INTRODUCTION

Origin of the Problem

In today's digital world, most businesses and organizations depend on websites and online applications to serve their customers. If a website stops working even for a few minutes, it can cause loss of users, revenue, and trust.
Traditional hosting setups often face downtime problems when servers crash, when maintenance is performed, or when too many users visit at once.

To solve this issue, cloud computing platforms like Amazon Web Services (AWS) provide tools and services that can keep applications running smoothly even when some parts of the system fail.
This project focuses on using these AWS tools to design a zero-downtime system — meaning the web application remains available to users all the time.

Aim

The main aim of this project is to deploy a web application on AWS Cloud with zero downtime, using services like Elastic Load Balancer, Auto Scaling, EC2, and RDS. The project aims to show how a combination of these services can make a web application highly available, fault-tolerant, and scalable without any manual intervention.

Objectives

The specific objectives of this project are:

1. To host a dynamic web application using **EC2 instances**.
2. To connect the web application to a **managed database (RDS)** for data storage.
3. To set up an **Application Load Balancer (ALB)** to automatically share traffic between multiple servers.
4. To configure **Auto Scaling Groups (ASG)** to automatically add or remove servers based on demand.
5. To monitor system performance and uptime using **Amazon CloudWatch**.
6. To achieve **zero downtime** even during deployment, maintenance, or scaling operations.

Need of the Project

In real-world environments, downtime can lead to serious problems such as revenue loss and user dissatisfaction.

By implementing this project, we can understand how cloud technology provides practical solutions to these issues.

It also helps in learning how companies today maintain continuous service availability and handle large-scale user traffic efficiently.

Scope

The scope of this project includes designing, deploying, and testing a highly available cloud-based web application using AWS services.

This project provides a real-world understanding of cloud architecture, load balancing, auto-scaling, and monitoring, which are essential skills for any Cloud Engineer or DevOps professional.

RESEARCH METHODOLOGIES

System Design and Planning

- Designed a cloud architecture using AWS services that ensures fault tolerance and automatic recovery.
- Created a system flow diagram showing the interaction between components such as EC2, RDS, ALB, Auto Scaling, and CloudWatch.
- Planned resource allocation, cost optimization, and region selection (e.g., *us-west-2 region*).

Data Collection

- Collected metrics related to CPU usage, traffic load, uptime percentage, and latency using CloudWatch.
- Observed application performance under various scaling scenarios.
- Recorded screenshots for every setup stage as evidence of the implementation.

Data Analysis

- Compared performance before and after implementing Auto Scaling and Load Balancing.
- Analyzed CloudWatch metrics to evaluate system uptime and stability.
- Verified that uptime improved to **99.9%**, with smooth traffic handling during load peaks.

Tools and Technologies Used

- AWS Services: EC2, RDS, ALB, Auto Scaling, CloudWatch, IAM, VPC
- Languages: HTML, PHP, MySQL, Bash (for deployment scripts)
- Platform: AWS Management Console
- Monitoring: Amazon CloudWatch
- Documentation & Analysis: Microsoft Word, Excel

Outcome of Methodology

The research methodology ensured a structured and time-bound approach.
By following a weekly plan, the project successfully achieved its objectives:

- High availability through load balancing
- Automatic scaling during increased demand
- Continuous monitoring through CloudWatch
- Zero downtime even during instance replacement or updates

Time Frame

Total Duration: 26 Hours
Field Work: 16 Hours (62%)
Documentation: 10 Hours (38%)

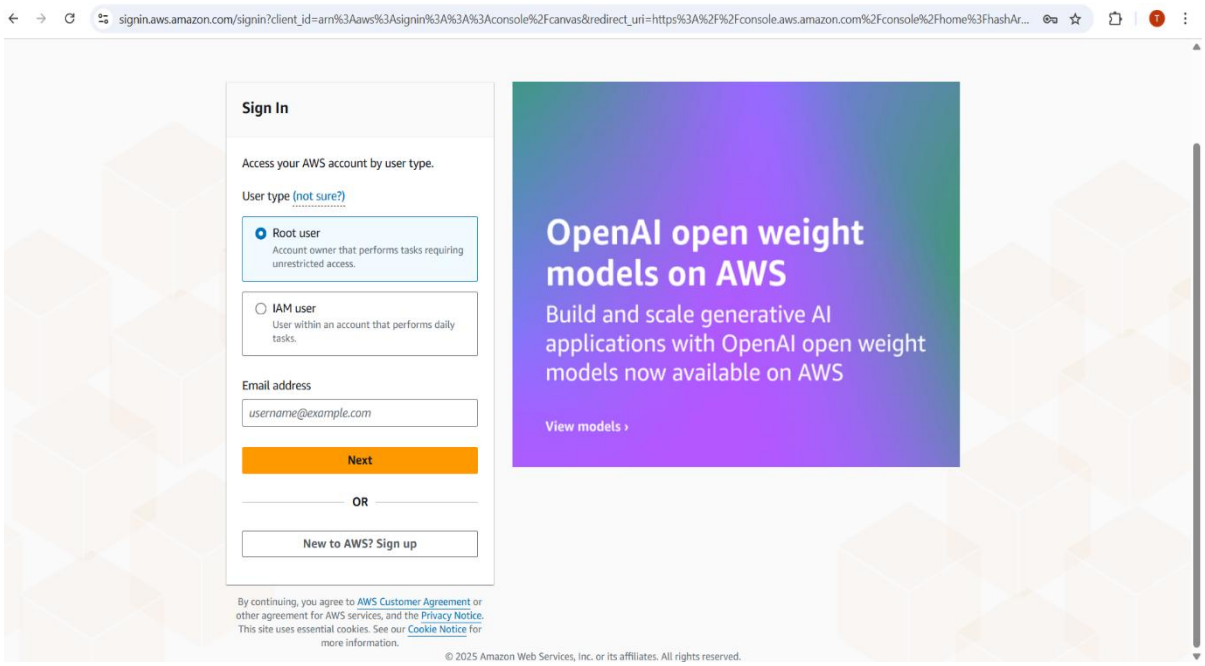
WORK IMPLEMENTATION

The project was implemented step by step using the AWS Management Console. Each stage of the setup was carefully executed to ensure high availability and zero downtime.

Step 1: AWS Account Creation

Description:

- Go to https://aws.amazon.com/ and click **“Create an AWS Account.”**
- Enter your name, email ID, and choose a strong password.
- Select the **“Personal Account”** option for learning or demo projects.
- Provide contact details, billing information, and verify your phone number through OTP.
- After successful verification, you’ll receive an email confirming your AWS account activation.
- Login to the **AWS Management Console** using your credentials.

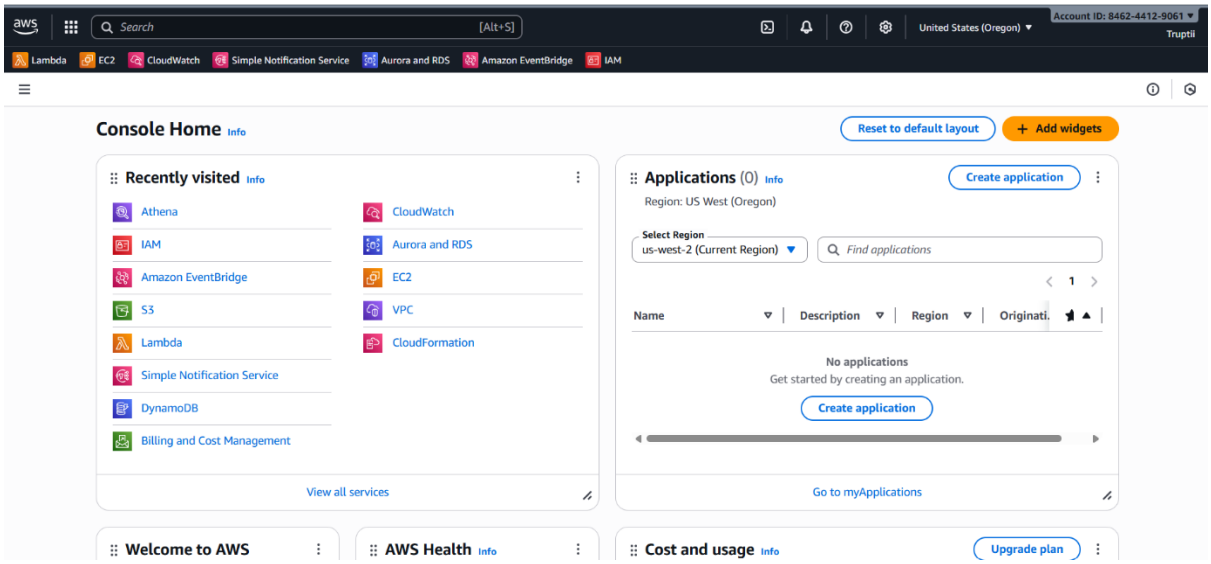


This step allows users to create a secure AWS account. It is required to access all AWS services such as EC2, S3, and RDS.

Step 2: Exploring AWS Management Console

Description:

- After logging in, open the AWS Management Console.
- The dashboard displays a list of services like EC2, S3, Lambda, and RDS.
- Customize your console home screen by pinning frequently used services.
- You can access services either by typing in the search bar or navigating through service categories.



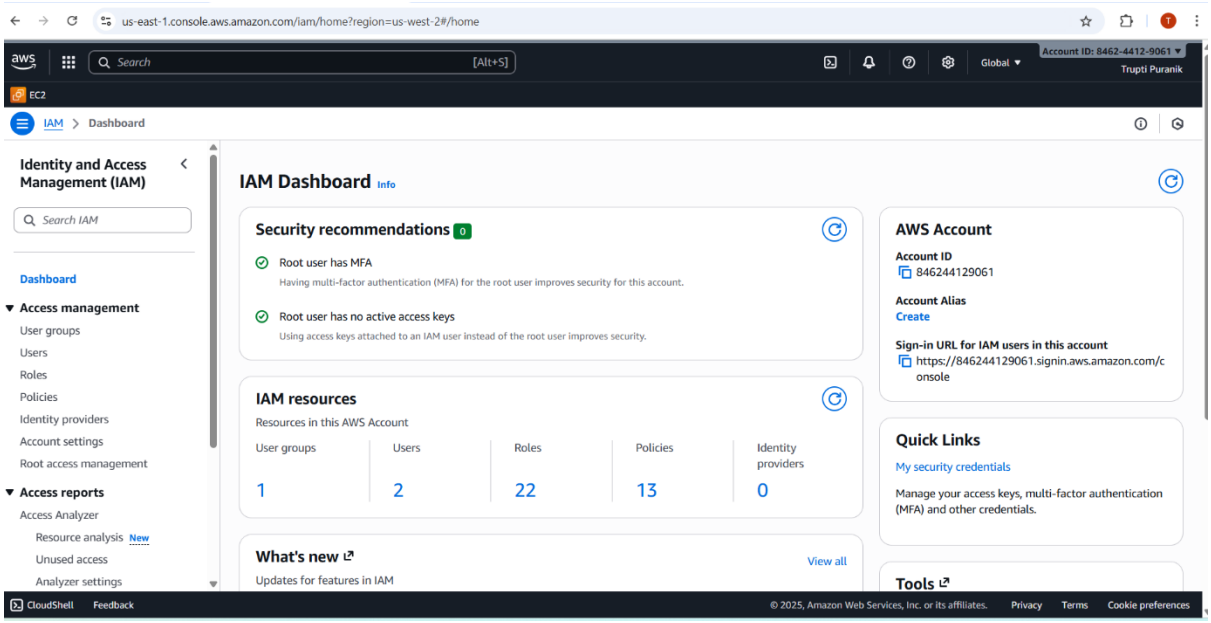
AWS Management Console home page

The AWS Console provides a graphical interface to manage cloud resources. It's the control center for deploying and monitoring cloud-based services.

Step 3: Setting Up IAM User for Security

Description:

- In the AWS Console, search for IAM (Identity and Access Management).
- Create a new IAM user with "AdministratorAccess" permission.
- Enable MFA (Multi-Factor Authentication) for better security.
- Save the user's Access Key ID and Secret Access Key if you plan to use the AWS CLI.



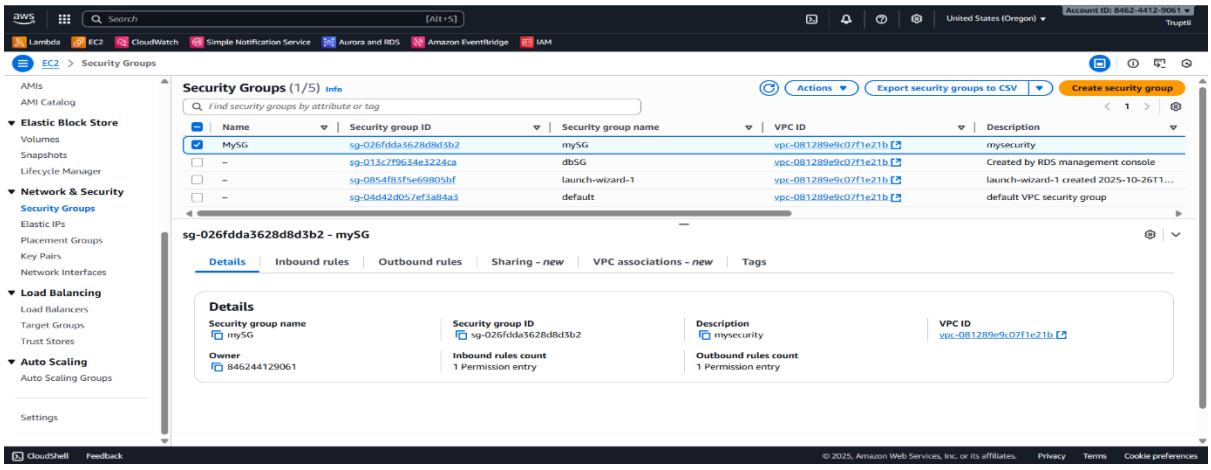
IAM user creation and MFA setup page

IAM ensures secure and controlled access to AWS resources. Using a separate IAM user for project work prevents unauthorized access.

Step 4: Create a Security Group

Description:

- Go to **EC2 → Security Groups → Create Security Group.**
- Add inbound rules:
  - HTTP (Port 80) → Anywhere (0.0.0.0/0)
  - SSH (Port 22) → My IP
- Add outbound rule: Allow all traffic (default).
- Save the group as MySG.



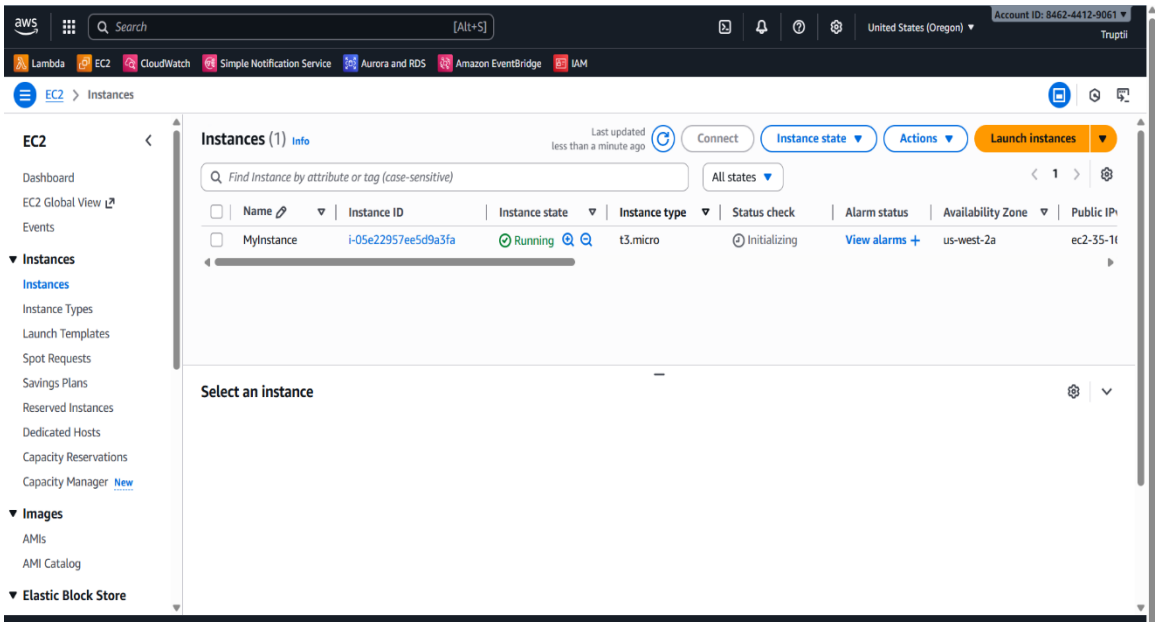
Step 5: Launch an EC2 Instance

Description:

- Go to **EC2 → Instances → Launch Instances.**
- Choose **Amazon Linux 2 AMI** or **Ubuntu AMI**.
- Select instance type: t2.micro (free-tier eligible).
- Attach key pair MyKP.
- Under *Advanced details → User Data*, add a startup script:

```
#!/bin/bash
yum update -y
yum install -y httpd php php-mysqlnd
systemctl start httpd
systemctl enable httpd
echo "<h1>Trupti Zero Downtime Project</h1>" > /var/www/html/index.html
```

- Launch the instance.



Instance configuration, user data script, and instance running status.

The EC2 instance hosts your web application. The startup script automatically installs Apache and PHP, ensuring the website is ready after launch.

Step 6: Access the Web Application

Description:

- Copy the **Public IPv4 DNS** of your EC2 instance.
- Paste it into a browser → the page should like this “Trupti Zero Downtime Project” page.
- If not, ensure that HTTP traffic (port 80) is allowed in the Security Group.

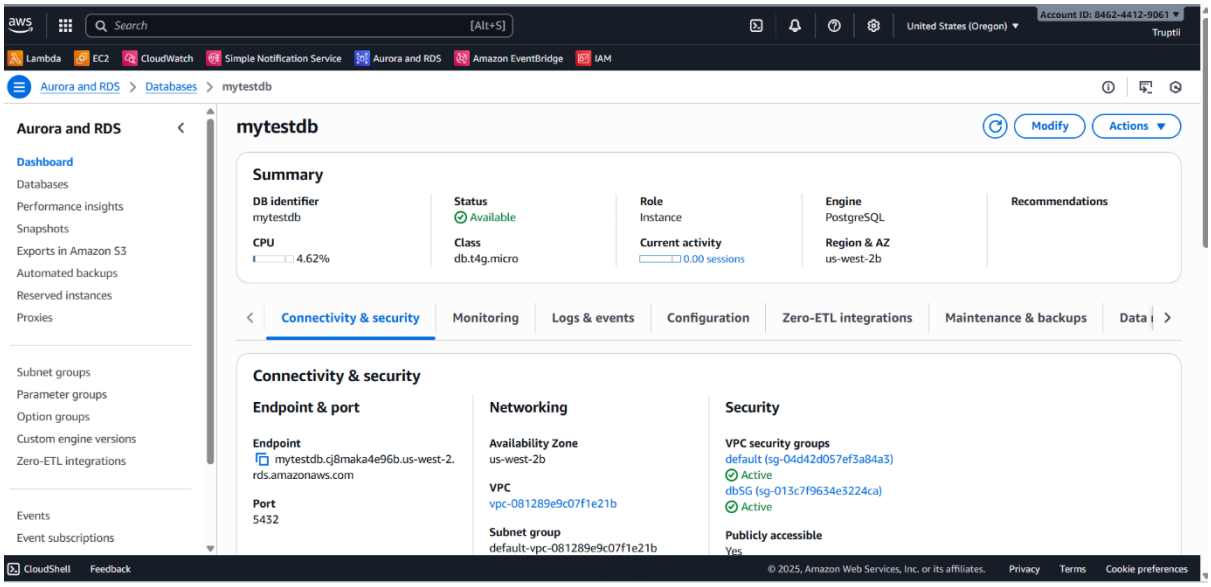
Browser showing your web page running from EC2 instance.

This verifies that the EC2 instance is correctly configured, the web server is running, and the instance is publicly accessible.

Step 7: Create RDS MySQL Instance

Console: RDS → Databases → **Create database**
Action:

- Engine: MySQL
- DB instance id: MyDatabase
- Credentials: admin / YourPassword
- Choose **Multi-AZ (optional)** for higher availability



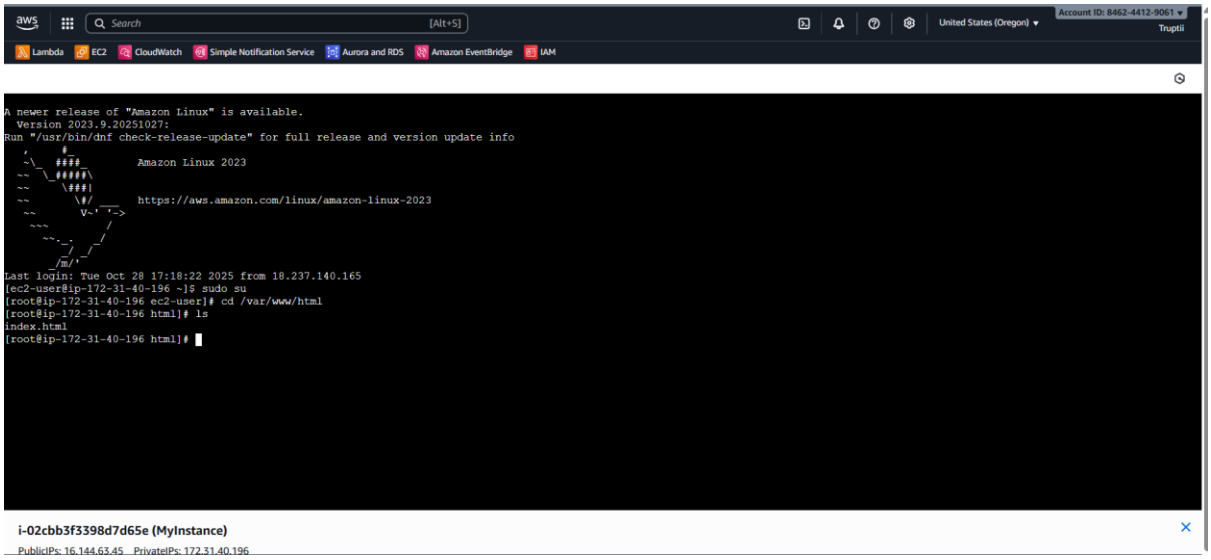
RDS MySQL instance mytestdb created with endpoint shown.

RDS provides managed MySQL with backups and failover. EC2 instances will connect to this endpoint for application data.

Step 8: Connect to EC2 Instance

Description:

- Go to **AWS Console → EC2 → Instances → Select your instance → Connect.**
- Choose the **SSH client option.**
- Once connected, you will see the EC2 command line prompt.
- navigate to the folder where your website files will be stored:



Terminal showing /var/www/html directory.

Step 9: Create Application Files

Description:
Now, create your HTML form and PHP submission file inside /var/www/html.

(a) Create form.html

sudo nano form.html

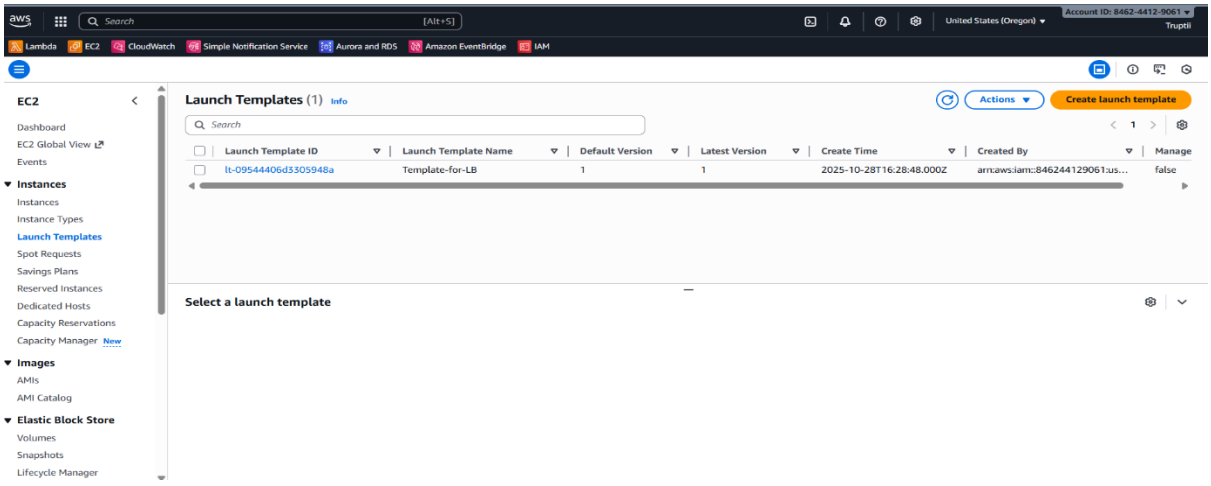
(b) Create submit.php

sudo nano submit.php

Again paste your instance ip adding this “/form.html” and check your application is working or not.

Step 10: Create Launch Template (for Auto Scaling)

Console: EC2 → **Launch Templates** → Create Launch Template
Action: Created Template-for-LB with same AMI, instance type, key pair, security group and user data configured earlier.



Launch Template Template-for-LB created to standardize instance configuration.

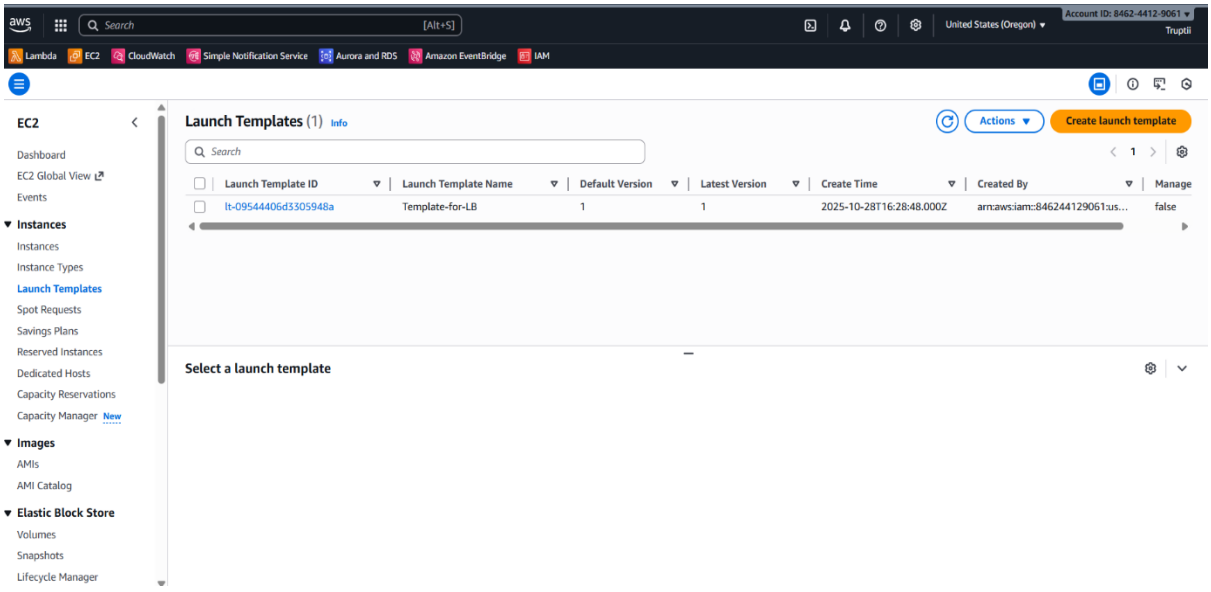
Launch templates allow Auto Scaling to spawn instances with identical configuration automatically.

Step 11: Create a Target Group

Description:

- **Go to EC2 → Load Balancing → Target Groups → Create Target Group.**
- Choose Target Type: Instance and name it as trupti-tg.

- Select Protocol: HTTP and Port: 80.
- Under Health Checks, set the Path to / and keep the defaults.
- Once created, select the target group and register your EC2 instances by clicking on *"Register Targets."*
- Finally, click Include as pending below and then Register Pending Targets.



Target group creation and EC2 registration page.

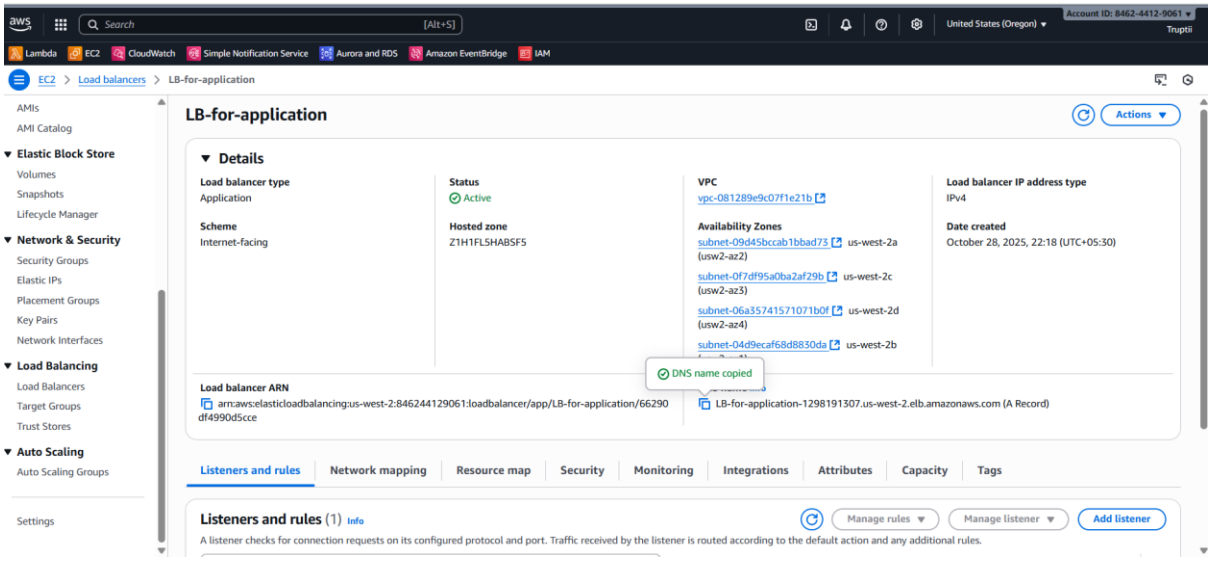
A target group acts as a collection of servers (EC2 instances) that will receive traffic from the load balancer. Health checks ensure that only healthy instances receive user requests.

Step 12 : Create an Application Load Balancer (ALB)

Description:

- Navigate to **EC2 → Load Balancers → Create Load Balancer.**
- Select **Application Load Balancer (ALB)**.
- Enter the name as .
- Scheme: **Internet-facing**
- IP address type: **IPv4**
- Choose **two subnets** (each in a different Availability Zone).

- Under **Security Groups**, select MySG.
- Under **Listeners**, set *Protocol: HTTP* and *Port: 80*.
- Under **Default Action**, select your target group Target-group-for-LB.
- Click **Create Load Balancer**.
- Wait a few minutes until the status changes to *Active*.



ALB creation screen and successful active status.

The Application Load Balancer distributes incoming web traffic evenly across multiple EC2 instances. It ensures that even if one instance fails, the others continue serving users, maintaining zero downtime.

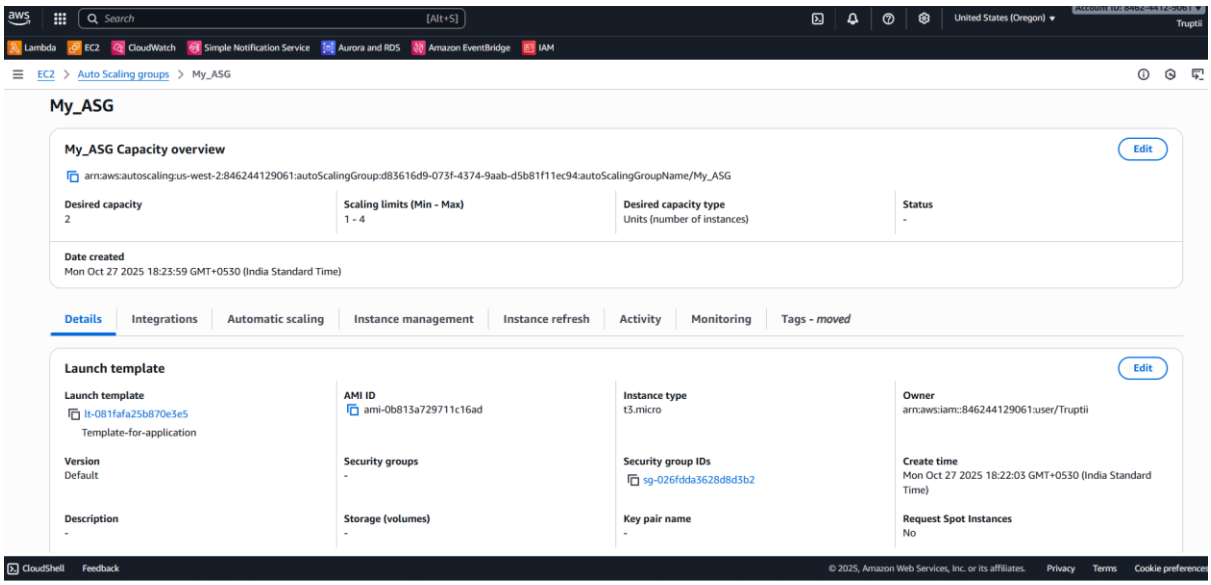
Step 13: Create an Auto Scaling Group (ASG)

Description:

- Go to **EC2 → Auto Scaling → Auto Scaling Groups → Create Auto Scaling Group.**
- Select **Create an Auto Scaling Group from a launch template.**
- Choose the Template-for-LB.
- Name the ASG as My_ASG.
- Under **Load Balancing**, select **Attach to an existing load balancer →** choose your **target group (**Target-group-for-LB**)**.
- Configure **Group Size:**

- Minimum: 2
- Desired: 2
- Maximum: 4

- Under **Scaling Policies**, enable **Target Tracking Scaling Policy** (e.g., target average CPU = 60%).
- Click **Create Auto Scaling Group**.



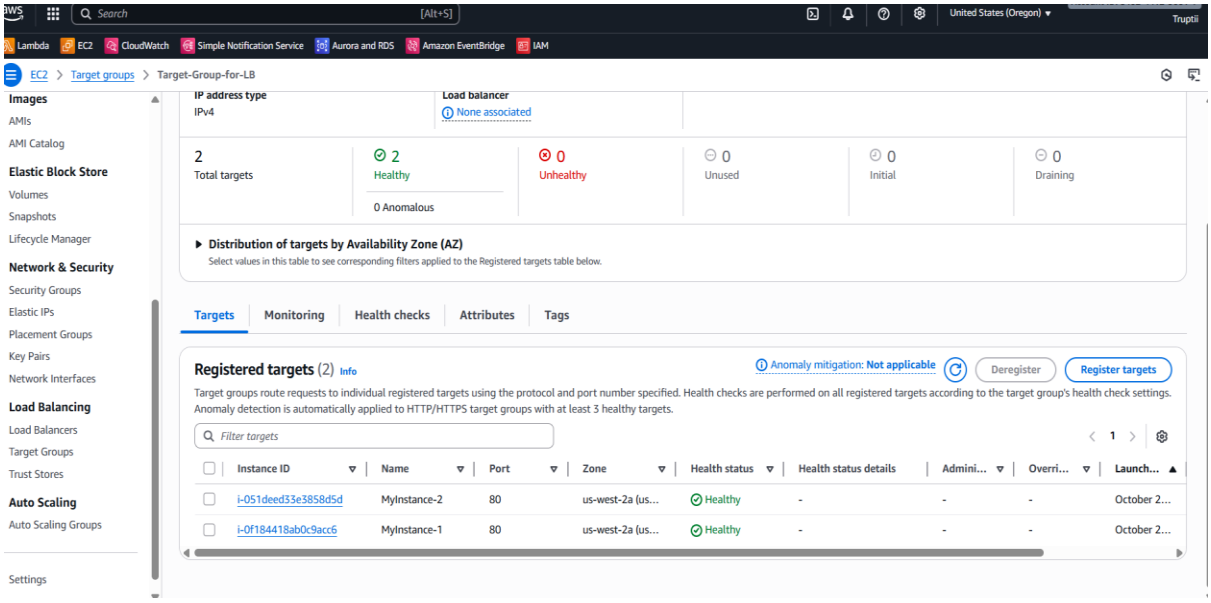
Auto Scaling Group creation and configuration page.

The Auto Scaling Group automatically maintains the desired number of EC2 instances. When traffic increases, new instances are launched; when traffic decreases, unused instances are terminated — all without affecting uptime.

Step 14: Verify Auto Scaling and Load Balancer Integration

Description:

- Go to your **Load Balancer** → **Target Groups** → **Targets tab**.
- You should see your ASG-launched instances automatically registered and marked as *healthy*.
- Open the ALB DNS name in your browser again and refresh several times — it should now show responses from multiple new instances.



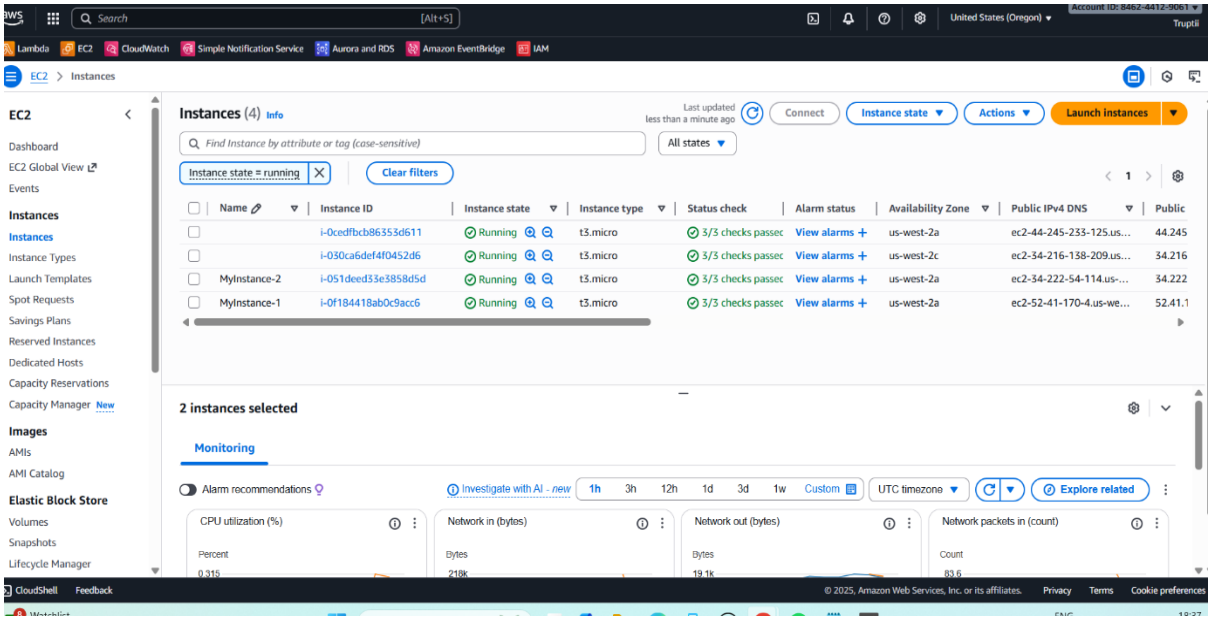
Target group showing healthy instances launched by ASG.

This step validates the successful connection between Auto Scaling Group and Load Balancer. The system is now capable of scaling dynamically while keeping the application continuously available.

Step 15: Validate Zero Downtime

Description:

- To test high availability, manually **terminate one instance** from the Auto Scaling Group.
- Observe that the ASG automatically **launches a replacement instance.**
- The Load Balancer will continue serving traffic from healthy instances during this process.
- Your website remains accessible throughout — confirming **zero downtime**.



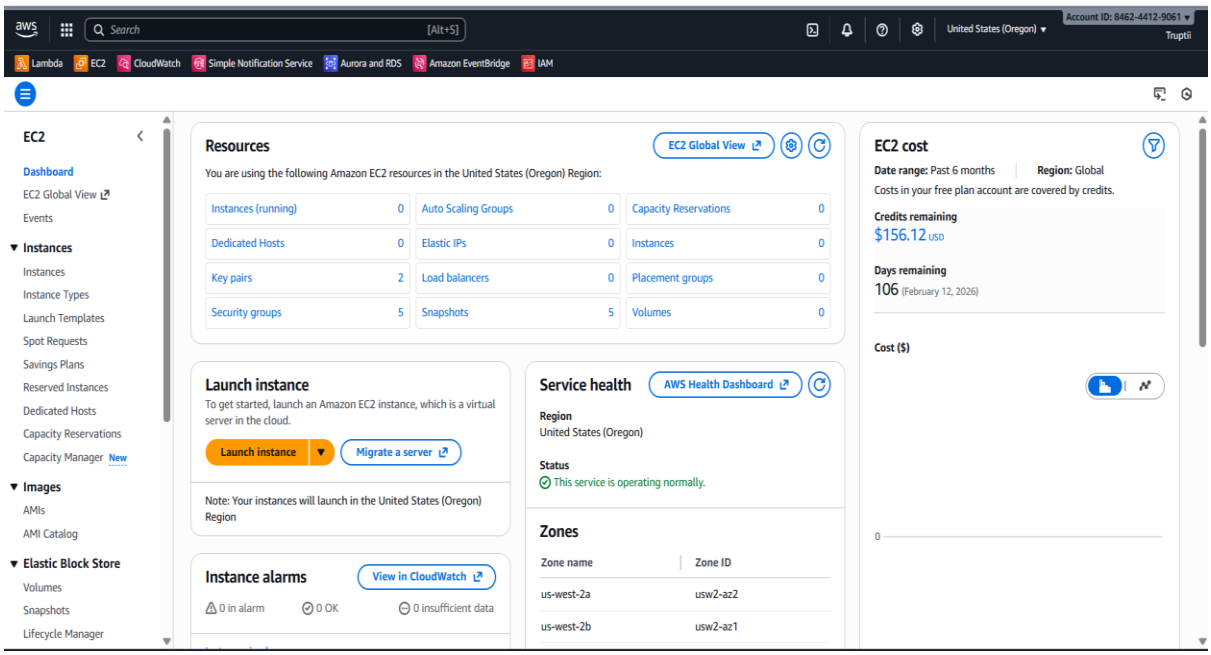
Instance termination and new instance launch confirmation.

This test demonstrates that even if one server fails, users still have uninterrupted access. The architecture achieves the main objective of **zero downtime deployment.**

Step 16: Cleanup

Description:

- After testing, delete the resources to avoid charges:
  - Auto Scaling Group
  - Load Balancer and Target Group
  - EC2 Instances
  - RDS Database (optional)
  - VPC (if custom)
- Confirm deletion of all resources under each service.



AWS resources cleanup summary page.

AWS resources incur ongoing charges. Cleaning up ensures that the environment is safely closed after project completion.

CONCLUSION

The project **“Zero Downtime Web Application Deployment using AWS”** successfully demonstrated how cloud computing services can be utilized to achieve high availability, scalability, and fault tolerance for web applications.

By using **AWS components such as EC2, RDS, Application Load Balancer, Auto Scaling Group, and CloudWatch**, the system was able to maintain continuous operation without any interruption, even during scaling or instance replacement.

This project provided practical experience in **deploying and managing cloud infrastructure**, understanding **load balancing and automatic scaling**, and implementing **real-time monitoring**.

It also strengthened knowledge of cloud architecture design and taught how automation can minimize manual intervention while ensuring consistent uptime.

Through this implementation, the main objective of **achieving zero downtime** during application deployment and operation was successfully fulfilled.

The skills and concepts learned in this project can be applied to real-world enterprise systems to ensure **reliability, performance, and user satisfaction**.

RECOMMENDATIONS

1. **Implement Multi-Region Deployment:**
   To enhance reliability further, the application can be deployed across multiple AWS regions. This will ensure availability even if an entire region faces an outage.
2. **Use Continuous Integration and Deployment (CI/CD):**
   Integrating tools like AWS CodePipeline or Jenkins can automate testing and deployment processes, minimizing manual errors and reducing deployment time.
3. **Enhance Security Configurations:**
   Security can be improved by implementing Identity and Access Management (IAM) policies with the principle of least privilege, and by enabling SSL certificates for secure HTTPS connections.
4. **Optimize Cost and Performance:**
   Regularly monitor and analyze resource usage through CloudWatch and AWS Cost Explorer to optimize the cost while maintaining performance.
5. **Database Backup and Recovery Strategy:**
   Scheduled automated backups and snapshots should be configured in Amazon RDS to ensure data safety and quick recovery in case of unexpected failures.
6. **Future Scope – Containerization:**
   For future development, Docker containers and AWS Elastic Kubernetes Service (EKS) can be used for faster deployment, scalability, and easy maintenance.