SYMBIOSIS
INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

CLOUD COMPUTING TOOLS AND TECHNIQUES LAB

PROJECT REPORT

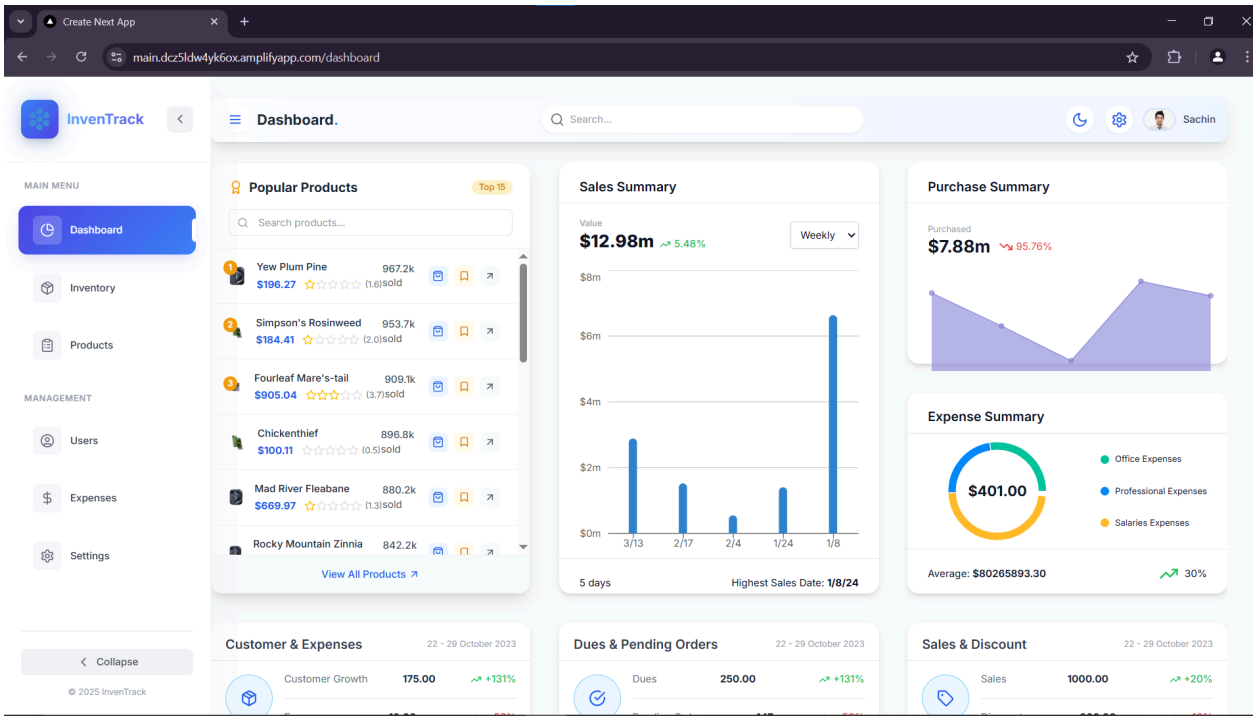
“Deploying a scalable 3 tier Inventory Management Web Application on AWS using AWS Amplify and EC2”

Presented By:
Sachin Mhetre - 22070122119
Mihir Hebalkar - 22070122120
Onkar Mendhapurakar - 22070122135

Under the guidance of:
Dr. Aditi Sharma

Index

Introduction



Modern digital business operations need advanced and expandable inventory systems which can handle product management across multiple platforms and locations. This project solves this need by creating a specific full-stack cloud-based inventory system which manages smartphone product inventories.

The application implements a contemporary 3-tier architecture which divides frontend from backend and database to achieve modular development and scalability and maintainability. The Next.js frontend component operates from AWS Amplify while GitHub handles continuous deployment to support efficient development processes. The backend runs on Node.js and Express.js and operates from an Amazon EC2 server within a protected scalable structure. The secure API Gateway service from AWS enables front-to-back communication by exposing APIs.

The system stores its data in Amazon RDS (PostgreSQL) relational database which operates from a private subnet to boost security measures. The system achieves high availability and scalability through the implementation of AWS Auto Scaling Groups alongside Application Load Balancer (ALB) in multiple Availability Zones.

The system operates at peak efficiency because Amazon CloudWatch enables centralized monitoring and logging functions which track EC2, RDS, ALB, and API Gateway systems. The infrastructure provides both support for growth along with traffic spikes and delivers high reliability and cost efficiency.

Our solution builds a professional inventory management platform on AWS cloud services to create an application which scales well and ensures security while being maintainable for enterprise-class operations.

Problem Statement

Businesses that sell physical goods must manage inventory as their core operational function especially when they operate in the electronics retail sector where smartphone inventory needs precise management. A growing business demands increasingly complex inventory management which needs real-time monitoring of stock amounts alongside sales patterns and purchase activities and user permissions and expense tracking between different departments and sites.

Traditional inventory management systems show multiple significant shortcomings in their operations:

- **Scalability**: The performance of legacy systems suffers when they must handle high volumes of concurrent users and frequent data updates or sudden spikes in traffic which results in performance slowdowns and system downtime.
- **Data Consistency and Accuracy**:A disorganized system without centralization and proper architecture design leads to frequent stock discrepancies and delayed order fulfillment along with human errors.
- **Security and Access Control**: A disorganized system without centralization and proper architecture design leads to frequent stock discrepancies and delayed order fulfillment along with human errors.
- **Lack of Real-Time Monitoring**: Business loss occurs from unidentified problems such as overselling and stockouts and infrastructure failures because automated dashboards and alerts are not implemented.
- **Limited Flexibility**: Systems which do not support modular updates or cloud-native CI/CD methods become difficult to maintain and adapt to evolving

business needs.

Businesses that operate across various regions or platforms including web and mobile and Point of Sale systems need scalable cloud-native inventory management systems. This includes:

- Elastic scaling to meet changing workloads with solutions such as Auto Scaling Groups (ASG).
- High fault tolerance and availability through multi-AZ deployment and load balancing (ALB).
- Constant deployment pipelines for assured delivery of latest features or bug fixes without downtime.
- Seamless interoperability with cloud-native monitoring and logging platforms (e.g., Amazon CloudWatch) for real-time system administration.

A 3-tier inventory management web application development on AWS serves to resolve essential operational requirements. The system implements a modular structure that places Next.js through AWS Amplify in the frontend layer and Node.js on EC2 connected to API Gateway for backend services and PostgreSQL on Amazon RDS as the database. The system architecture supports both quick feature deployments through CI/CD pipelines and automatic horizontal scaling and dynamic response to business needs in real time.

Tech Stack

Frontend

- **Framework:** Next.js (via AWS Amplify Hosting)
- **Hosting Platform:** AWS Amplify
- **Deployment:** Continuous deployment from GitHub

Backend

- **Server:** Amazon EC2 (Ubuntu/Linux)
- **Language/Framework:** Node.js / Express.js
- **API Exposure:** AWS API Gateway (for HTTPS proxy to EC2 backend)

Database

- **Type:** Amazon RDS (PostgreSQL)
- **Access:** Private subnet access from backend EC2

Infrastructure & Scalability

- **Load Balancer:** AWS Application Load Balancer (ALB)
- **Auto Scaling:** Amazon EC2 Auto Scaling Group (ASG)
- **Subnets:** Public subnets across multiple Availability Zones (AZs)

Monitoring & Logging

- **Monitoring:** Amazon CloudWatch (for EC2, ALB, RDS, and API Gateway)
- **Log Management:** CloudWatch Logs and Custom Dashboards

Cloud Platform

- **Provider:** Amazon Web Services (AWS)
- **Region:** ap-south-1 (Mumbai)

Version Control & CI/CD

- **Version Control:** Git (via GitHub)
- **CI/CD:** GitHub to AWS Amplify (auto-deploy on push)

Objectives

The primary goal of this project is to **design, develop, and deploy a scalable 3-tier inventory management web application** on Amazon Web Services (AWS) that supports efficient stock tracking, real-time analytics, and seamless scalability for growing businesses. The application is intended to serve both end-users and administrators with optimized performance, high availability, and ease of management.

Specifically, the objectives are:

1. **To build a modular and scalable architecture** using AWS services, enabling smooth handling of increasing user and data loads.

2. **To provide an intuitive frontend interface** using Next.js hosted via AWS Amplify, supporting responsive design and seamless user interaction.

3. **To implement a secure and reliable backend** with Node.js and Express.js hosted on Amazon EC2, exposed via AWS API Gateway.

4. **To ensure secure data storage and efficient queries** through Amazon RDS (PostgreSQL) with access limited to backend servers.

5. **To leverage cloud-native tools** like AWS Auto Scaling, Load Balancer, and CloudWatch for dynamic scaling, availability, and monitoring.

6. **To automate deployment workflows** using GitHub and AWS Amplify for rapid iteration and continuous delivery.

7. **To offer essential inventory management features** that address real-world business challenges such as sales monitoring, stock tracking, and purchase summaries.

Key Features

Inventory Tracking

- Add, update, and manage product listings with pricing, stock, and ratings.

- View summaries of sales, purchases, and stock levels in real-time.

Dashboard & Analytics

- Interactive dashboard with key metrics: sales trends, stock status, customer growth, and expenses.
- Visual reports for better decision-making.

Scalable & Secure Cloud Setup

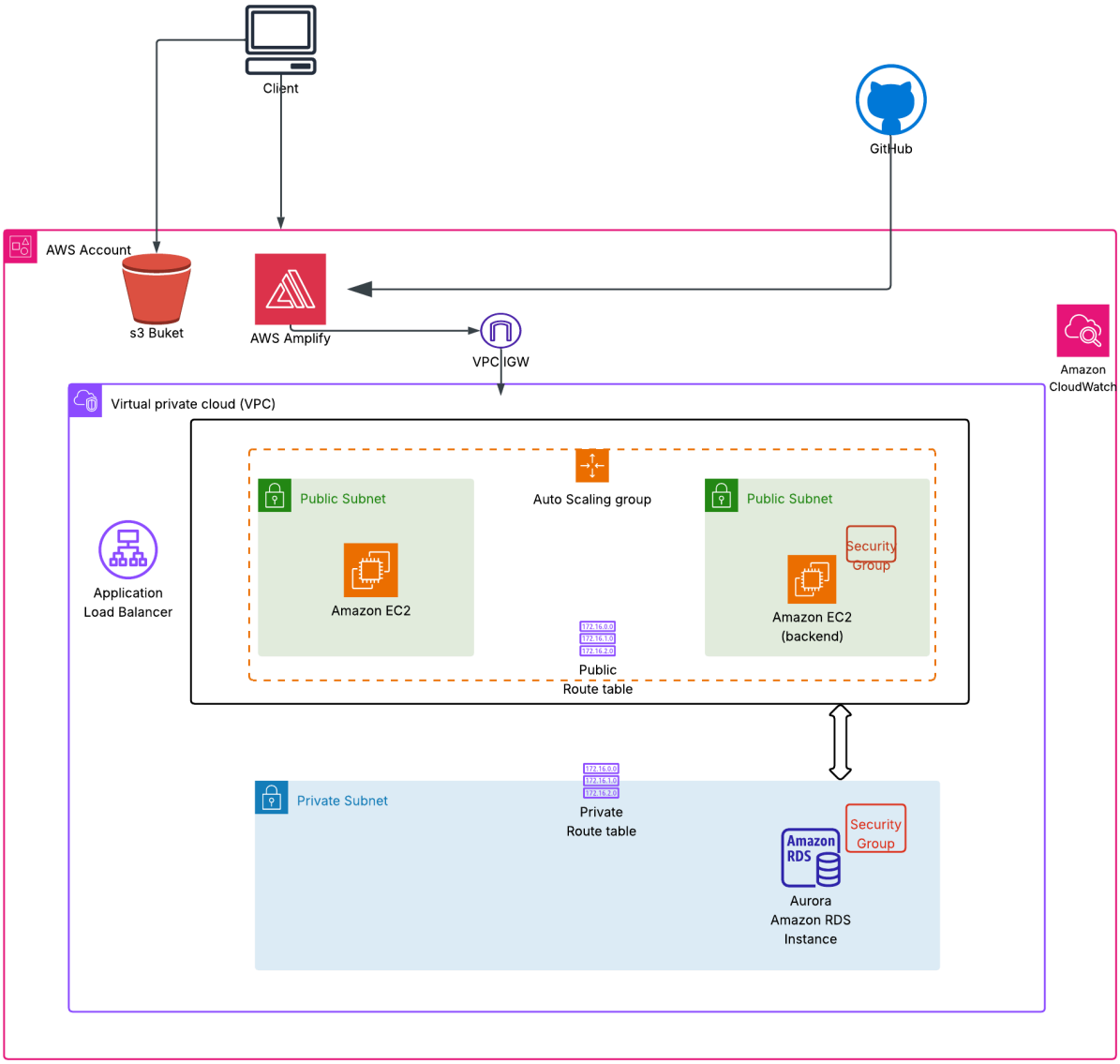
- Auto-scaling backend (EC2) with load balancer for high availability.
- Secure database access via private subnet (Amazon RDS).

Monitoring & Reliability

- Real-time performance tracking and error alerts via Amazon CloudWatch.

- CI/CD integration for seamless deployment using GitHub and AWS Amplify.

AWS Cloud Architecture

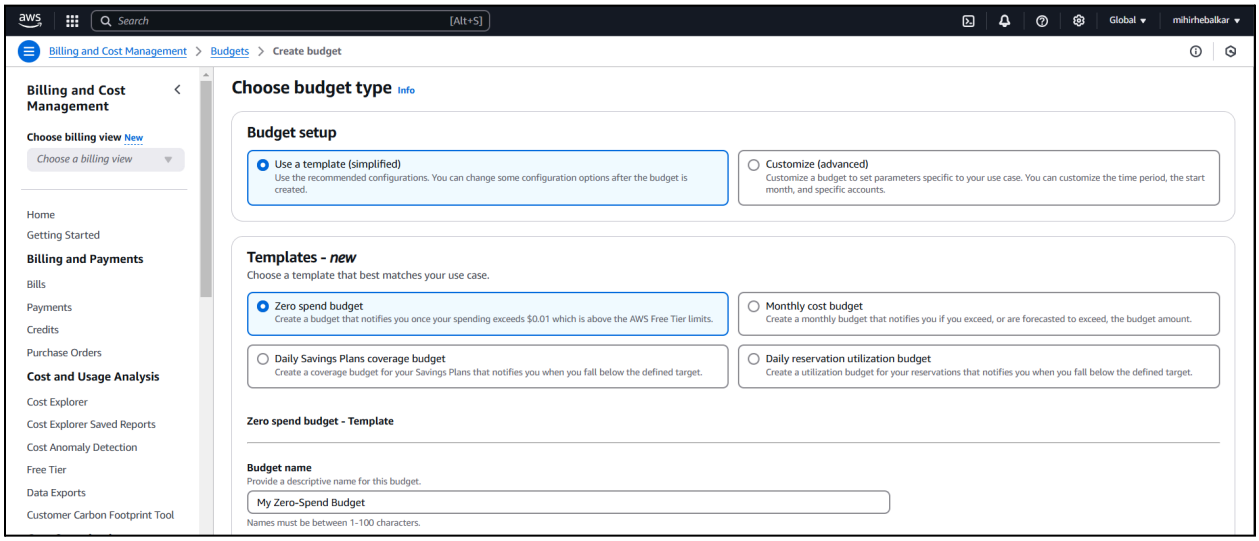


Project Setup and Steps followed:

Network Setup

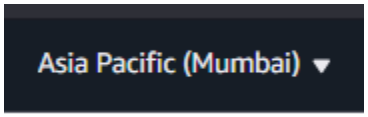
1. Setup budget

Setting up zero spend budget so that we get notified once our spending exceeds $0.01 which is above the AWS free tier limits.



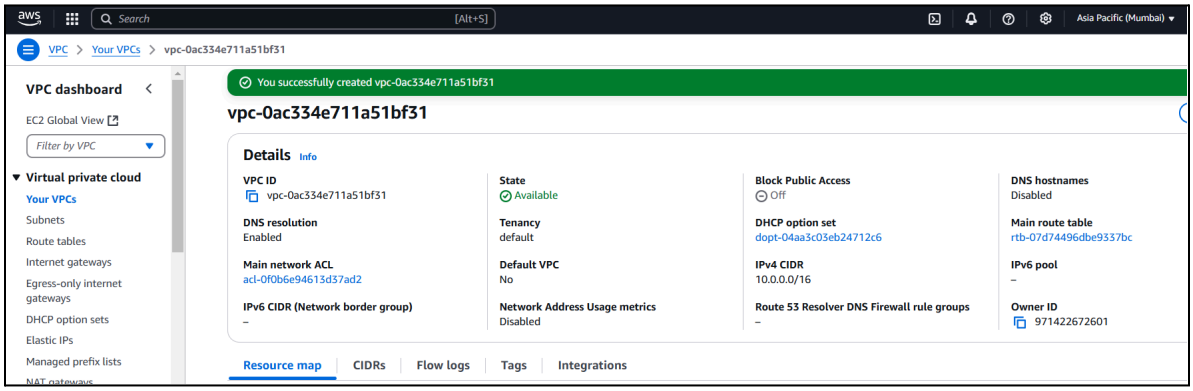
2. Selecting Region

We chose the Asia Pacific (Mumbai) region to ensure low latency for Indian users and comply with local data residency norms. It also offers cost-effective services with all required AWS features for our architecture.

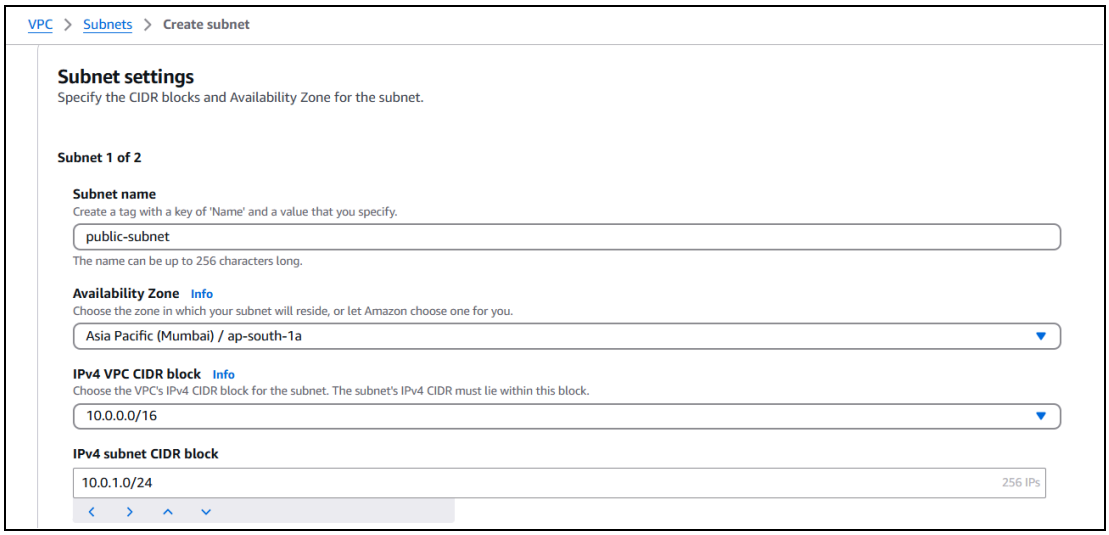


3. Creating VPC

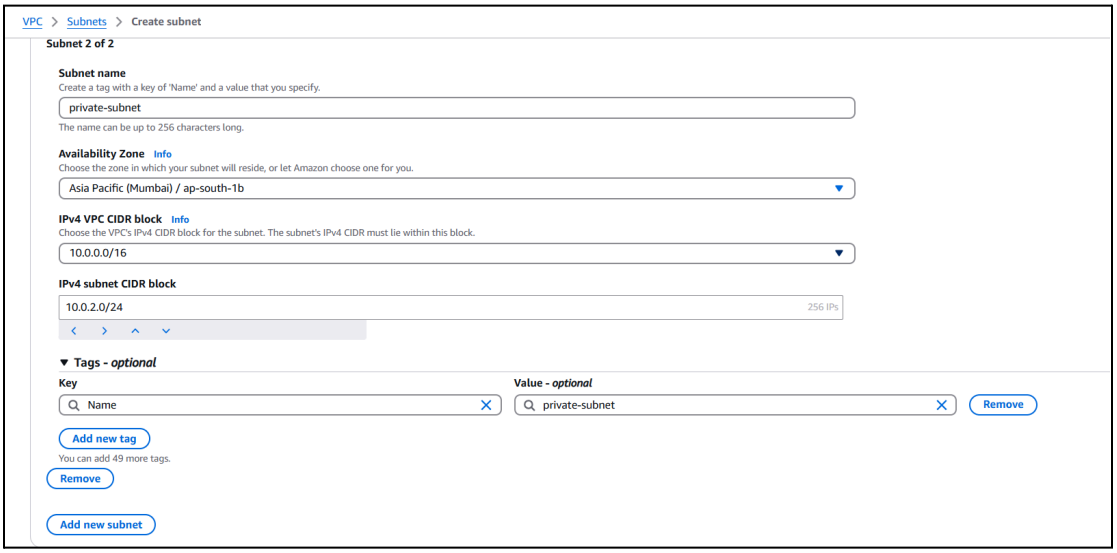
VPC creation with 10.0.0.0/16 CIDR block

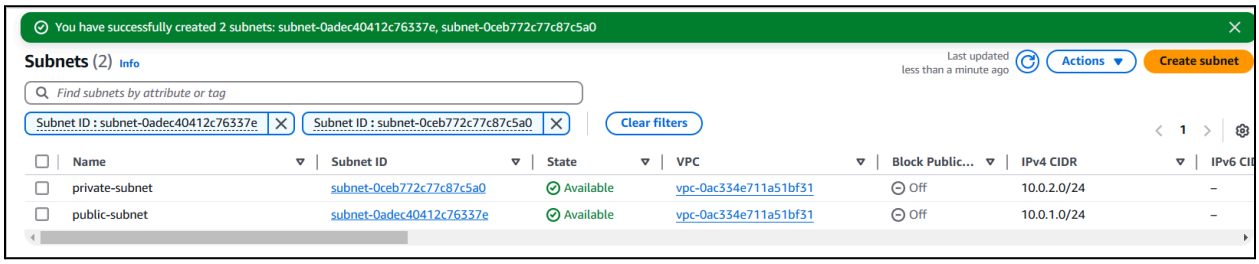


a. Public subnet (10.0.1.0/24) for internet-facing components

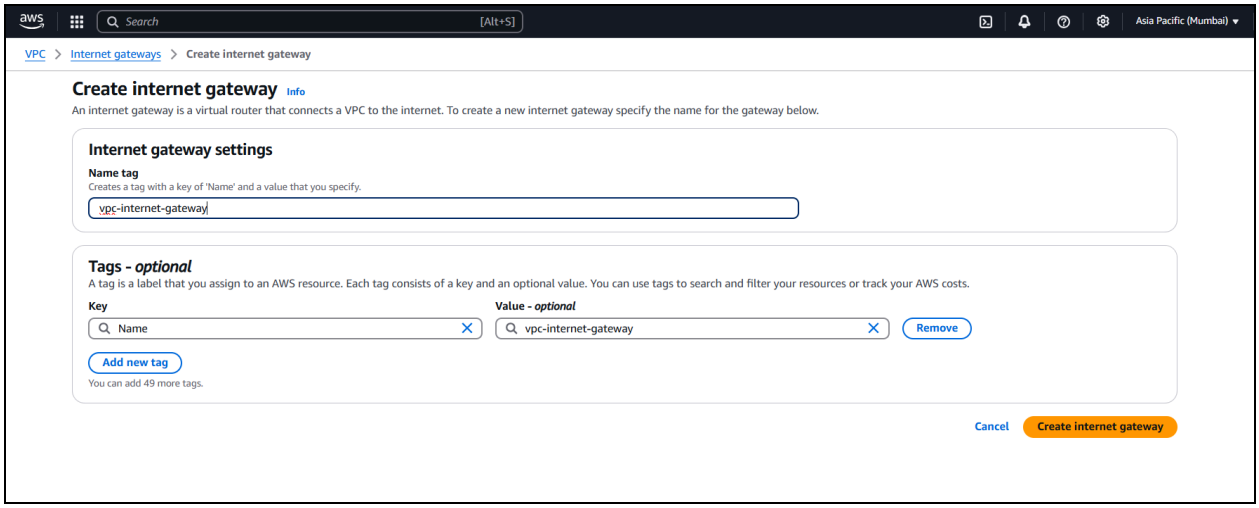


b. Private subnet (10.0.2.0/24) for database isolation



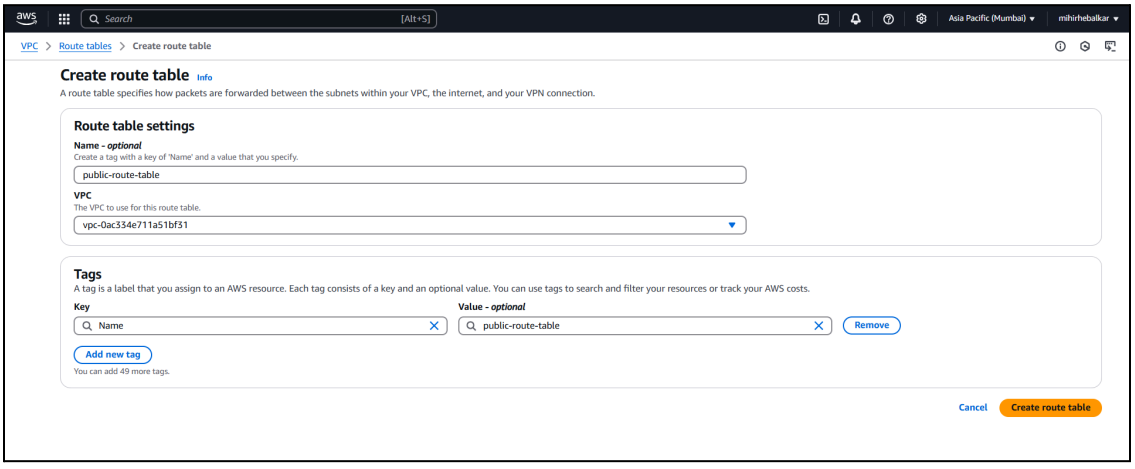


c. Created a VPC internet gateway to access subnets that were created



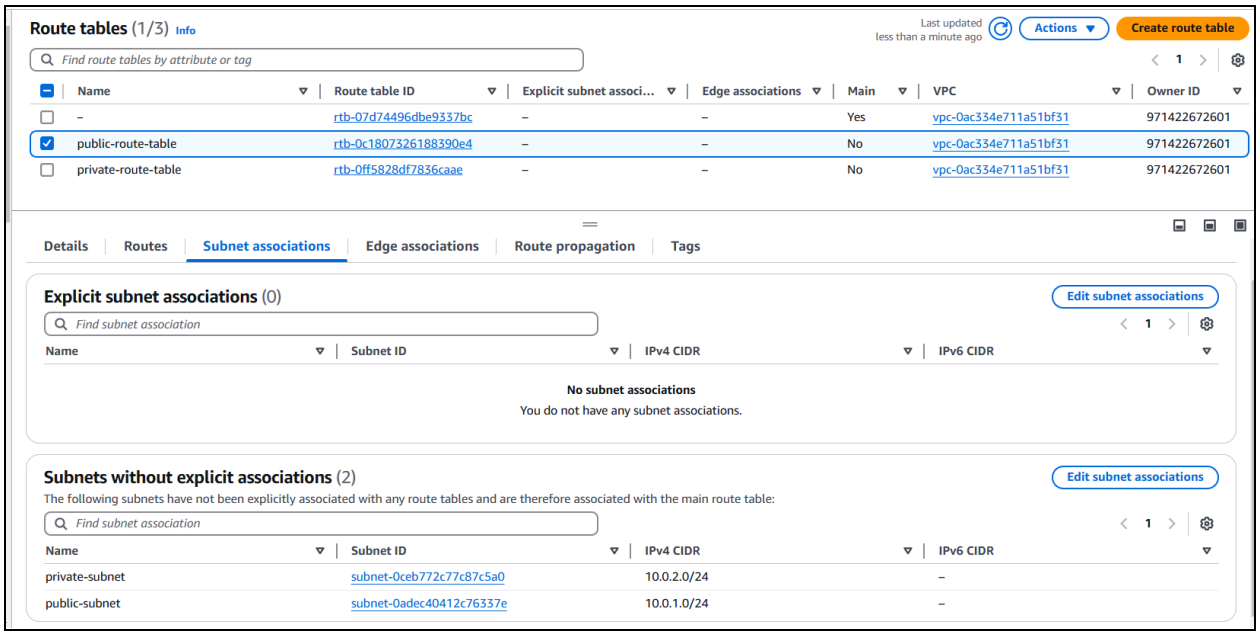
d. Attaching the created Internet gateway to VPC

e. Route table configuration



Same for Private route table

f. Subnet associations

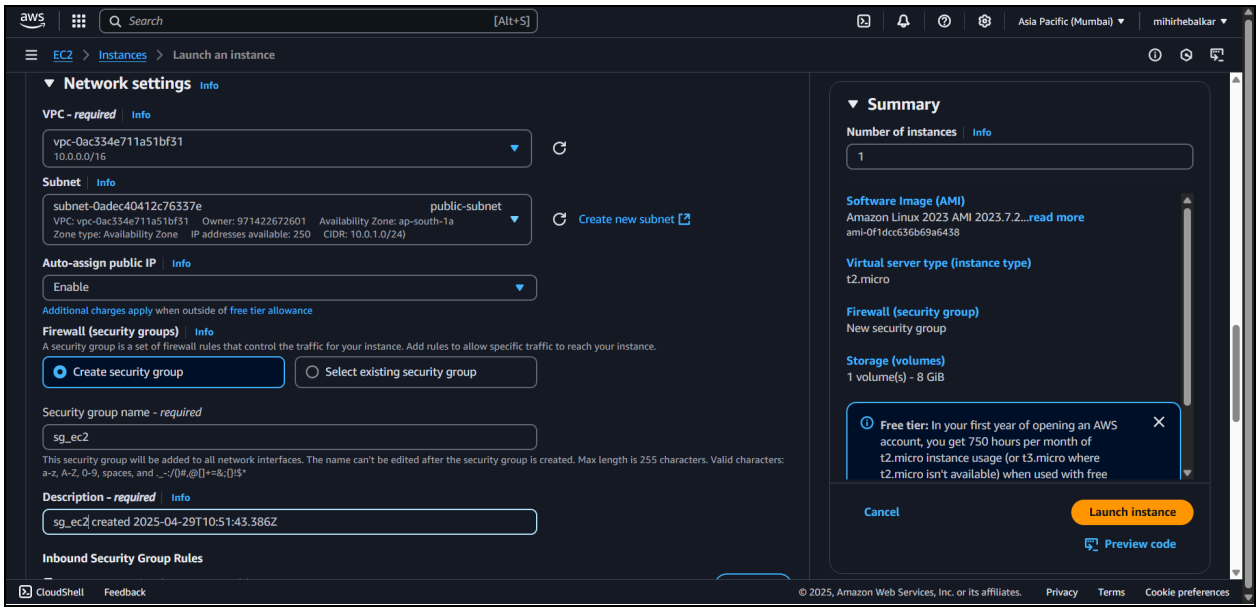


g. Private route table has only local access,Public route table is configure to 0.0.0.0/0 allowing access from anywhere.

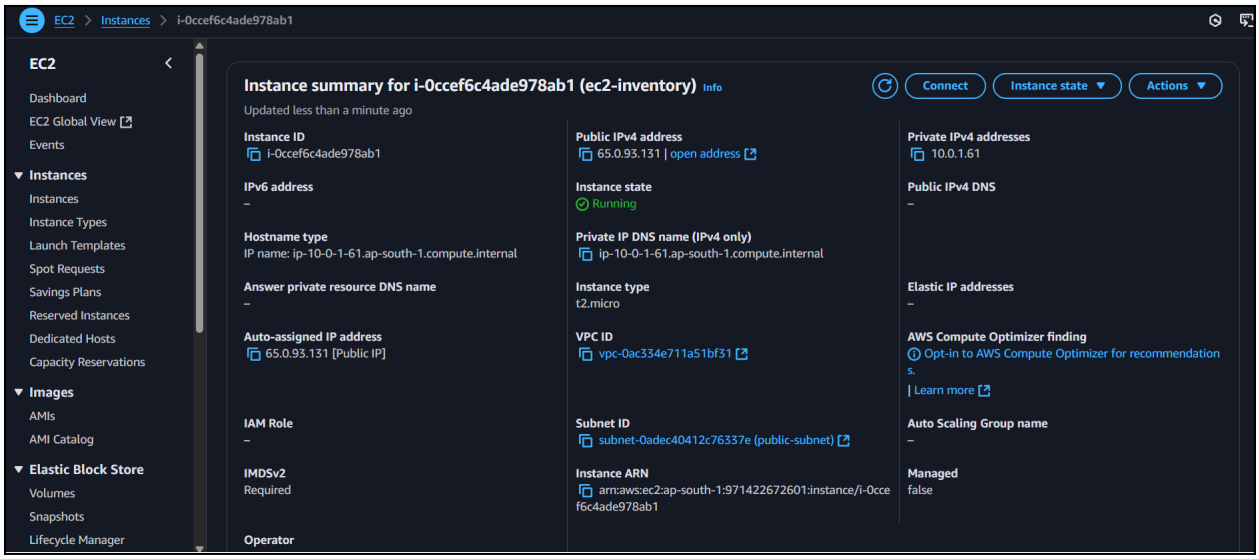
Compute Layer Deployment

4. Ec2 backend, Creating an EC2 Instance

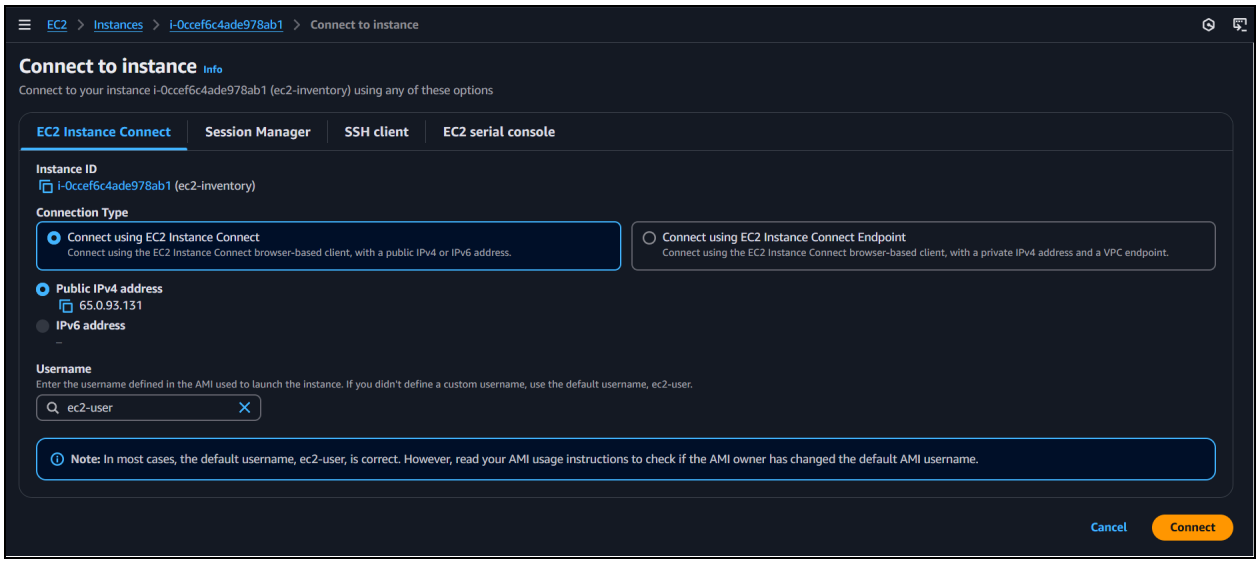
a. Creating an EC2 instance
Selecting the created vpc and public subnet in network settings.
Selected T2.micro for free tier eligibility
Creating an EC2 instance
And Launching Instance



Instance information

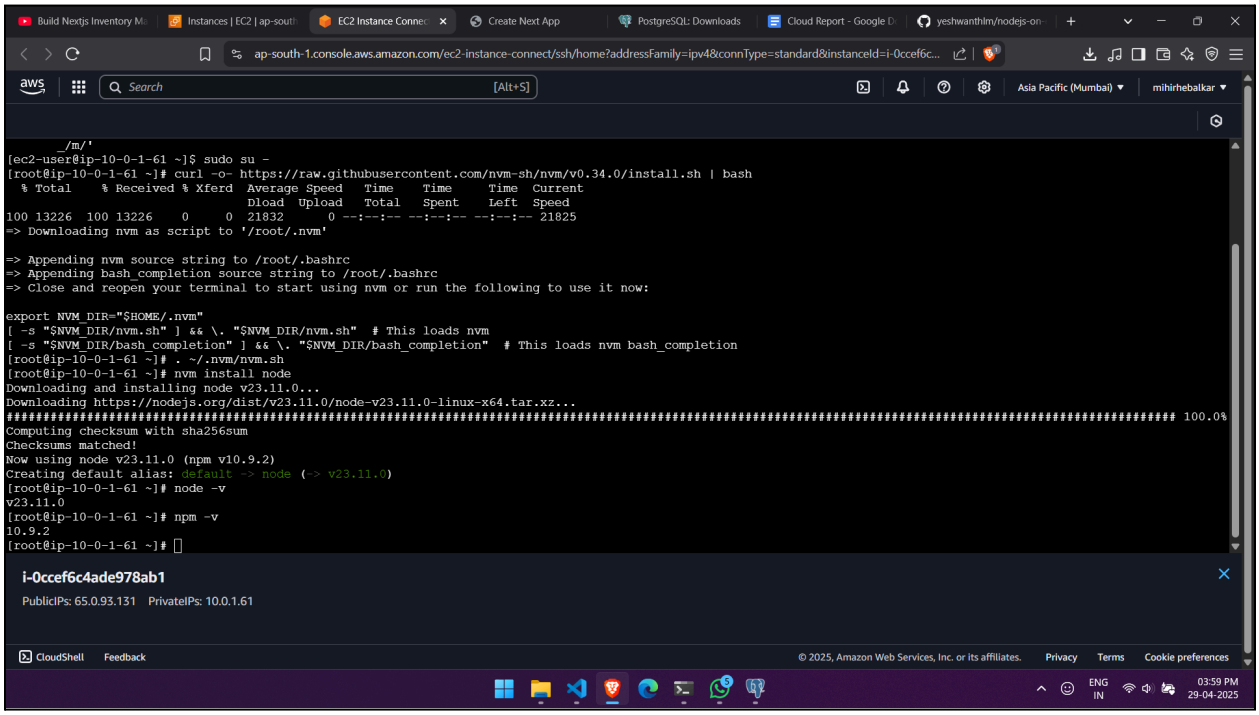


b. Connecting to instance using EC2 Instance Connect



c. Deploying Backend on ec2 instance

Installing node, npm and git



After cloning repo, install packages and run backend on port 80

```
10:38:06 AM - Starting compilation in watch mode...
[0]
[1] Server running on port 80
[0]
[0] 10:38:11 AM - Found 0 errors. Watching for file changes.
[1] 103.197.74.172 - - [29/Apr/2025:10:39:03 +0000] "GET / HTTP/1.1" 404 139
^C[0] npx tsc -w exited with code SIGINT
[1] nodemon --exec ts-node src/index.ts exited with code 130
[root@ip-10-0-1-61 server]# npm i pm2 -g

added 134 packages in 7s
```

Installing pm2 which is a production process manager, particularly for Node.js applications, that helps ensure applications run continuously by automatically restarting them if they crash, and provides features like log management, monitoring, and load balancing

And running backend using pm2

```
[root@ip-10-0-1-61 server]# pm2 start ecosystem.config.js
[PM2] Spawning PM2 daemon with pm2_home=/root/.pm2
[PM2] PM2 Successfully daemonized
[PM2][WARN] Applications inventory-management not running, starting...
[PM2] App [inventory-management] launched (1 instances)
```

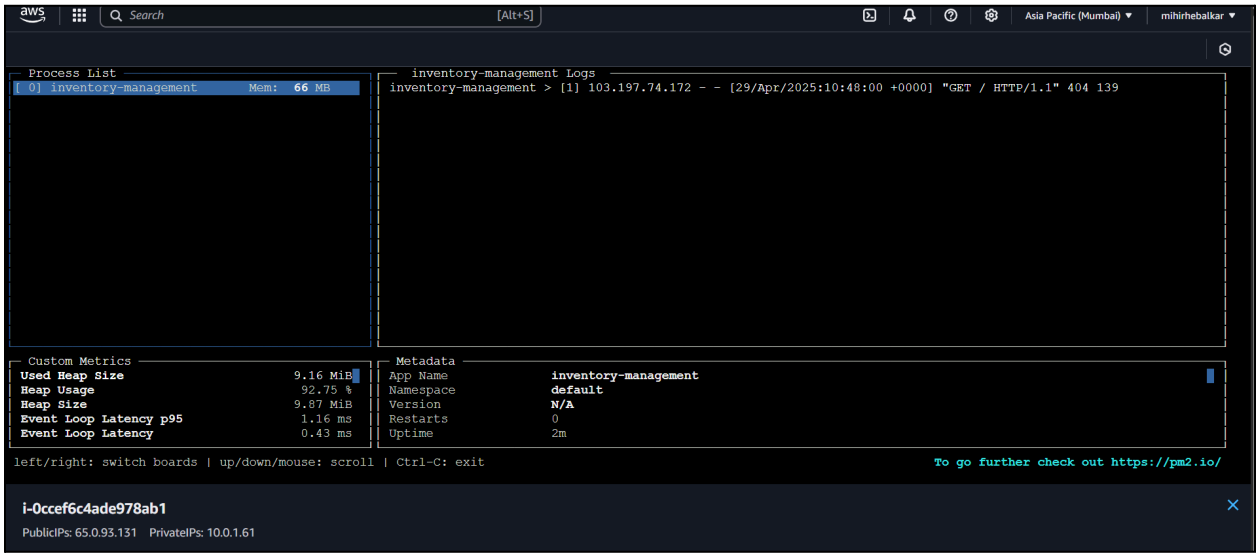
id	name	namespace	version	mode	pid	uptime	↺	status	cpu	mem	user	watching
0	inventory-management	default	N/A	fork	29912	0s	0	online	0%	35.1mb	root	disabled

```
[root@ip-10-0-1-61 server]# pm2 status
```

id	name	namespace	version	mode	pid	uptime	↺	status	cpu	mem	user	watching
0	inventory-management	default	N/A	fork	29912	14s	0	online	0%	68.8mb	root	disabled

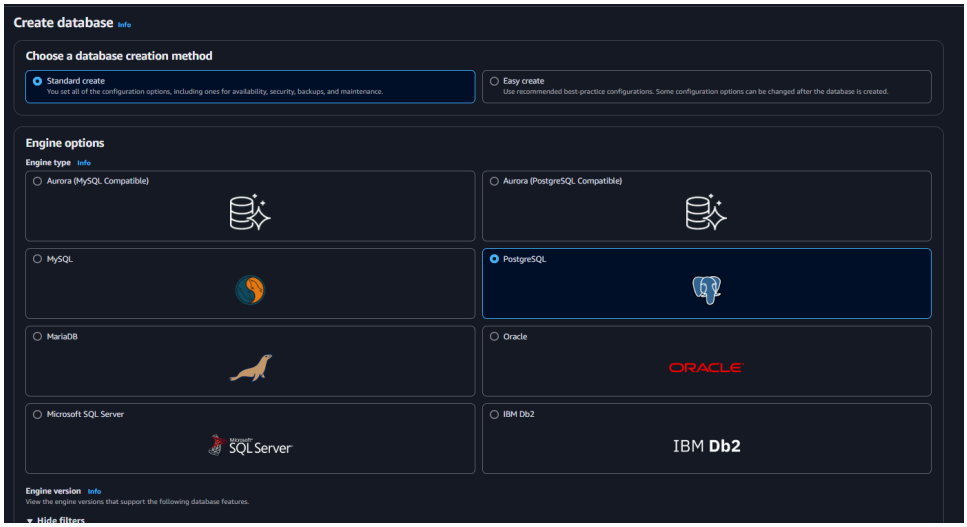
```
[root@ip-10-0-1-61 server]# pm2 monit
```

Now backend has been successfully deployed on ec2 and can be used by the frontend

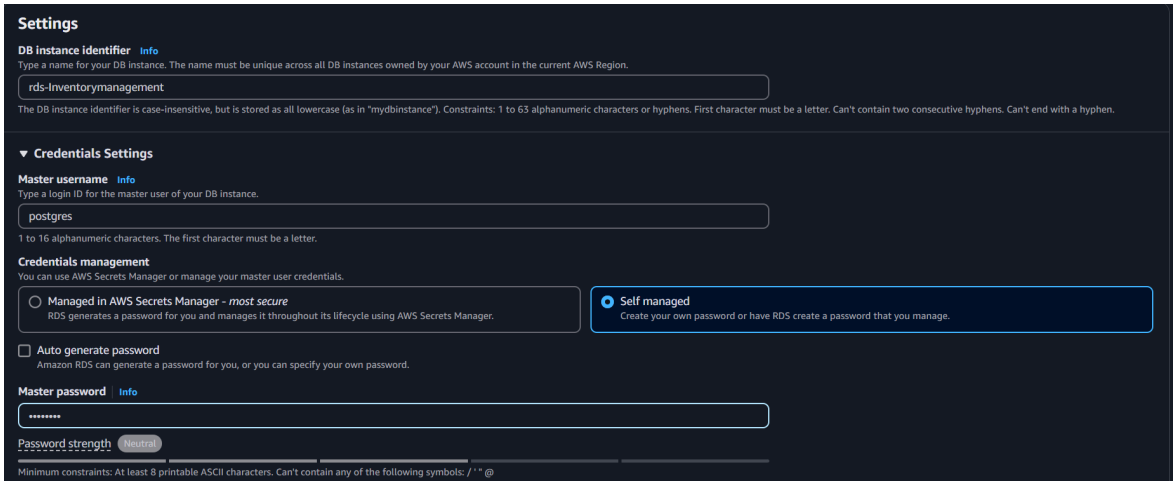


5. Configuring AWS RDS - Relational database

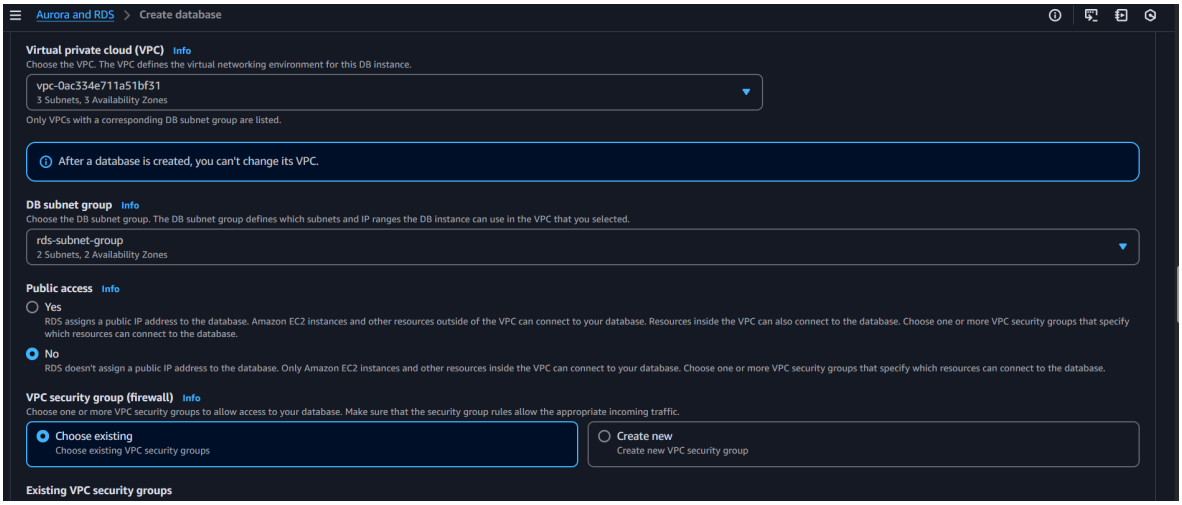
a. Selecting type of database which we have used - postgresql



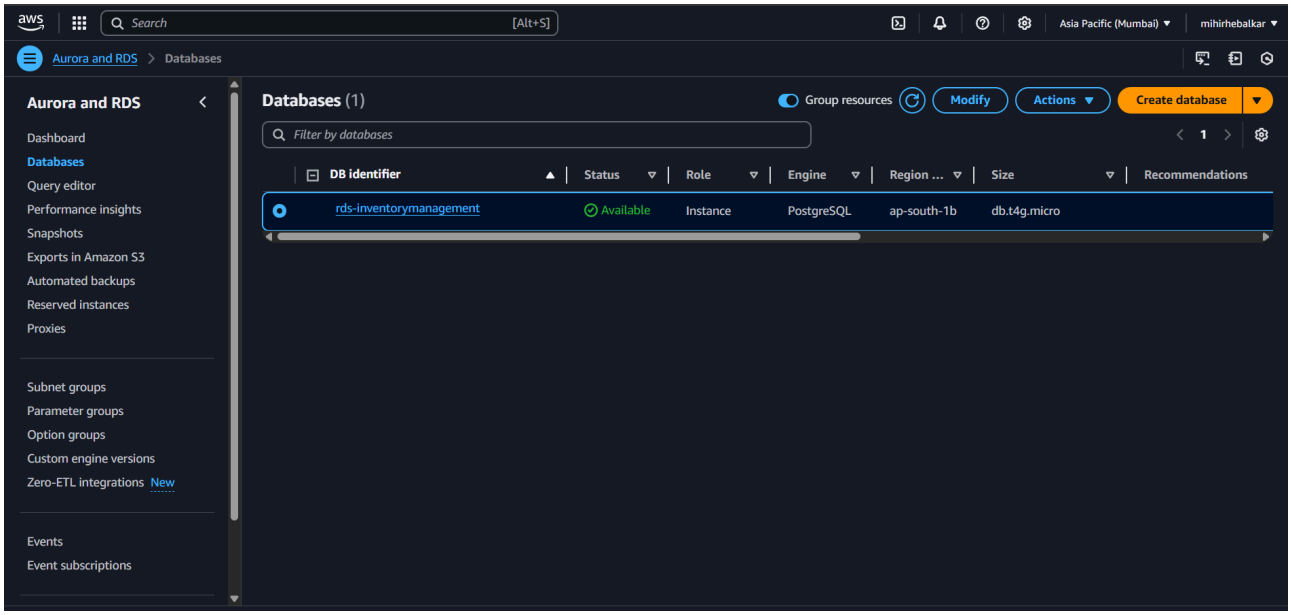
b. Settings Credentials



c. Assigning VPC and Subnet Groups to the DB



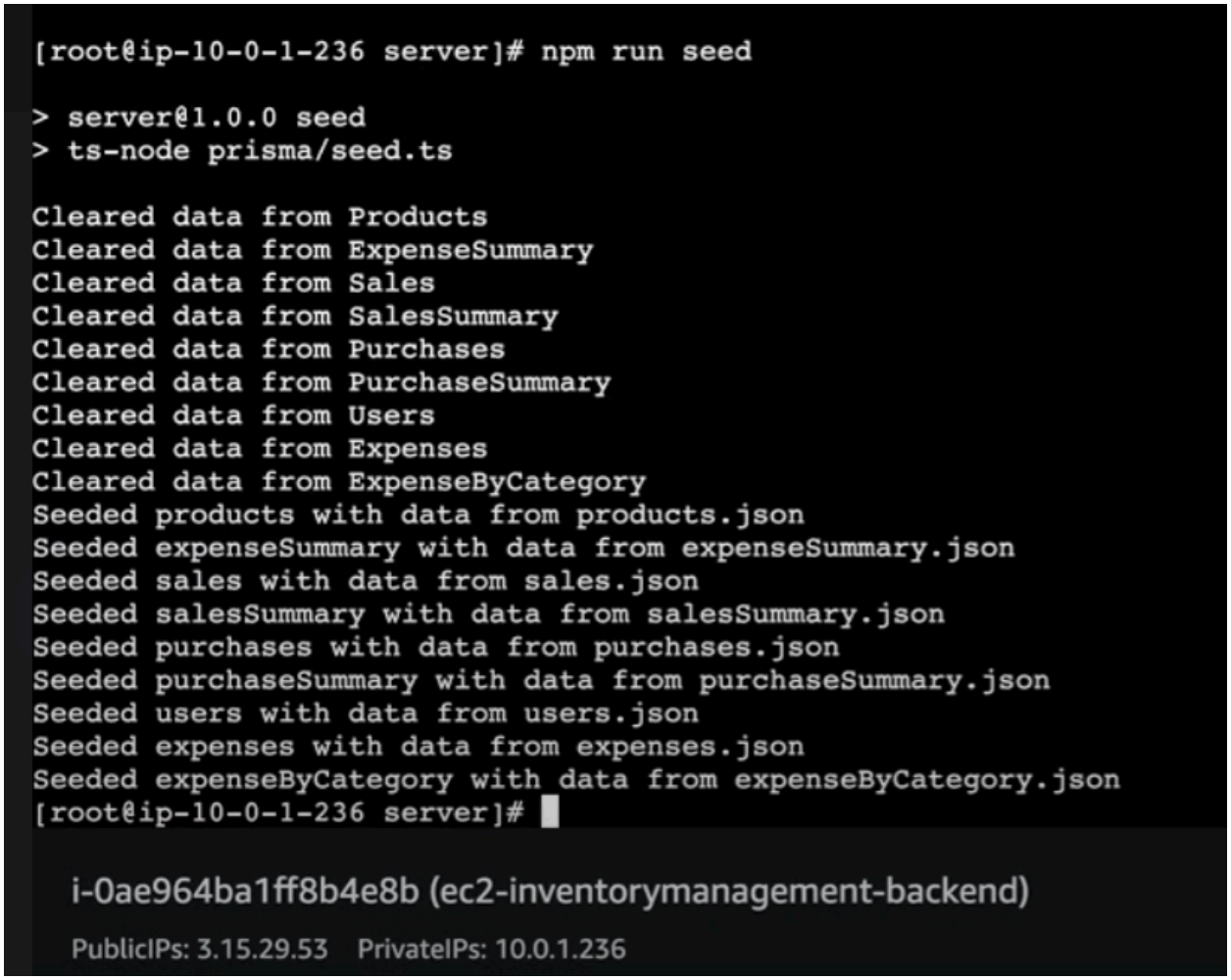
e. Database has been successfully Created



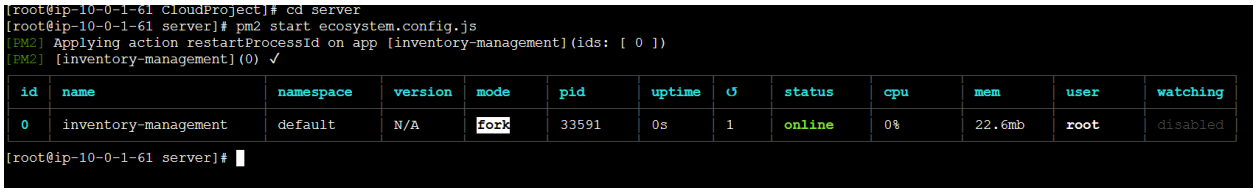
f. Prisma schema is synchronized with your AWS RDS PostgreSQL database (inventorymanagement)

```
[root@ip-10-0-1-236 server]# nano .env
[root@ip-10-0-1-236 server]# npx prisma migrate dev --name init
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": PostgreSQL database "inventorymanagement", schema "public" at "rds-inventorymanagement.ct8ay0em2c8e.us-east-2.rds.amazonaws.com:5432"

Already in sync, no schema change or pending migration was found.

✔ Generated Prisma Client (v5.16.2) to ./node_modules/@prisma/client in 147ms


[root@ip-10-0-1-236 server]# npm run seed
```

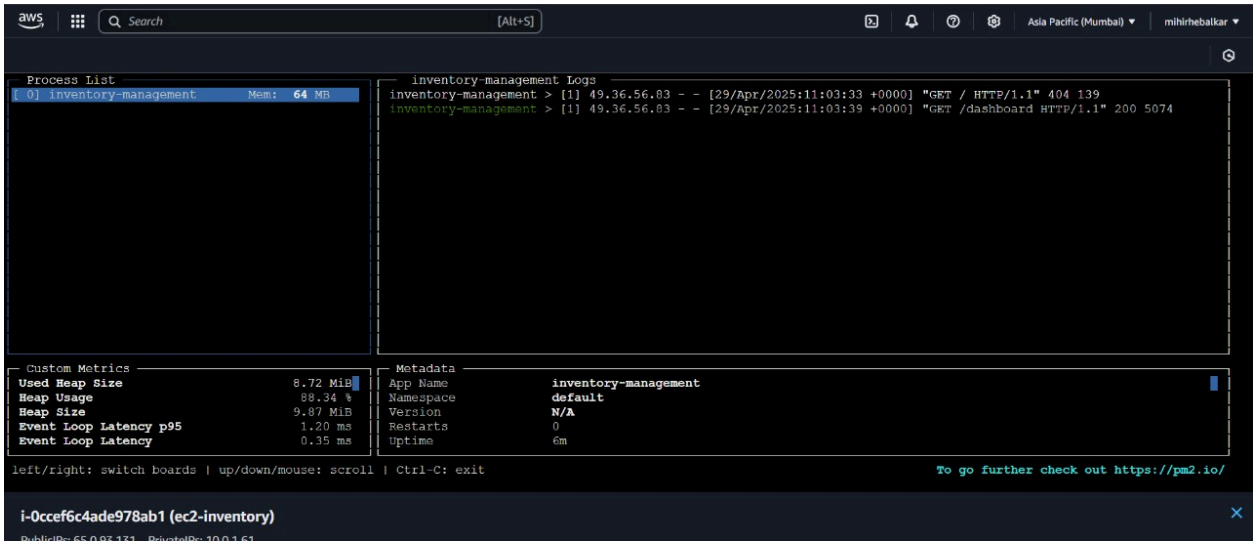
The database was successfully seeded with initial data for all tables (Products, Sales, Purchases, Users, Expenses, etc.) using JSON files



The inventory-management application is running in online status with minimal resource usage (0% CPU, 22.6MB memory) under PM2 process manager.

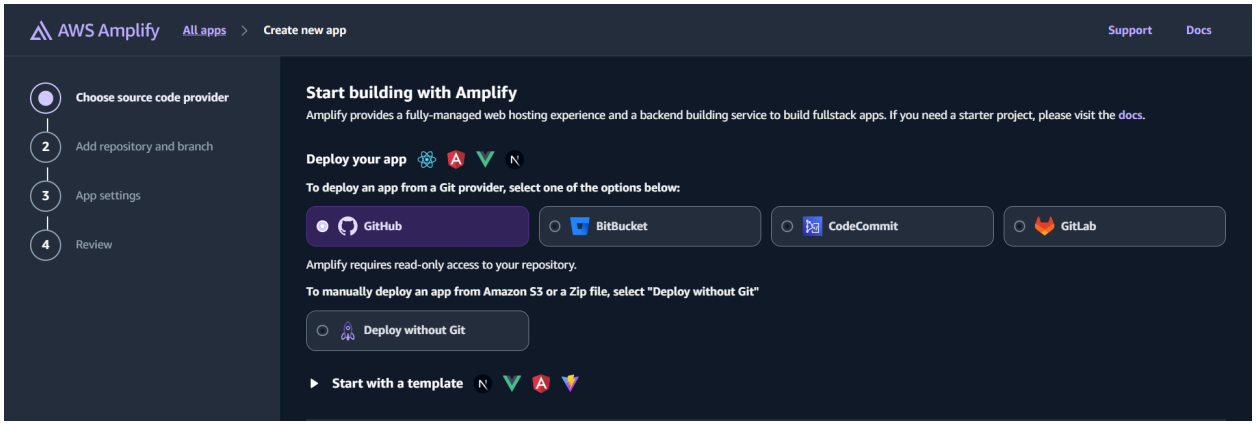


App is running with successful /dashboard access and healthy performance

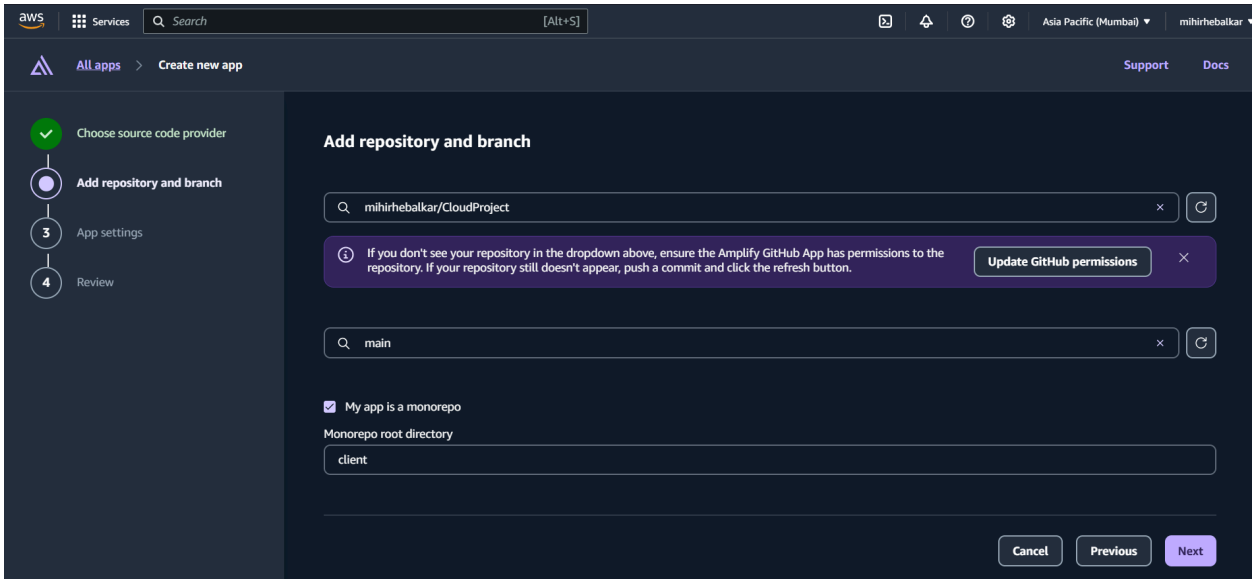


Using http://65.0.93.131 public ec2 IP

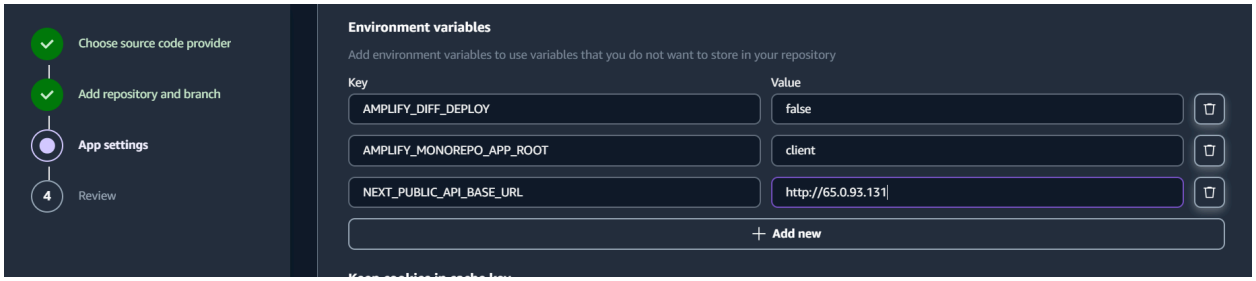
5. Deploying Frontend using AWS Amplify



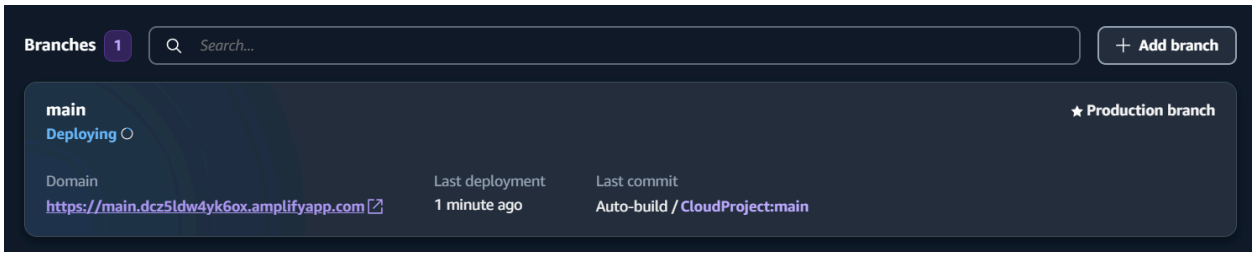
a. Select repo and as monorepo select root directory for frontend



b. Add env to access backend



c. Deploying Nextjs application

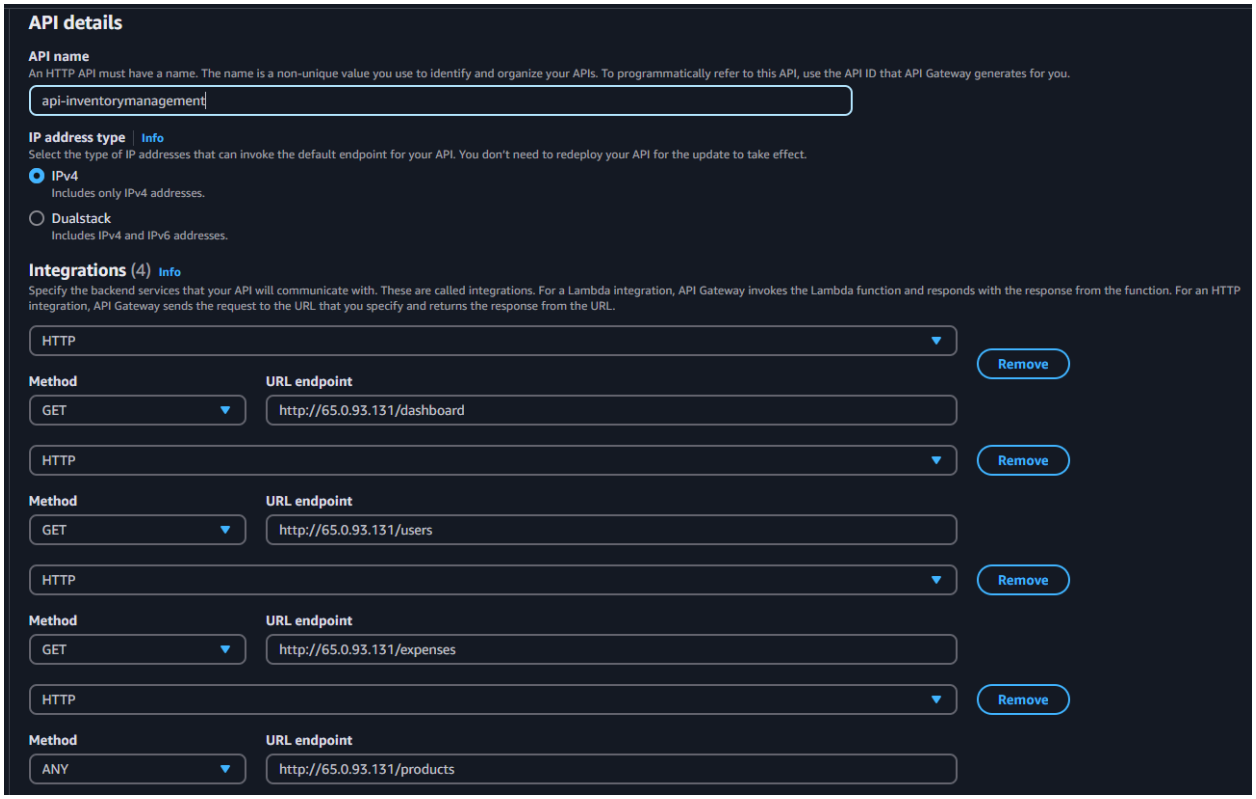


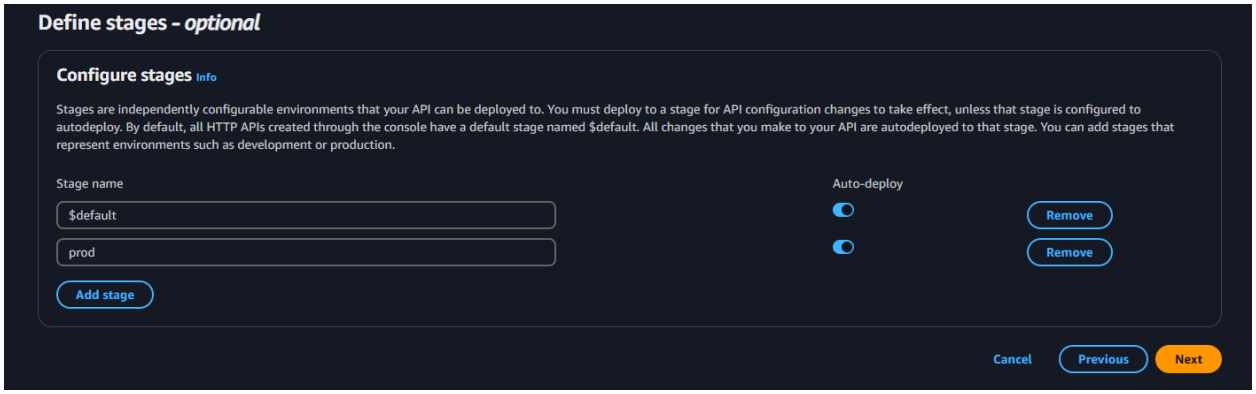
Backend requires http but amplify gives https hence creating api gateway

d. API Gateways

Create http API
Add HTTP integrations for endpoints like

```
/* ROUTES */
app.use("/dashboard", dashboardRoutes); // http://localhost:8000/dashboard
app.use("/products", productRoutes); // http://localhost:8000/products
app.use("/users", userRoutes); // http://localhost:8000/users
app.use("/expenses", expenseRoutes); // http://localhost:8000/expenses

/* SERVER */
const port = Number(process.env.PORT) || 3001;
app.listen(port, "0.0.0.0", () => {
  console.log(`Server running on port ${port}`);
});
```





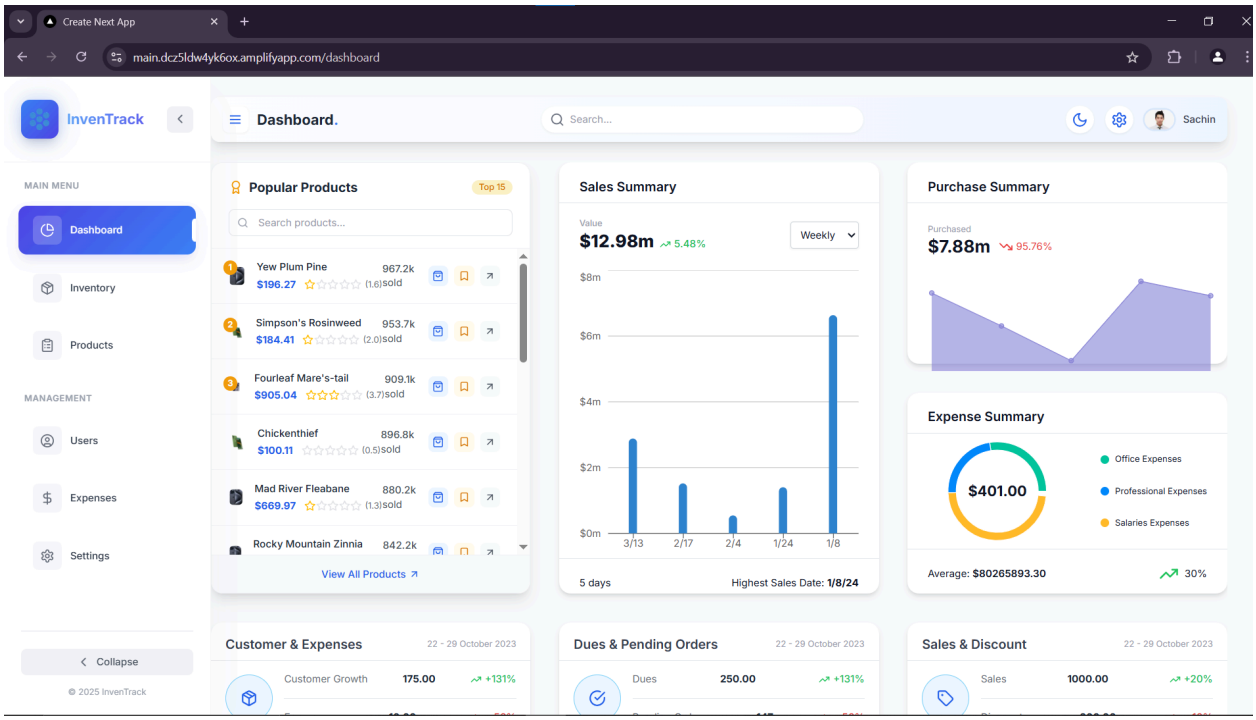
e. Add prod stage with autodeploy

Stages prod -> https://unp7x0fnp0.execute-api.ap-south-1.amazonaws.com/prod

f. Add into env of Amplify

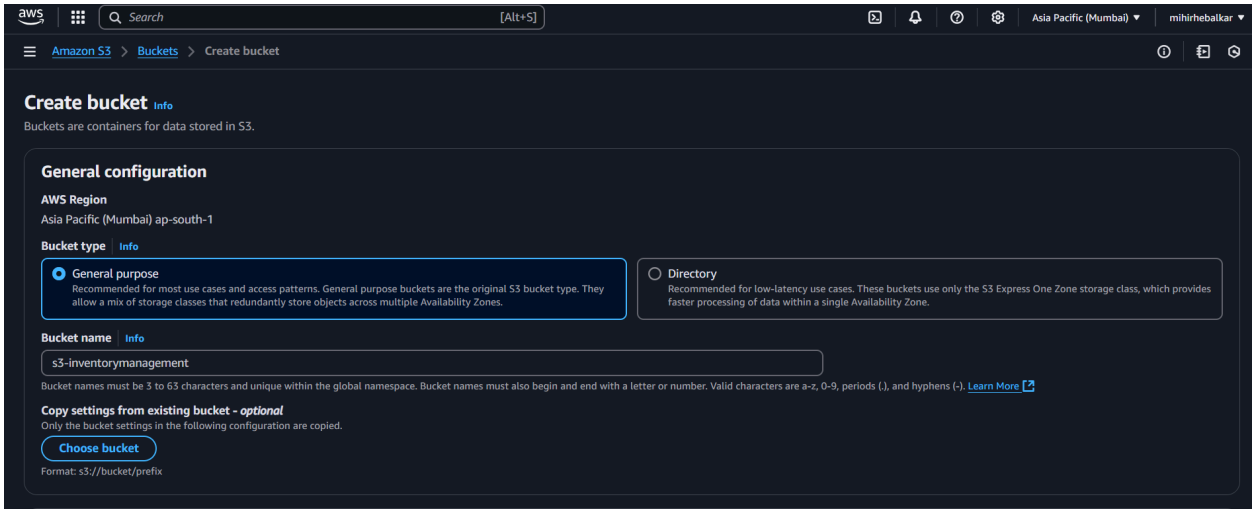
Hosting -> Environment variables -> Manage variables

Redeploy

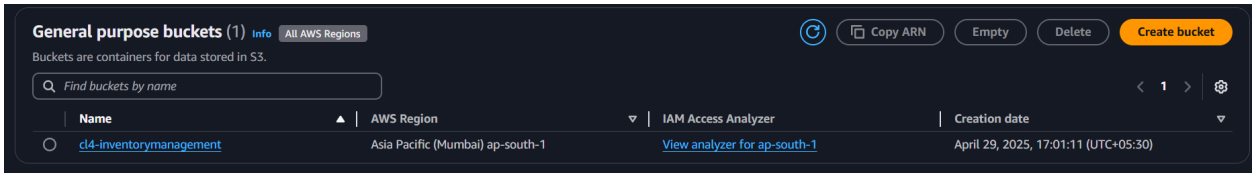


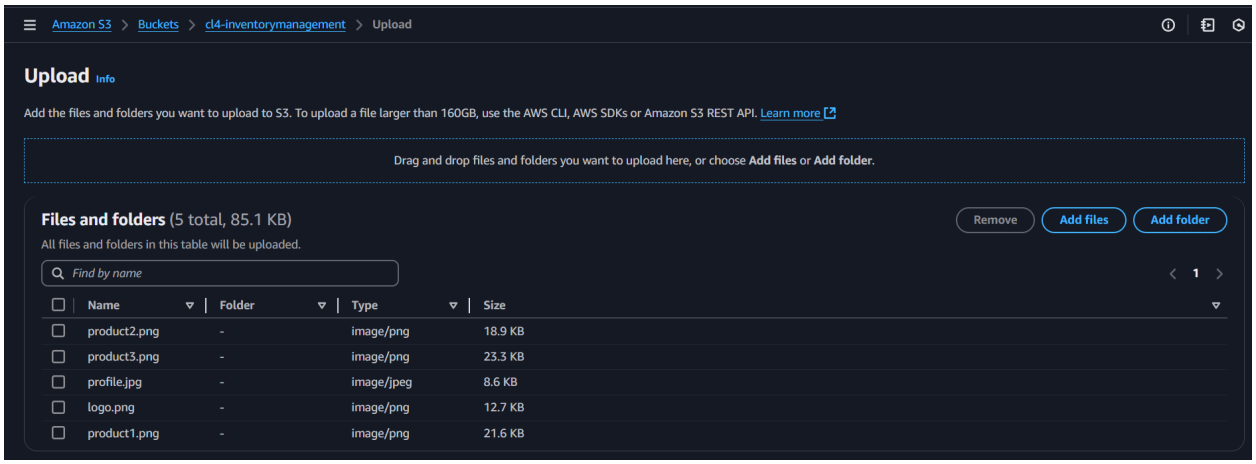
6. S3 Bucket for Files storage

a. Create s3 bucket



b. Bucket has been successfully created



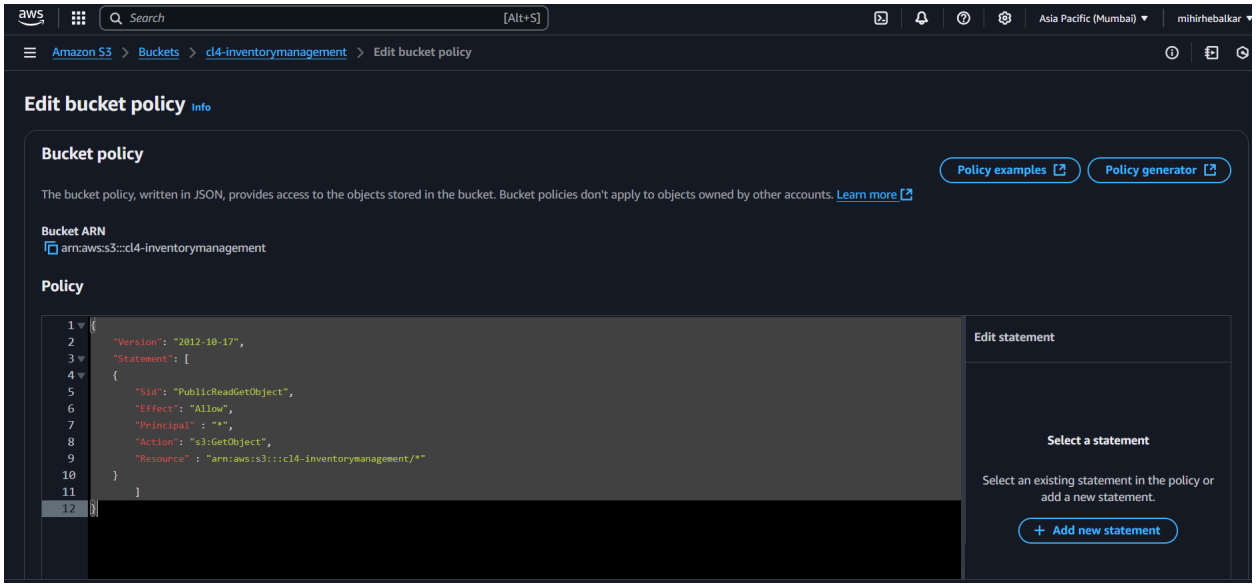


c. Upload assets

d. Edit bucket policy

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Sid": "PublicReadGetObject",
        "Effect": "Allow",
```

```
        "Principal" : "*",
        "Action": "s3:GetObject",
        "Resource" : "arn:aws:s3:::cl4-inventorymanagement/*"
    }
        ]
}
```



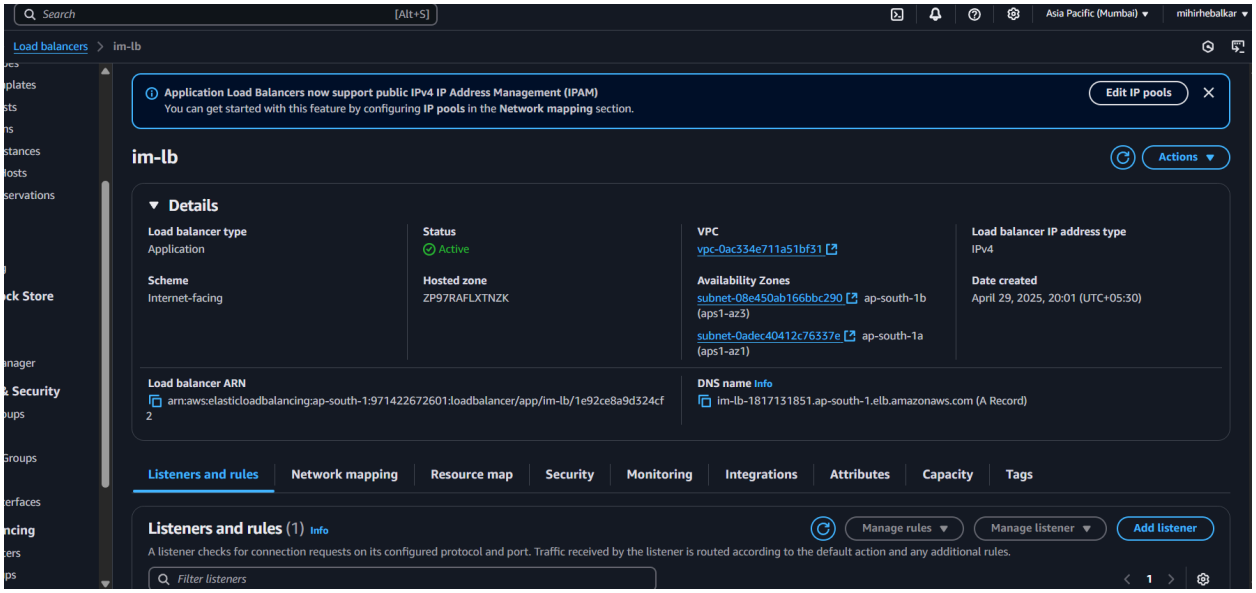
e. Now object URLs are visible

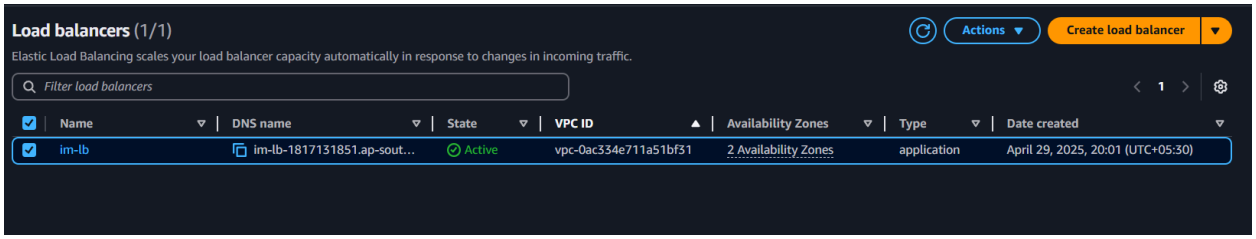
https://cl4-inventorymanagement.s3.ap-south-1.amazonaws.com/

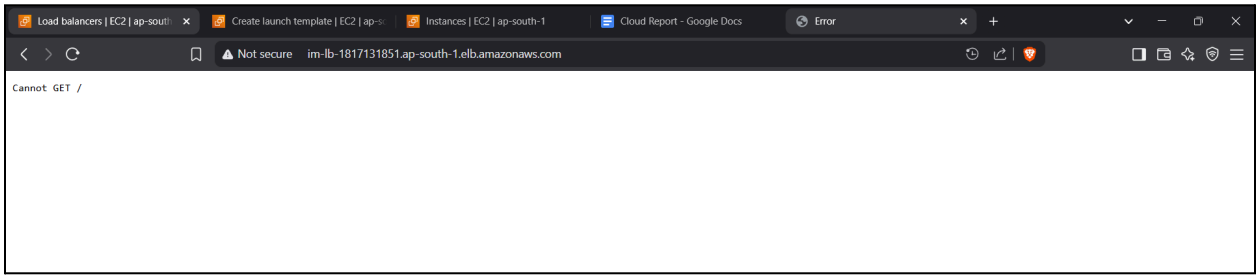
f. Update next.config.mjs to allow traffic from s3 bucket

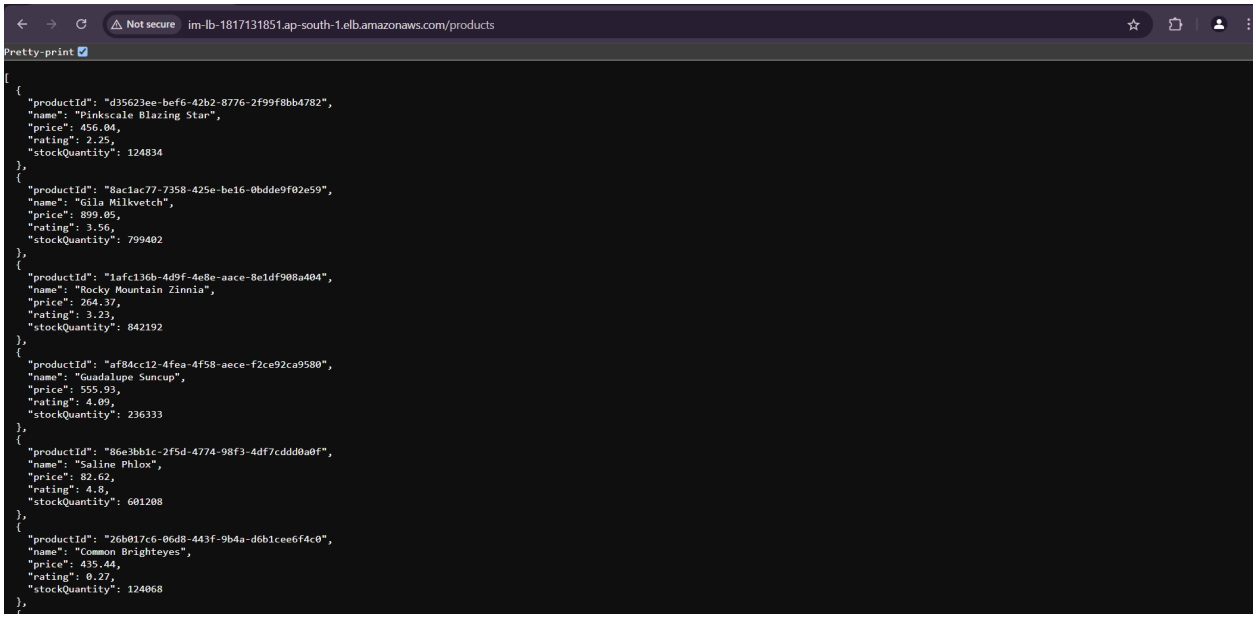
7. Load balancing

Creating load balancer,
Selected current vpc and 2 availability zones
Ap-south-1b and ap-south-1a for 2 separate instances which run our backend, now load balancer will choose instance based on availability and balance load





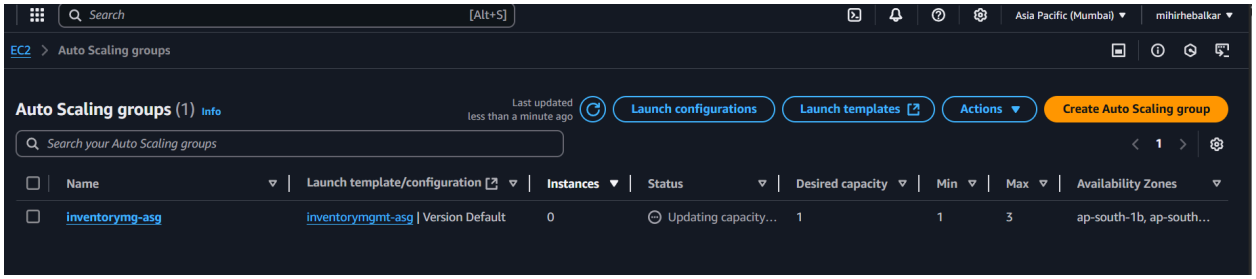


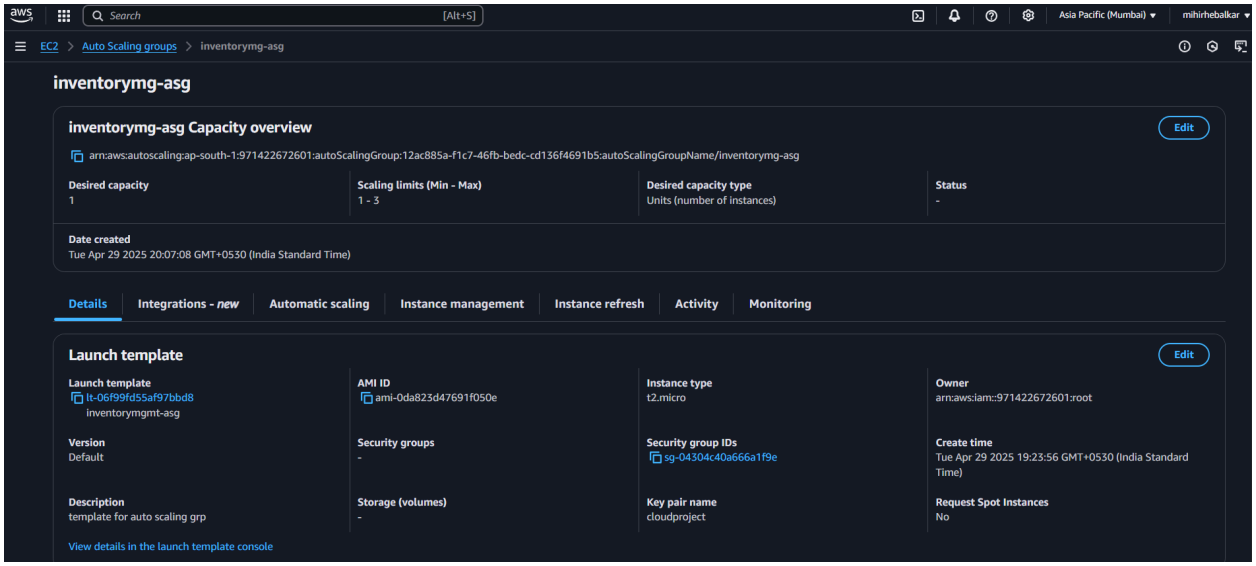


Now aws amplify can use the load balancer url im-lb.. (inventory management)

8. Auto Scaling

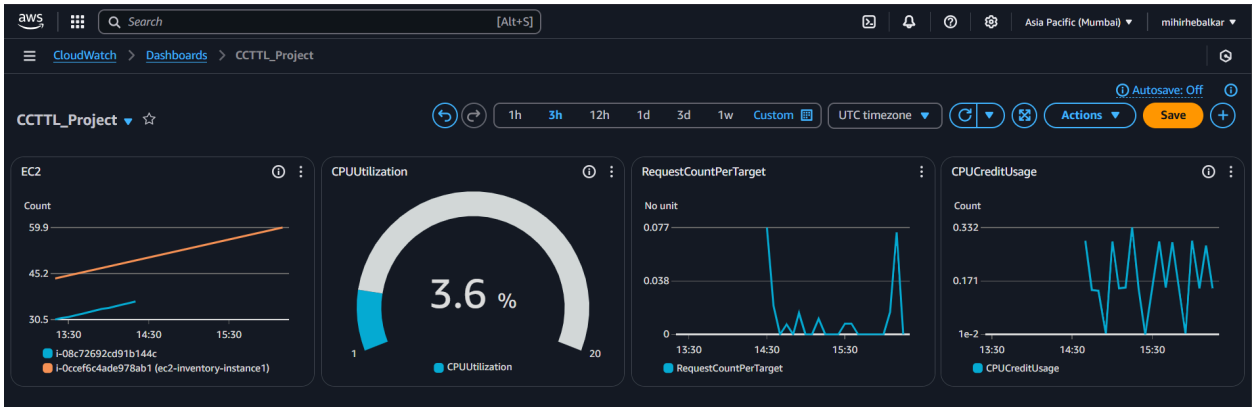
a. Created autoscaling group and used existing load balancer





9. CloudWatch implementation

Implemented CloudWatch dashboard which shows EC2 instance utilization and CPU utilization, along with it our cloudwatch dashboard shows Request count per target and CPU Credit Usage. Implementing this allows us to monitor usage of our project on AWS



Conclusion

The report discusses a very strong and up-to-date cloud-native implementation for inventory management. By strategically utilizing AWS cloud services coupled with an infrastructure that is entirely cloud-based, the system efficiently solves shortcomings of traditional inventory systems and provides significant benefits in operations. The benefits of the architecture include:

- Elastic scalability – Enabling the system to automatically cope with different workloads, assuring optimal performance at peak and off-peak periods.
- High availability – By ensuring that the system will be resilient and have a minimum failure time.
- Enhanced security – Follow best practices for network isolation, access control, and data protection to maintain system integrity and compliance.
- Continuous deployment – Rapid, reliable updates are achieved via CI/CD pipelines, reducing the time taken for new features and fixes to be deployed.
- Comprehensive monitoring – Includes state of the art monitoring and alerting tools for proactive management and keeping problems at bay.

In short, this solution serves as a reliable, scalable, and secure infrastructure, founded on microservices for inventory management, which can be used for both vibrant and growing business environments.