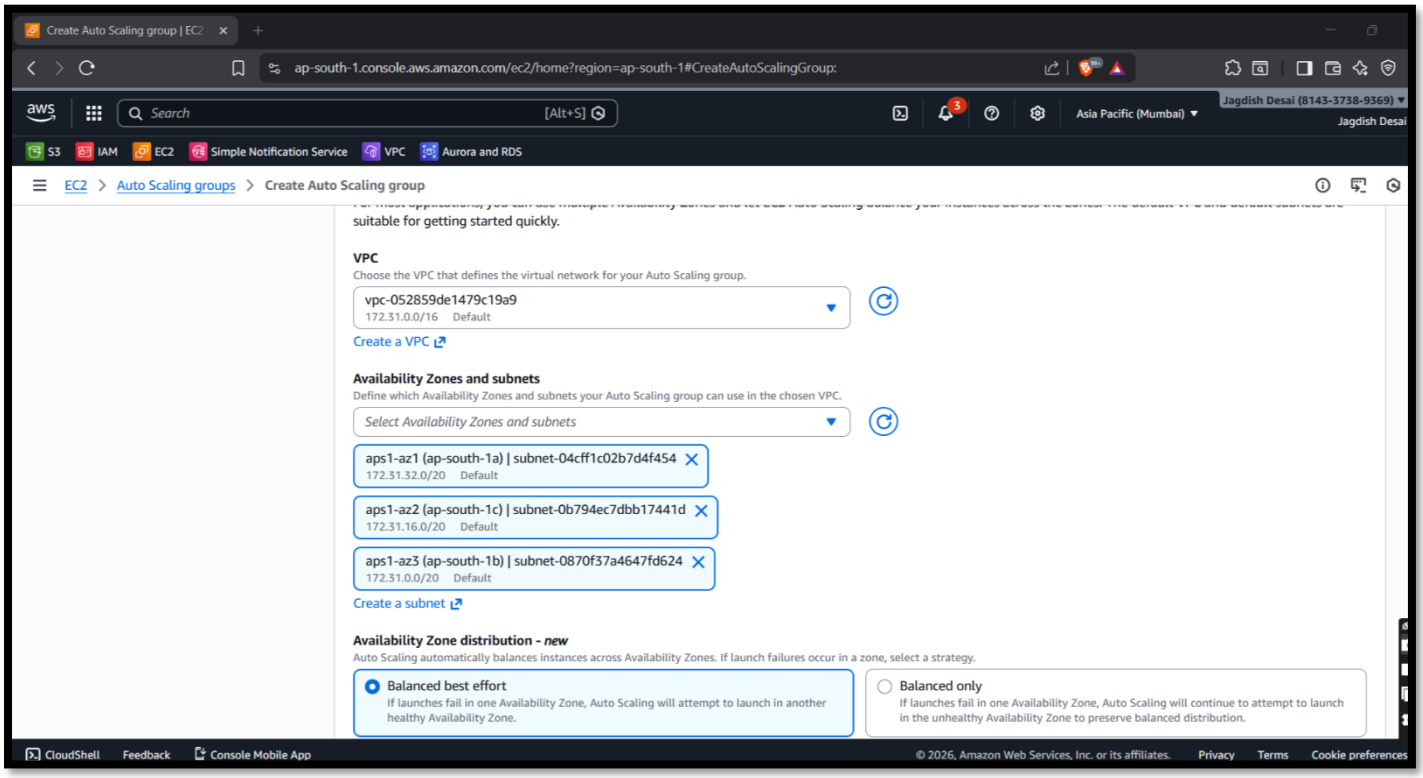


Step 5: Creating Auto Scaling Group

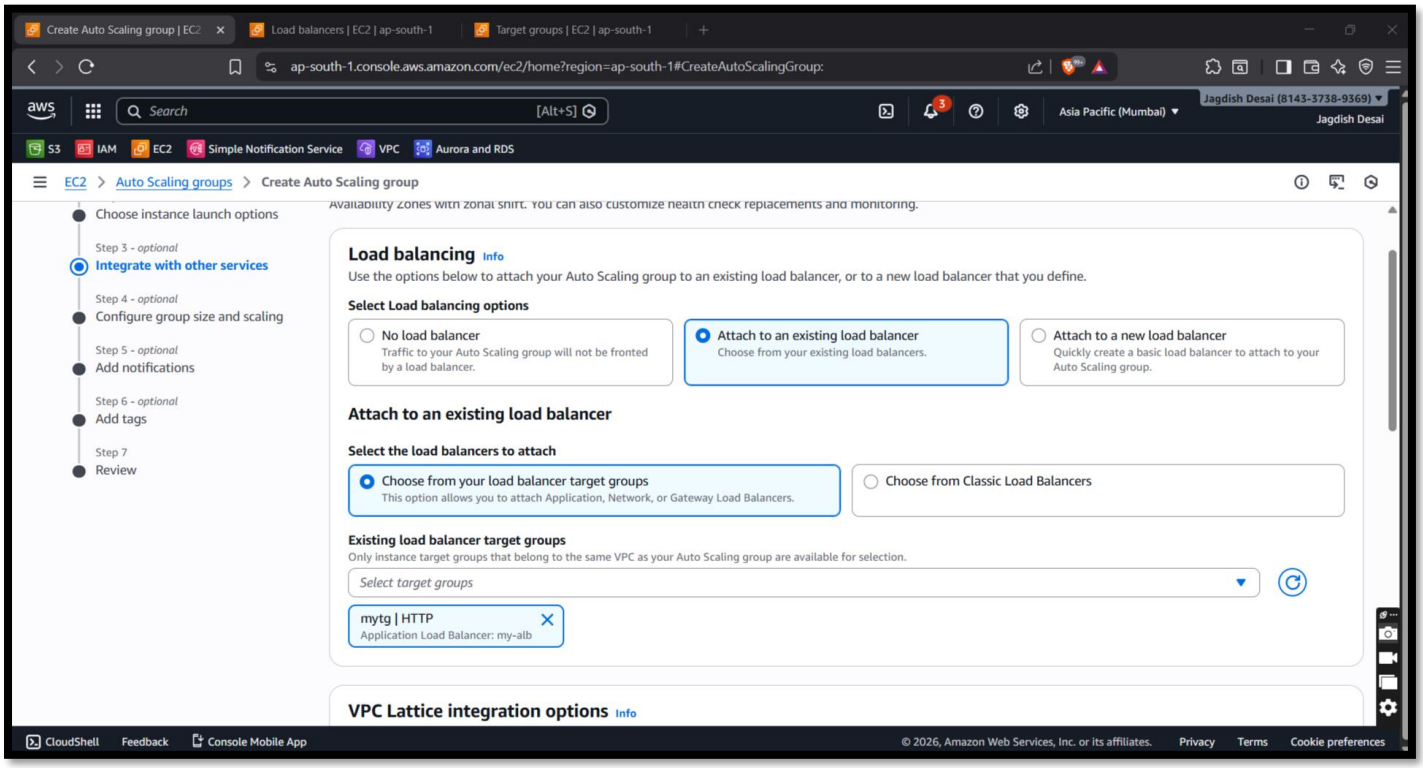
Select launch template



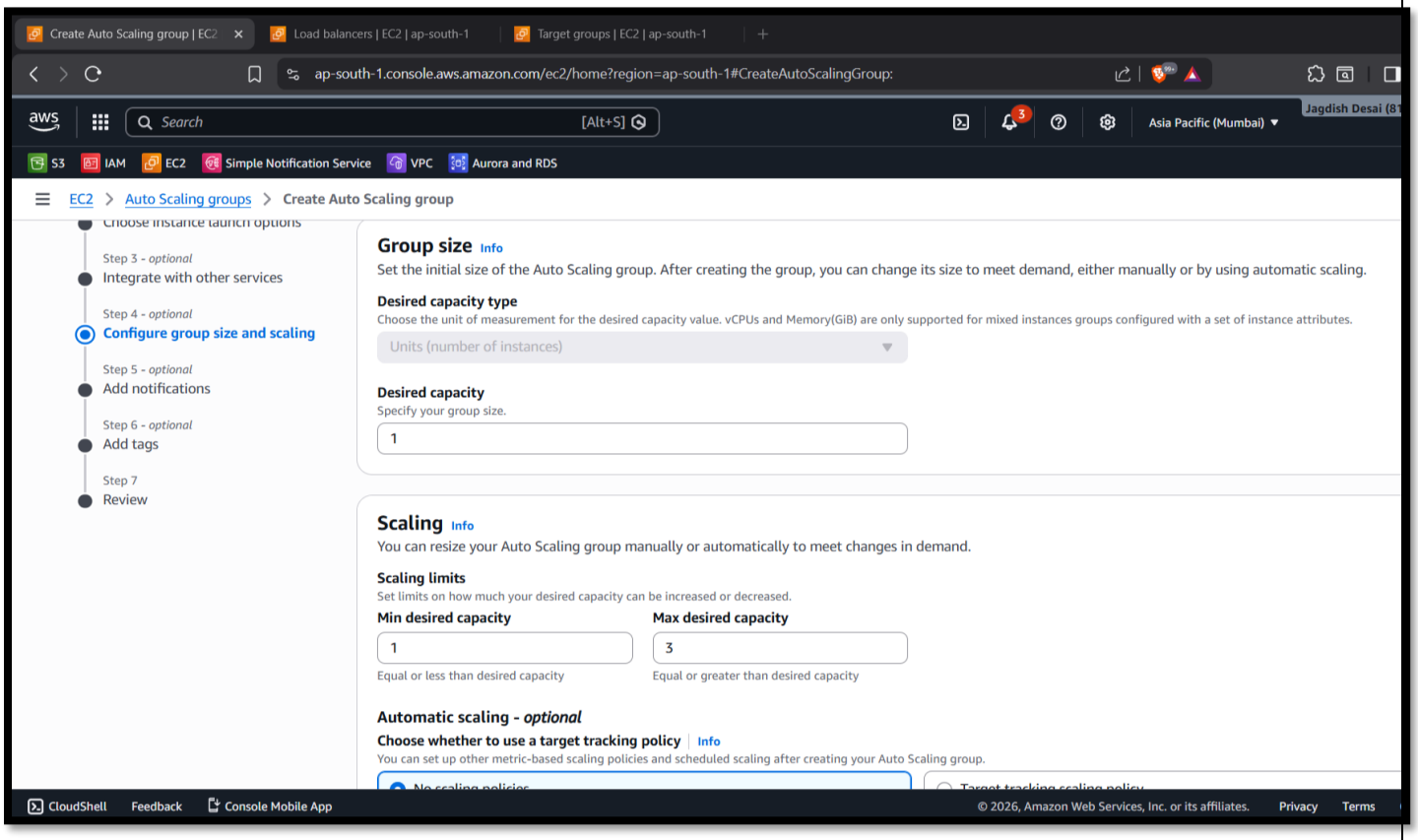
Choose VPC and subnets



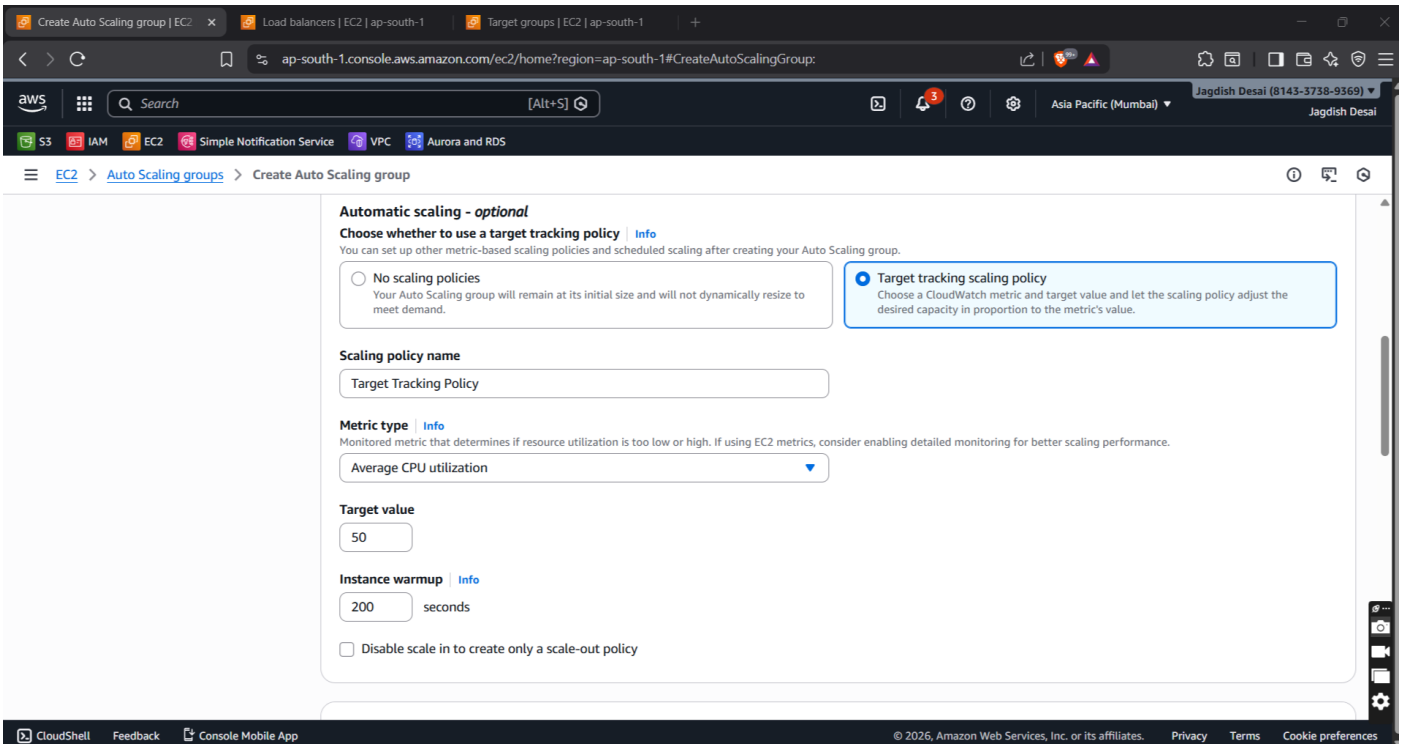
Integrate with other services –



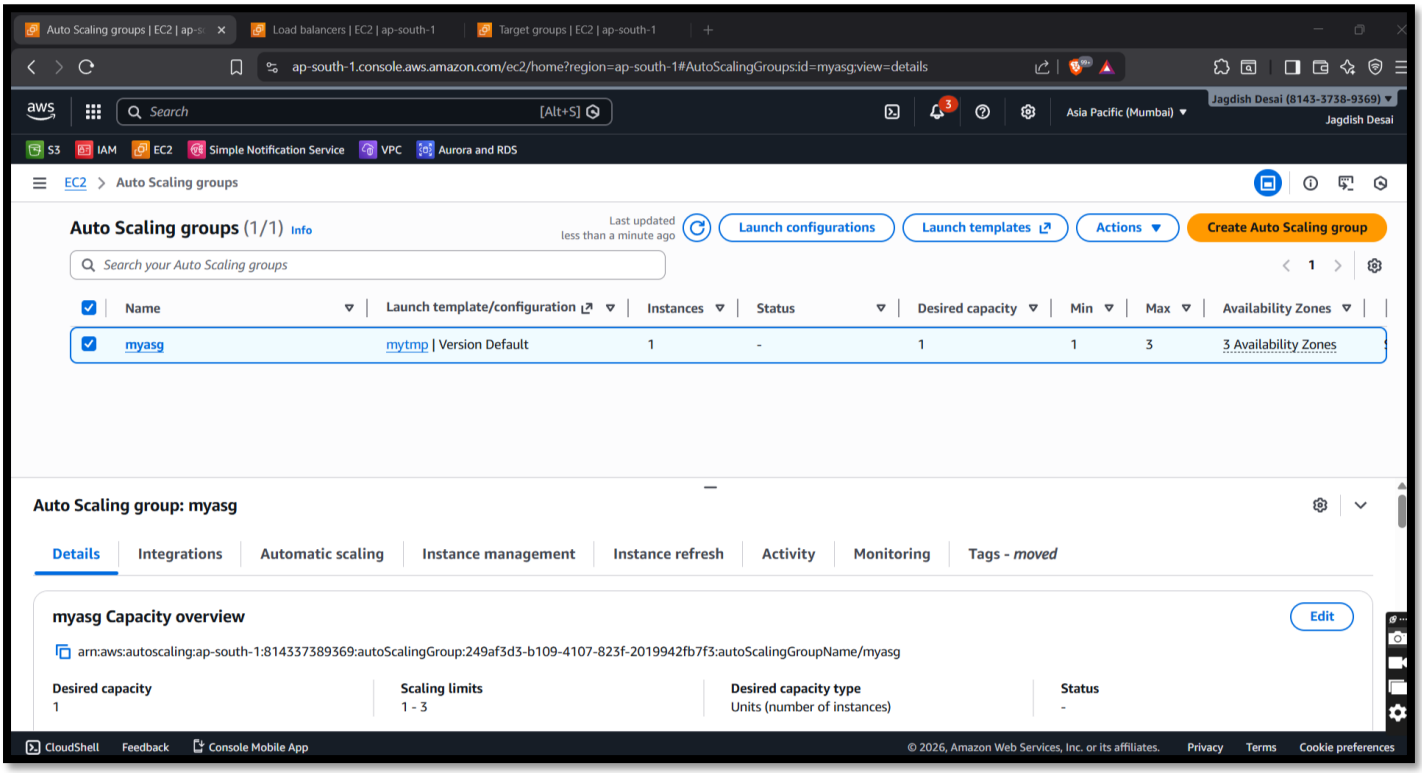
Configuring group size:



Choose whether to use a target tracking policy

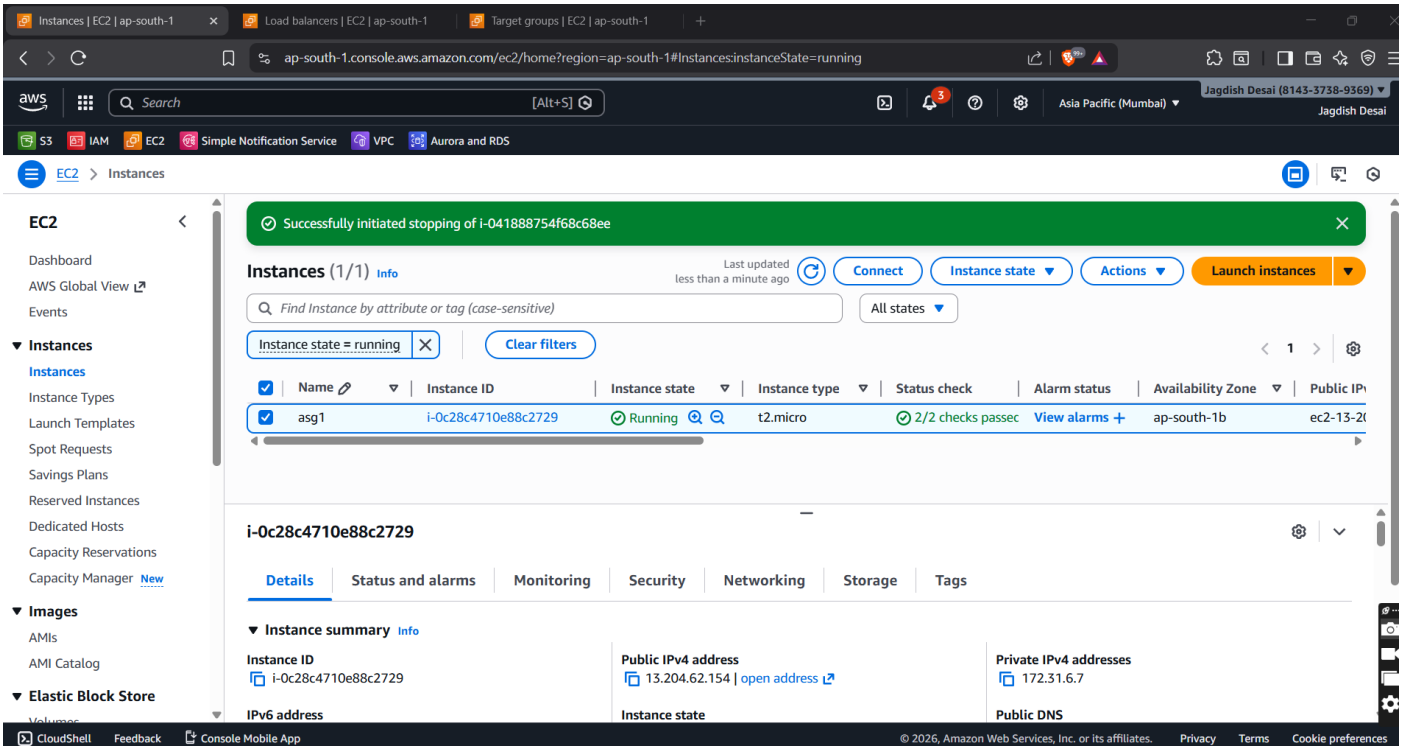


Successfully created Auto Scaling group: myasg



- **Users** → Send requests to **ALB**
- **ALB** → Distributes requests to **Target Group**
- **Target Group** → Monitors **EC2 instances** for health
- **ASG** → Automatically scales EC2 based on load (CPU, traffic)

First Instance will be created by auto scaling groups-asg 1

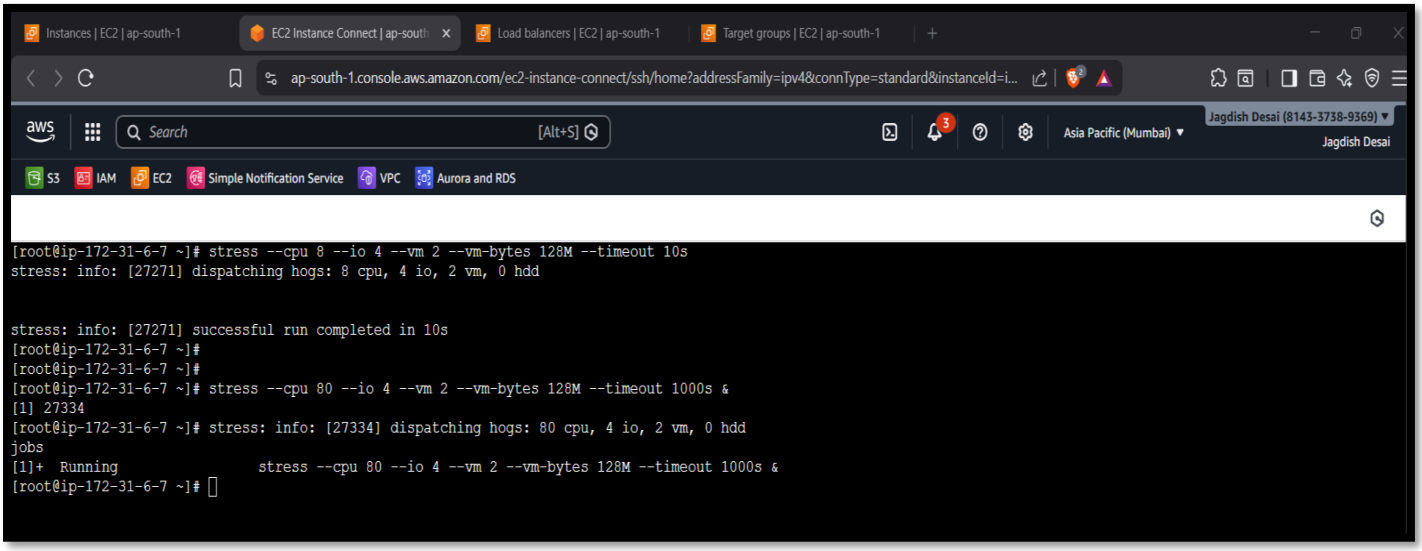


Increasing Load On that instance for for check working using stress command

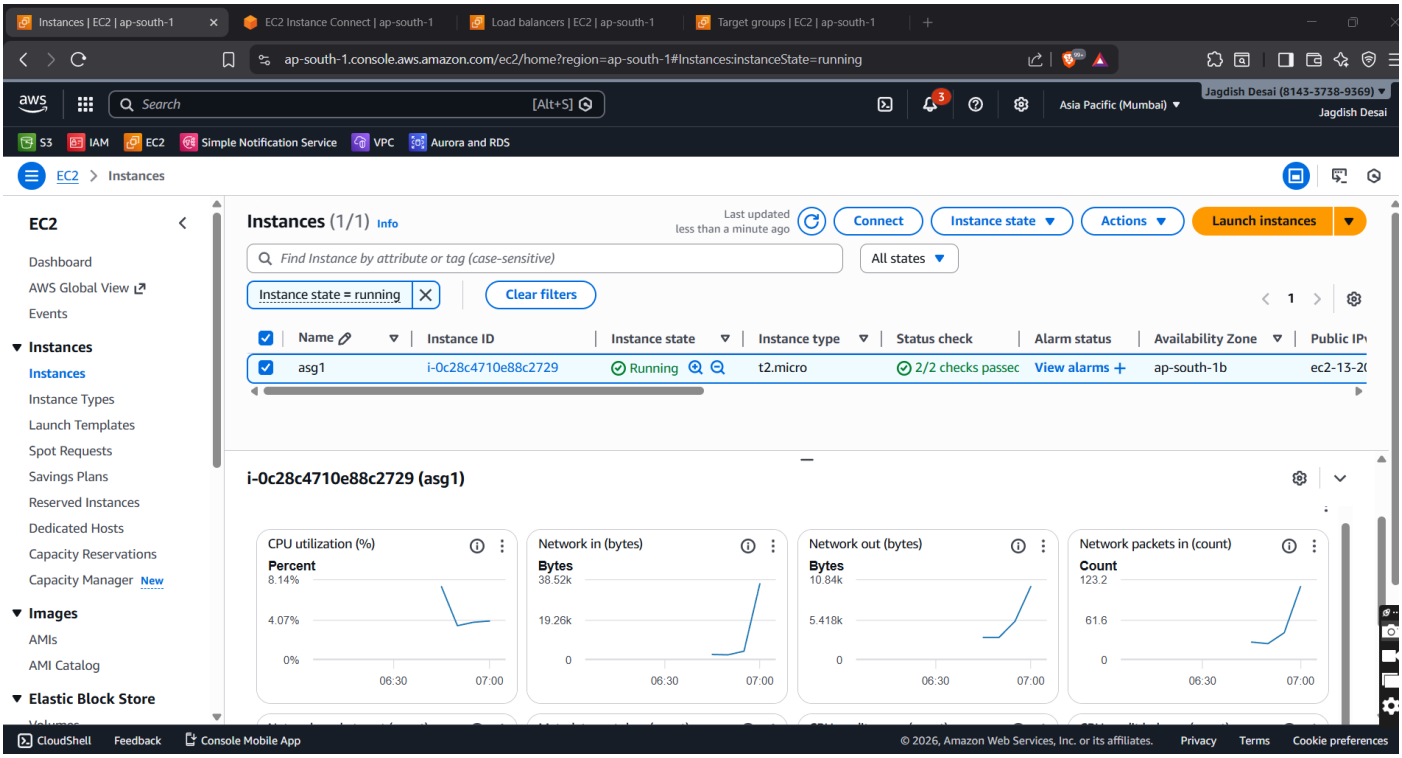
[root@ip-172-31-6-7 ~]# yum install stress

Increasing stress using that command

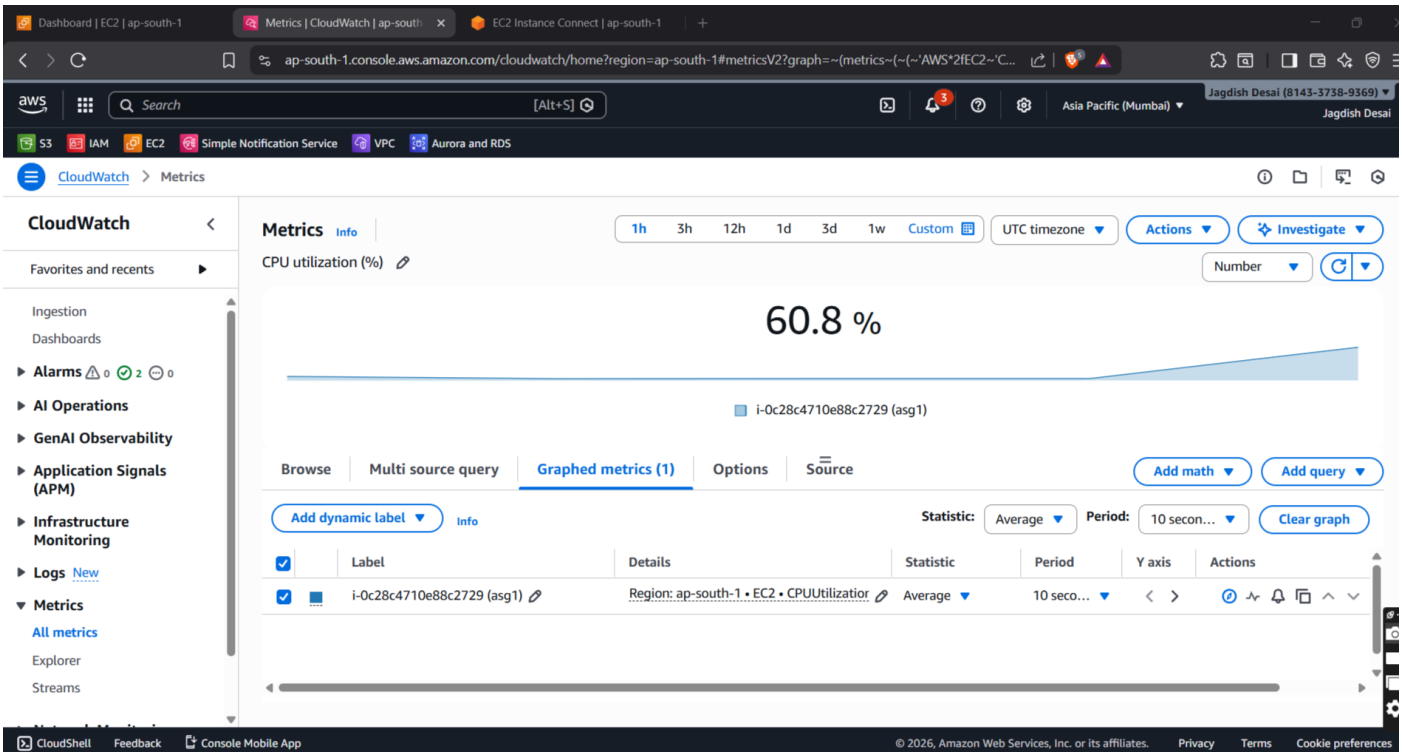
stress --cpu 80 --io 4 --vm 2 --vm-bytes 128M --timeout 1000s&



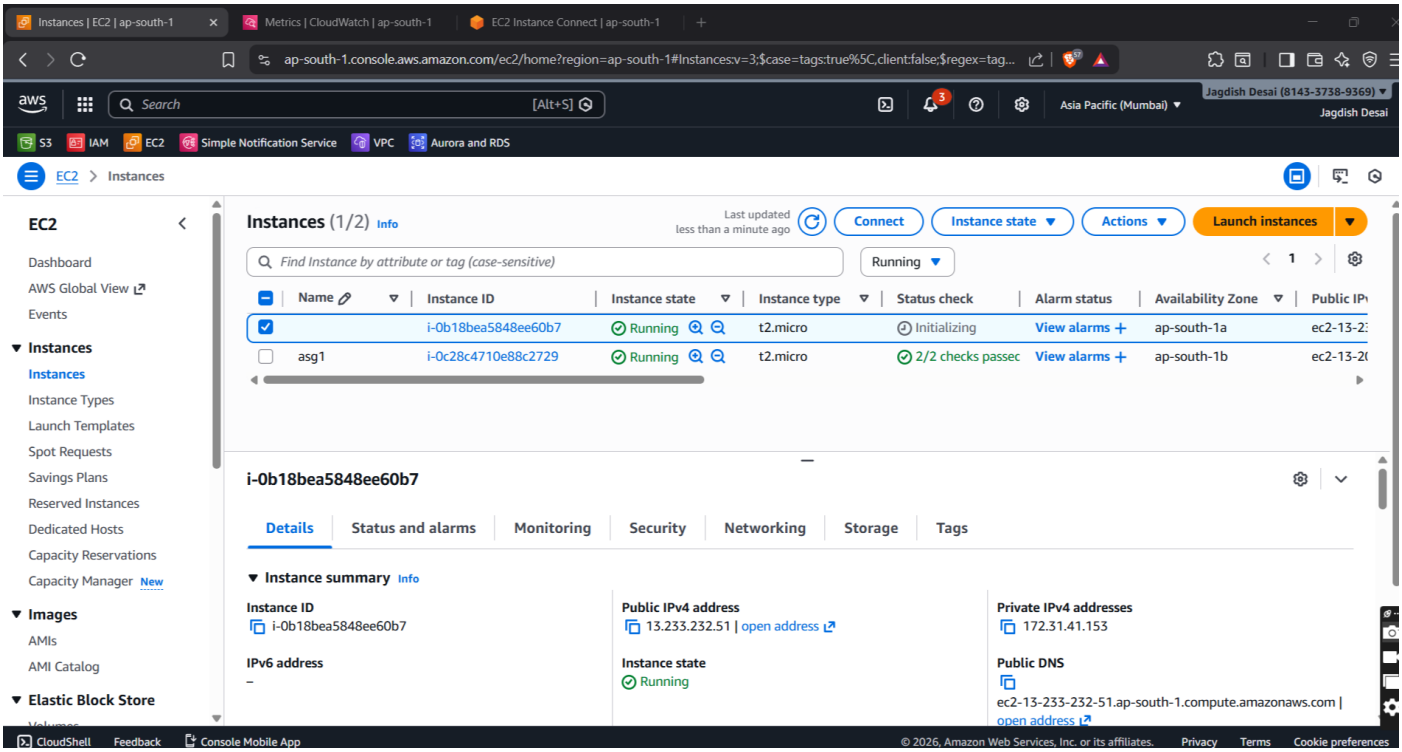
Then monitor cpu load.



Load Goes Above 50% so check instances because new instance will be creates



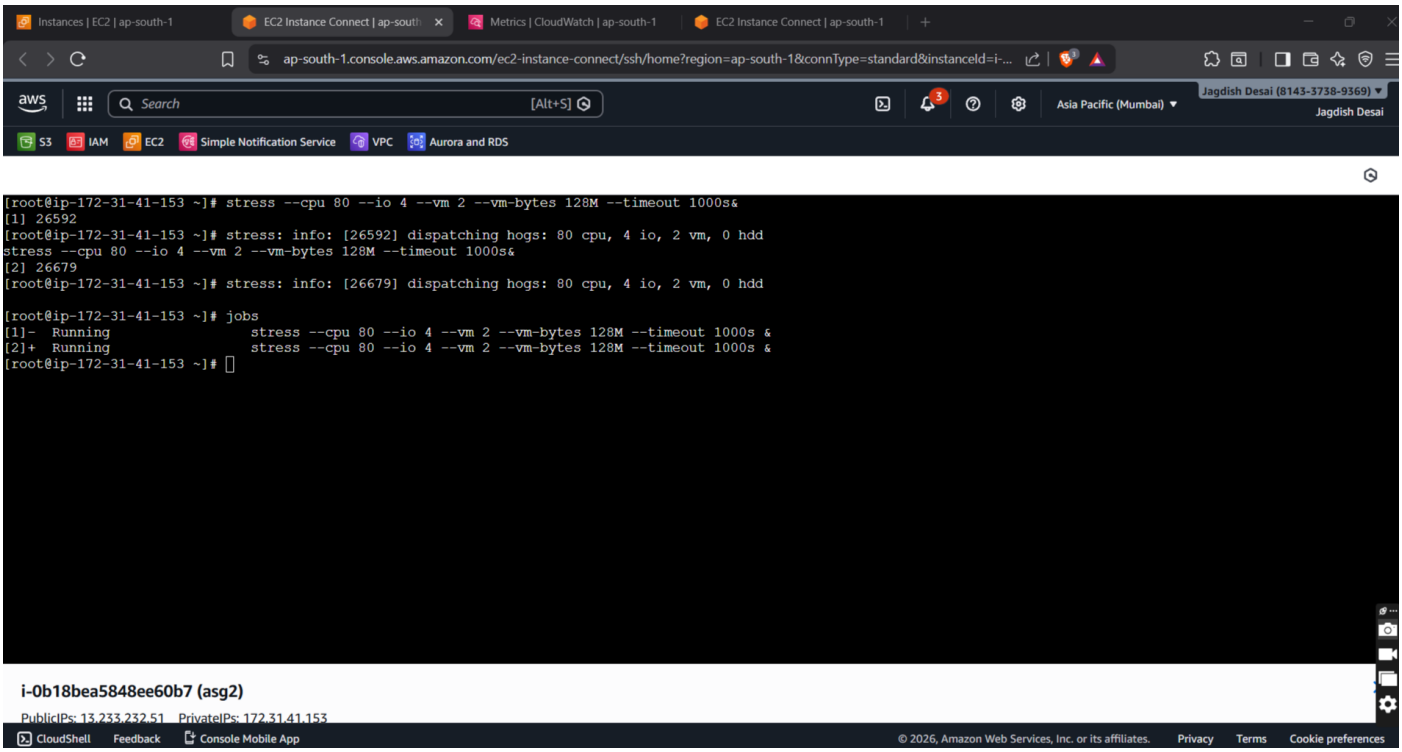
Checking instances



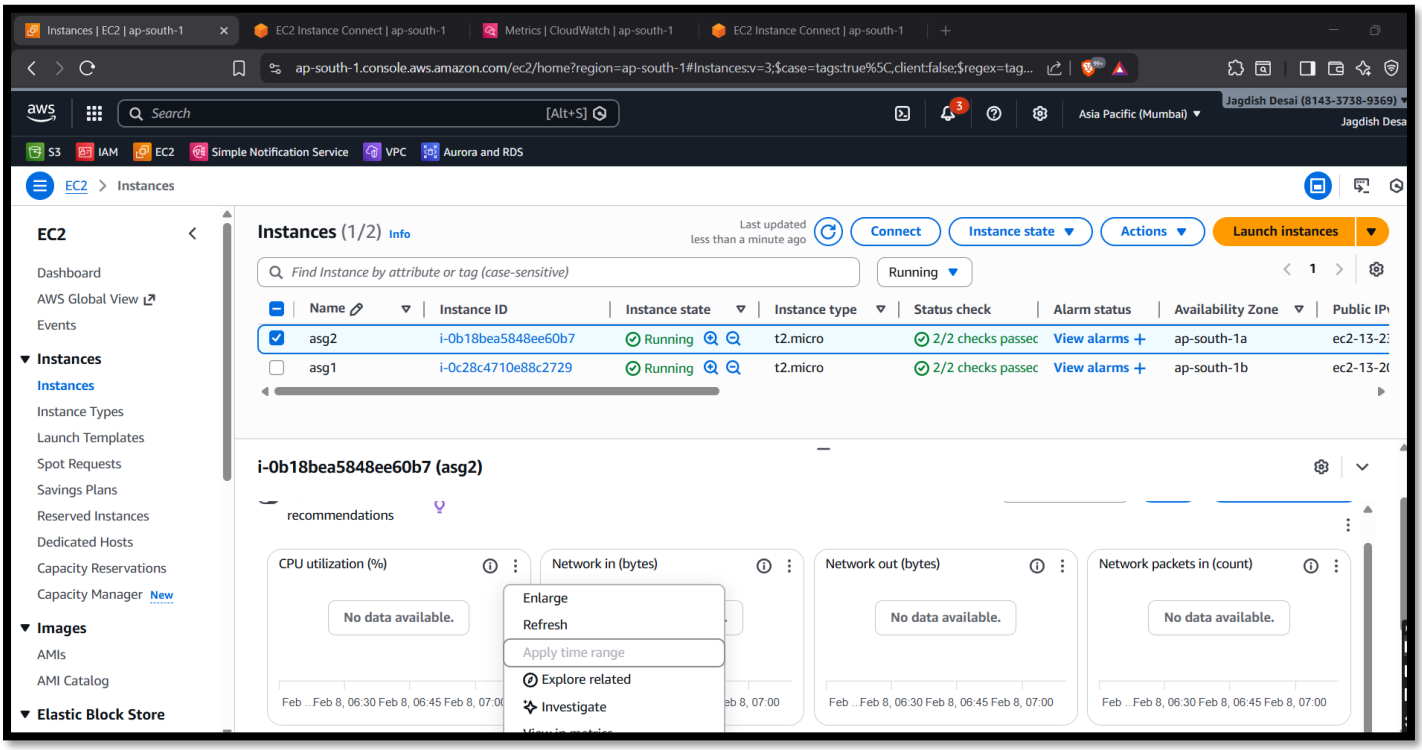
New Instance will Be automatically created after load will increases.

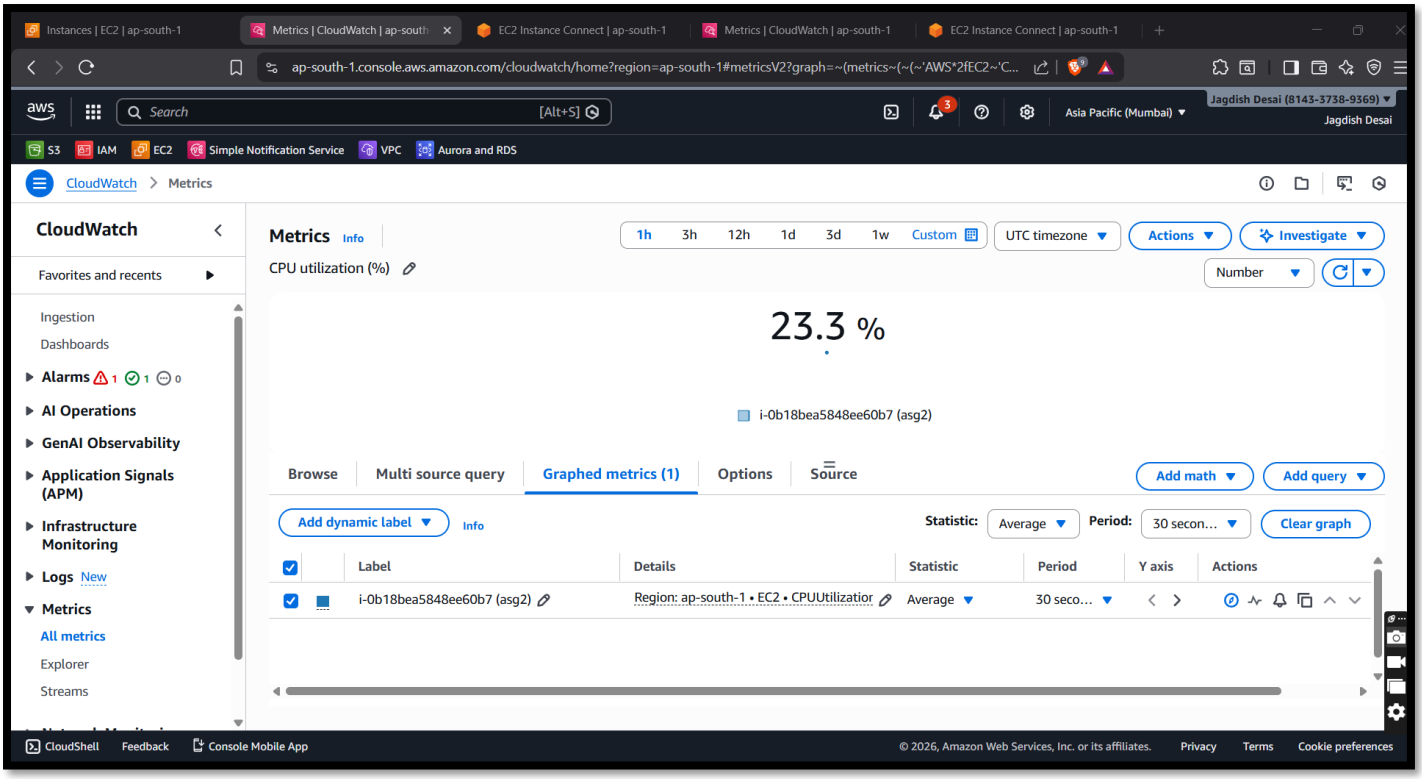
Connect to new instance and increasing load

stress --cpu 80 --io 4 --vm 2 --vm-bytes 128M --timeout 1000s&

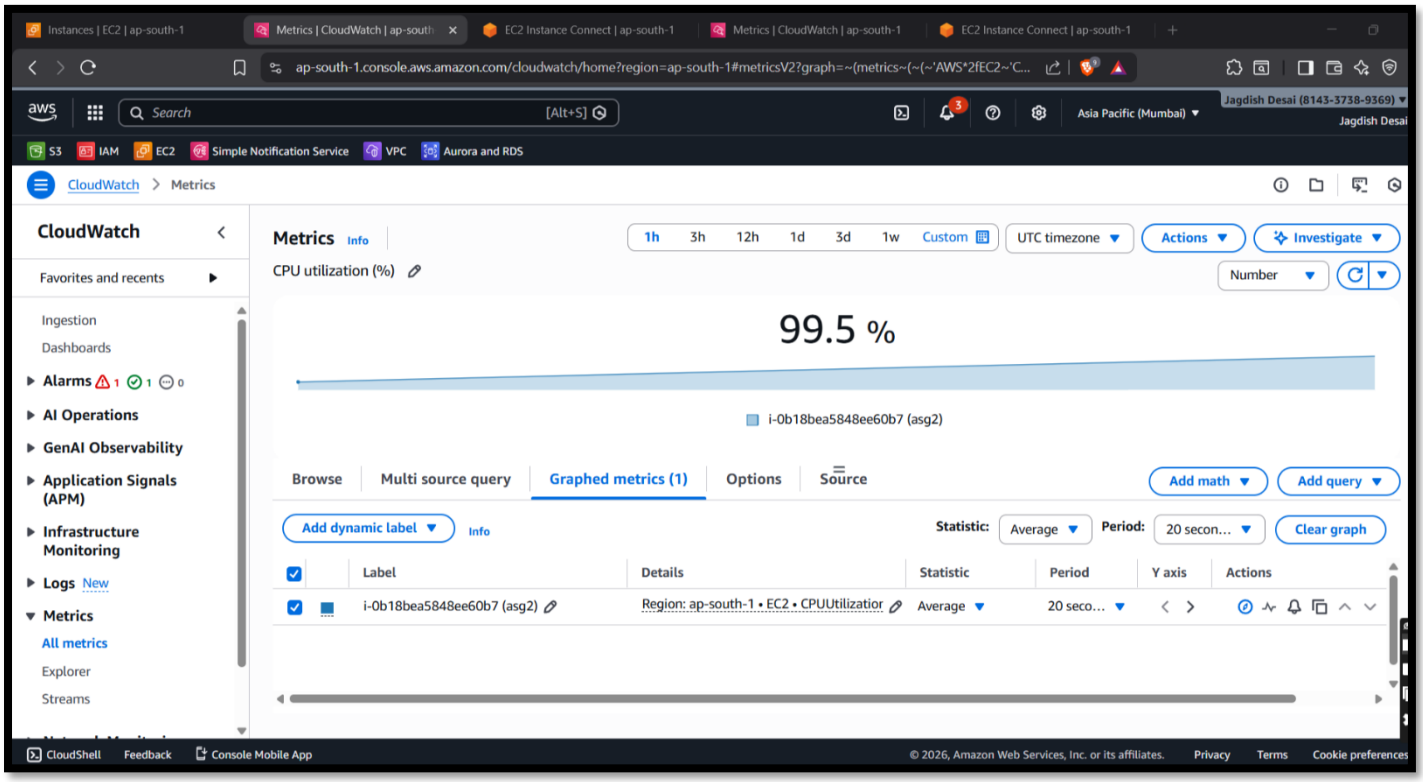


Monitor Cpu Utilization

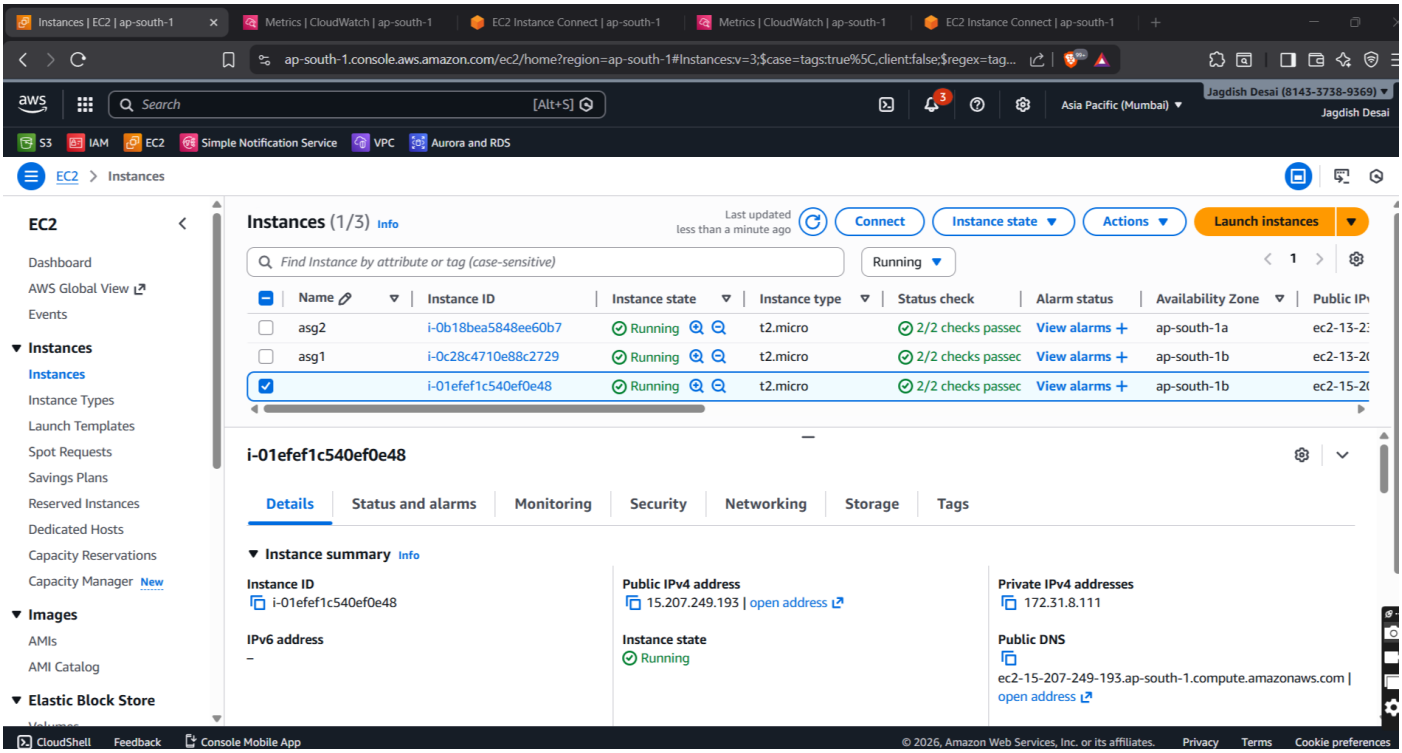




After Giving More Stress :



Go To Instances and check



New Instance will Be automatically created after load will increases.

Asg1 ip - 13.204.62.154 private ip- 172.31.6.7

Asg2 ip - 13.233.232.51 private ip- 172.31.41.153

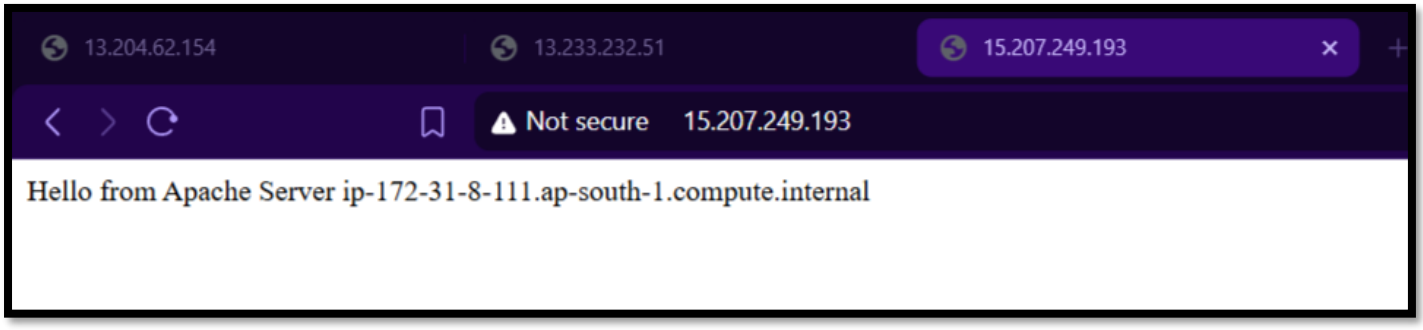
Asg3 ip - 15.207.249.193 private ip- 172.31.8.111

Checking Public Ip of each instance on browser:

Asg1 Instance:

Asg2 Instance:

Asg3 Instance:



It Shows and runs our script i.e hostname means Name of the machine (AWS uses private Ip).

ALB + ASG Setup workflow:

1. **Launched EC2 / Prepare Launch Template**
    - Free-tier AMI (Amazon Linux 2 / Ubuntu)
    - Instance type: t2.micro
    - User Data: install Apache / app
2. **Launched Template**
3. **Created Target Group**
    - Type: Instance
    - Protocol: HTTP, Port 80
    - Health check path: /
4. **Created Application Load Balancer (ALB)**
    - Internet-facing, 2 public subnets
    - Listener: HTTP → 80
    - Attach Target Group
5. **Created Auto Scaling Group (ASG)**
    - Select Launch Template
    - VPC + 2 subnets
    - Attach ALB + Target Group
    - Capacity: Min 1, Desired 1, Max 2
    - Scaling policy: CPU 50%
6. **Verified Setup**
    - ALB DNS → App accessible
    - Target Group → Healthy
    - ASG → EC2 scales automatically

Over-Provisioning Means?

Over-provisioning = Allocating more resources (EC2 instances, CPU, memory) than actually needed.

In my project:

- If we manually launch 3 EC2 instances but our app only needs 1 → extra 2 instances are **unused** → you **pay unnecessarily**.
- This wastes money and is inefficient.

How Auto Scaling Helps

1. **Auto Scaling Group (ASG)** monitors CPU or traffic.
2. It **launches instances only when needed** (high traffic).
3. It **terminates instances when traffic is low**.

Result:

- No extra EC2 running → **avoids over-provisioning**
- Cost is optimized → you only pay for what is used

Example in our Project

Scenario	**EC2 Instances Needed**	**Manually Launched**	**Auto Scaling**
Low traffic	1	3 (over-provisioned)	1 (optimal)
High traffic	2	3	2–3 (scaled automatically)