SnapStream: AWS-Powered Media Upload and Analysis Platform

Project Description:

At a time when digital content creation and media management are rapidly growing, organizations often struggle with storing, processing, and analyzing large volumes of multimedia data efficiently. To address these challenges, the development team created SnapStream — an AWS-Powered Media Upload and Analysis Platform designed to automate and streamline end-to-end media handling.

SnapStream leverages a Flask-based backend for managing user interactions and API integrations, while AWS EC2 provides scalable hosting for the web application. Uploaded media files — including images, audio, and video — are securely stored in Amazon S3, with each upload triggering AWS Lambda for serverless, event-driven processing. The platform integrates AI/ML services such as Amazon Rekognition, Transcribe, and Comprehend to automatically extract metadata, generate transcripts, and perform intelligent content analysis.

All metadata and analysis results are stored in AWS DynamoDB, ensuring fast, reliable, and schema-flexible data retrieval. AWS SNS provides real-time notifications to users about upload and processing status, while AWS IAM roles ensure secure and role-based access to all cloud resources.

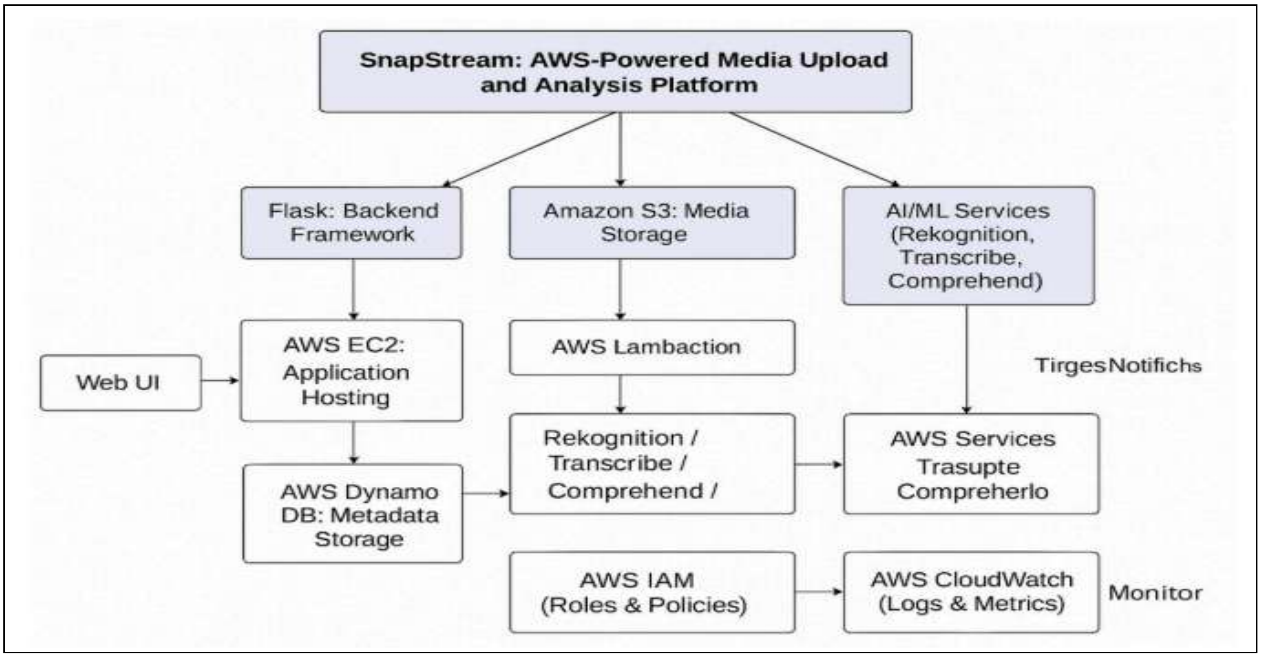
Built with a responsive HTML/CSS frontend, SnapStream delivers a smooth and intuitive user experience, enabling users to upload, manage, and analyze their media effortlessly. This cloud-native platform provides a cost-effective, scalable, and intelligent solution for modern media workflows—reducing manual effort and empowering users with deeper insights into their multimedia content.

Scenario 1: Media Archiving for a News Agency

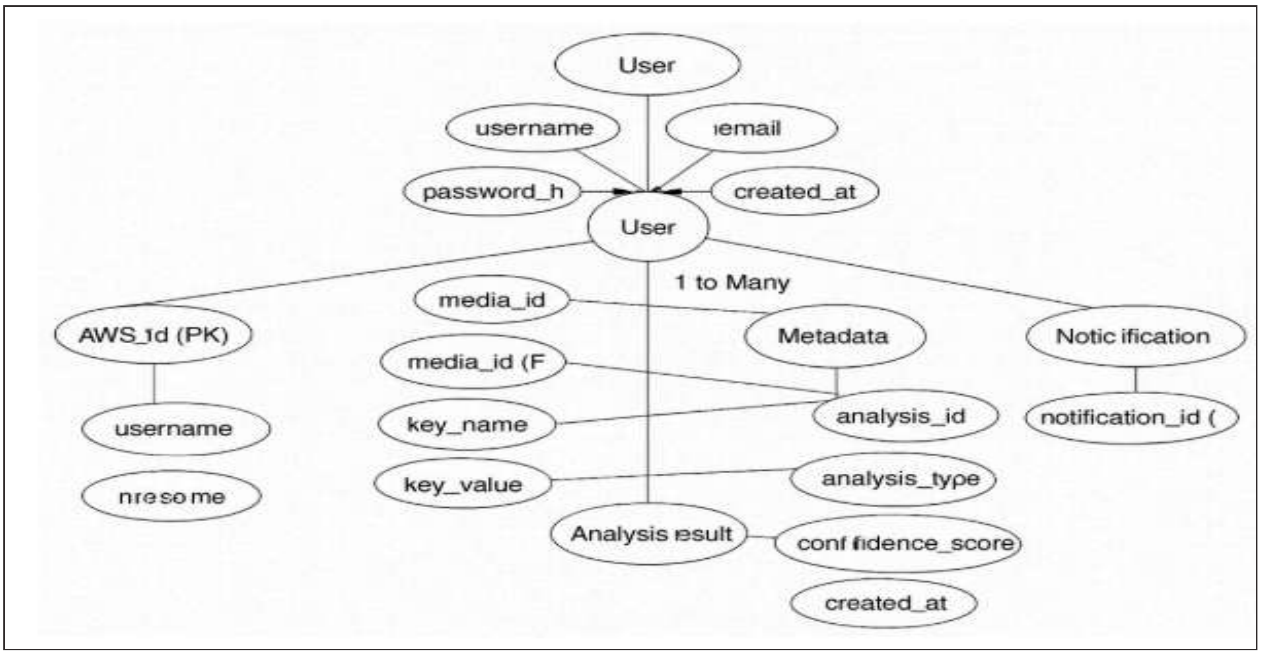
A leading digital news agency leverages SnapStream to manage its extensive library of daily news footage. Each time a journalist uploads a video, it is securely stored in Amazon S3, and an AWS Lambda function is automatically triggered to generate preview thumbnails for quick browsing. Using Amazon Rekognition, the platform detects key visual elements such as people, logos, and text appearing in the footage. This automated tagging system allows reporters to search for specific events, individuals, or topics with ease. What once required manual sorting and labeling is now completed in seconds, significantly improving workflow efficiency and freeing up time for content creation and reporting.

.

Architecture



Entity Relationship (ER)Diagram



Pre-requisites

1.AWS Account Setup: AWS Account Setup

2.Understanding IAM: _ IAM Overview

3.Amazon EC2 Basics:_ EC2 Tutorial

4.DynamoDB Basics: DynamoDB Introduction

5.Git Version Control: Git Documentation

Project WorkFlow

1. AWS Account Setup and Login

Activity 1.1: Set up an AWS account if not already done.

Activity 1.2: Log in to the AWS Management Console

2. DynamoDB Database Creation and Setup

Activity 2.1: Create a DynamoDB Table.

Activity 2.2: Configure Attributes for User Data and Book Requests.

3.SNS Notification Setup

Activity 3.1: Create SNS topics for sending email notifications to users and library staff

Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time notifications

4. Backend Development and Application Setup

Activity 4.1:Develop the Backend Using Flask.

Activity 4.2: Integrate AWS Services Using boto3.

5. EC2 Instance Setup

Activity 5.1: Launch an EC2 instance to host the Flask application.

Activity 5.2: Configure security groups for HTTP, and SSH access.

6. Deployment on EC2

Activity 6.1:Upload Flask Files

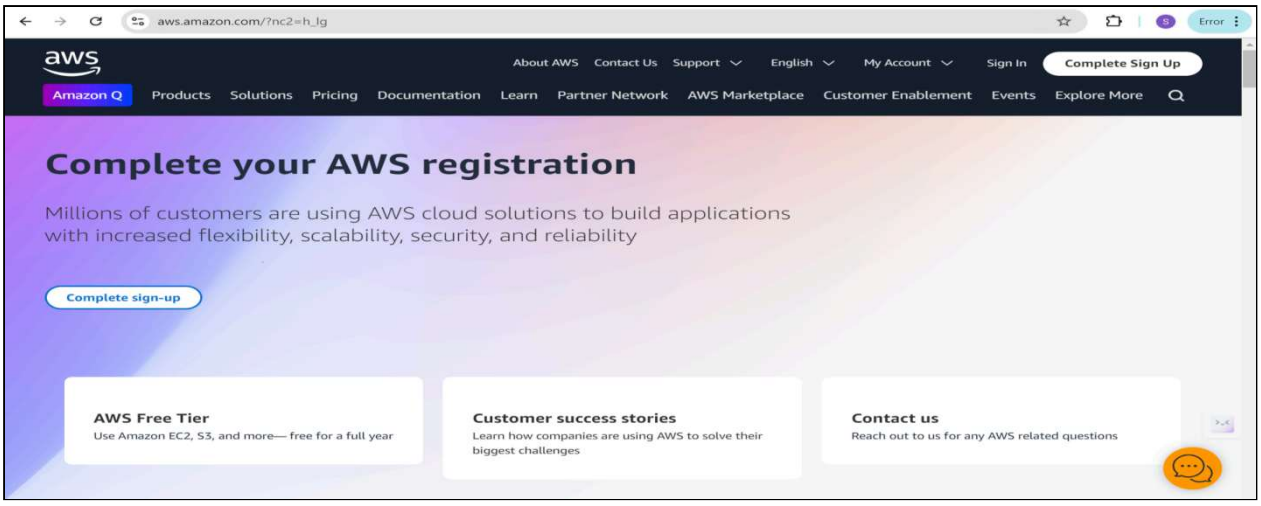
Activity 6.2: Run the Flask App

7. Testing and Deployment

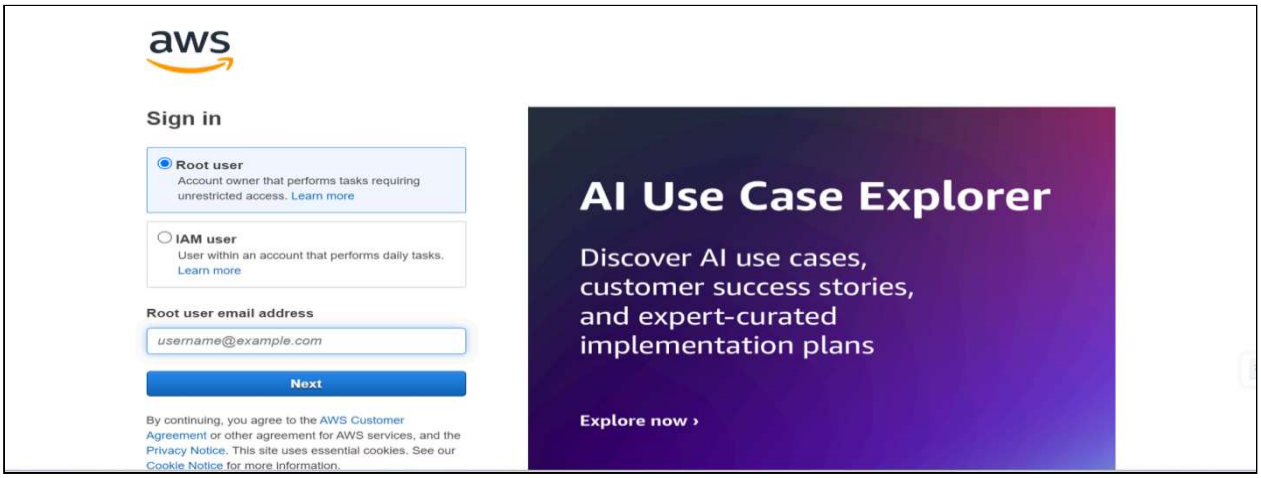
Activity 7.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

Milestone 1: AWS Account Setup And Login

Activity 1.1: Set up an AWS account if not already done.

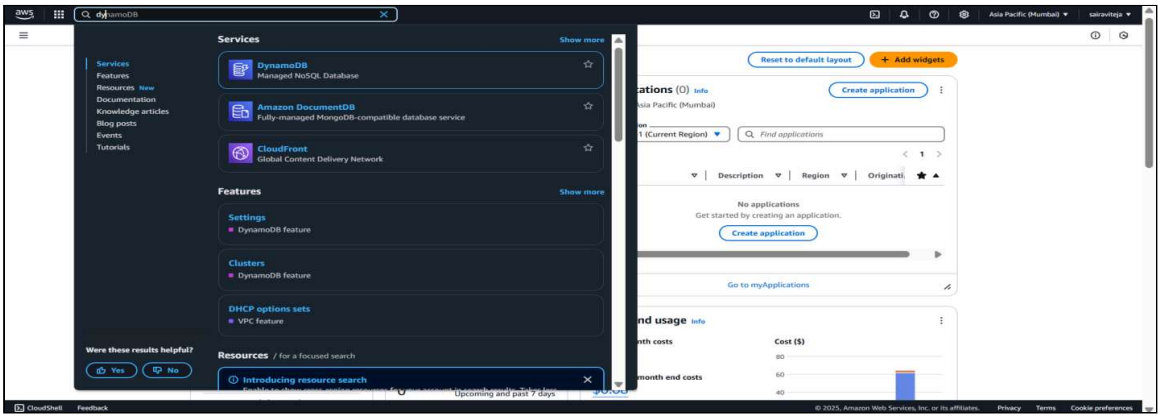


Activity 1.2: Log in to the AWS Management Console

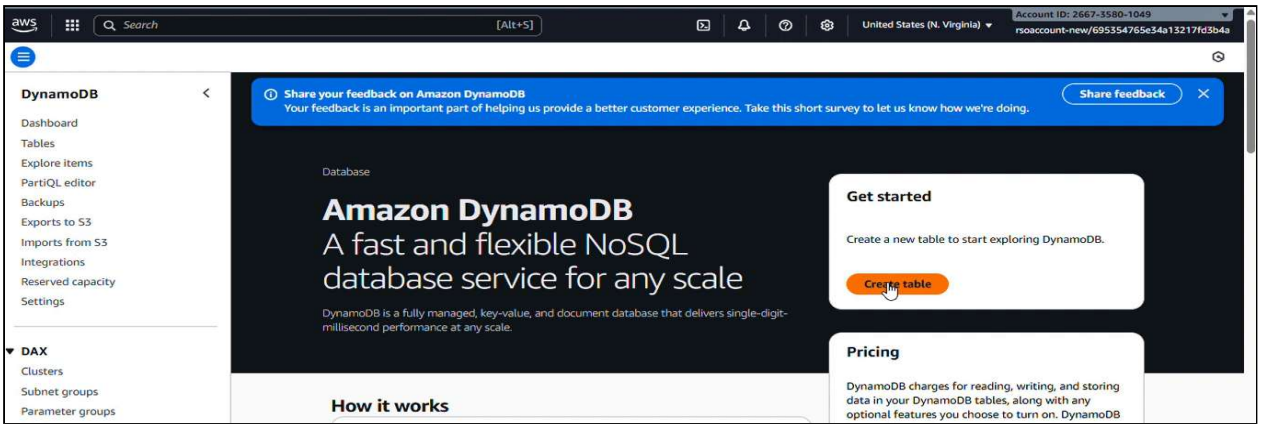


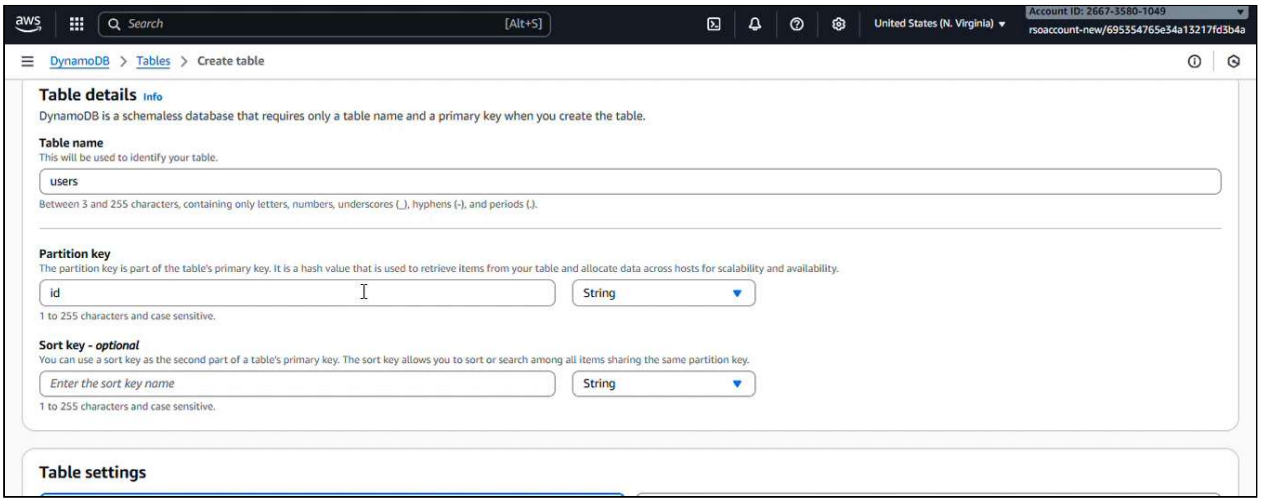
Milestone 2: DynamoDB Database Creation and Setup

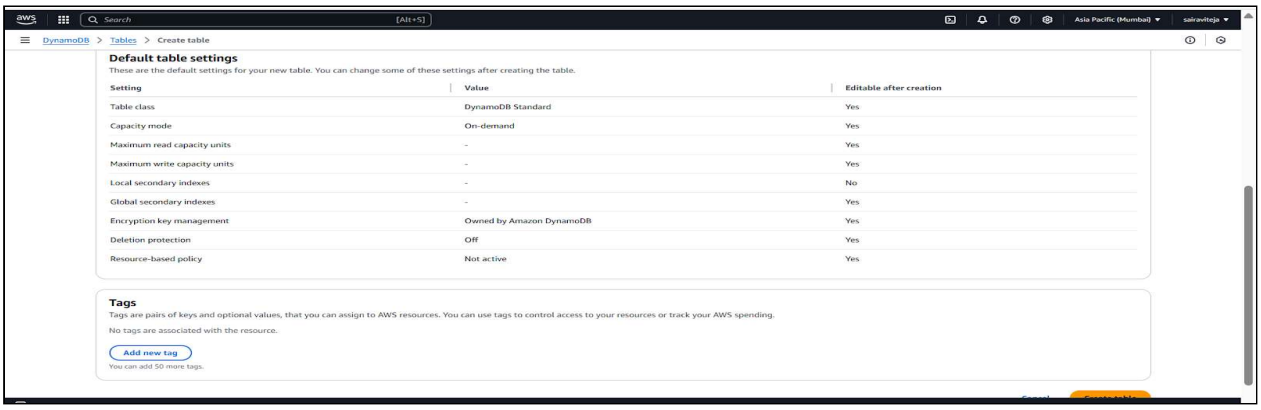
Activity 2.1: Create a DynamoDB Table.

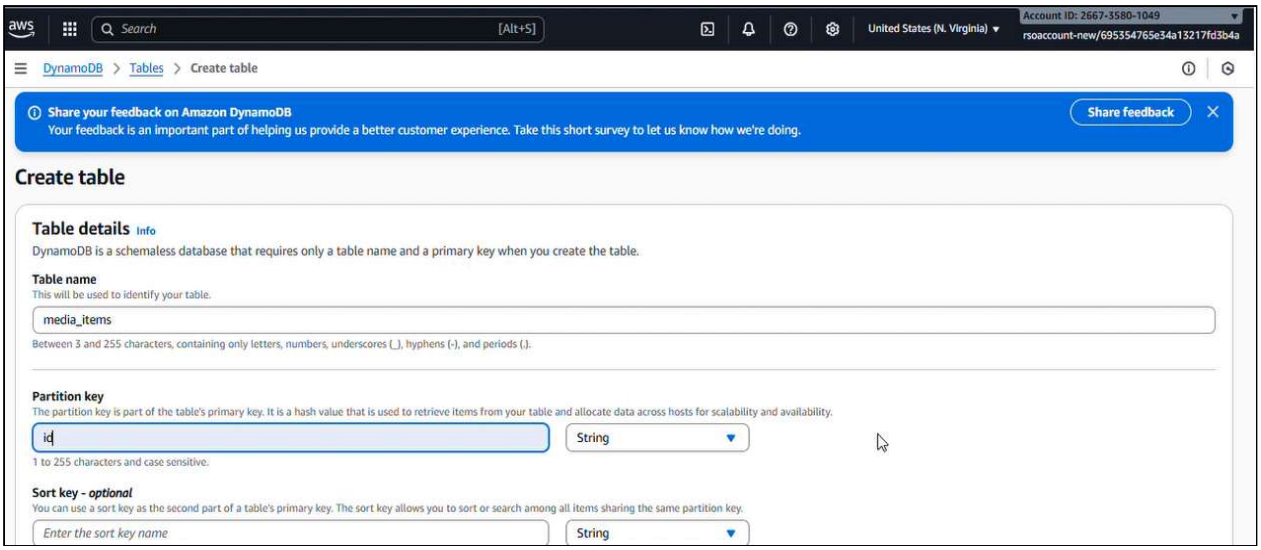


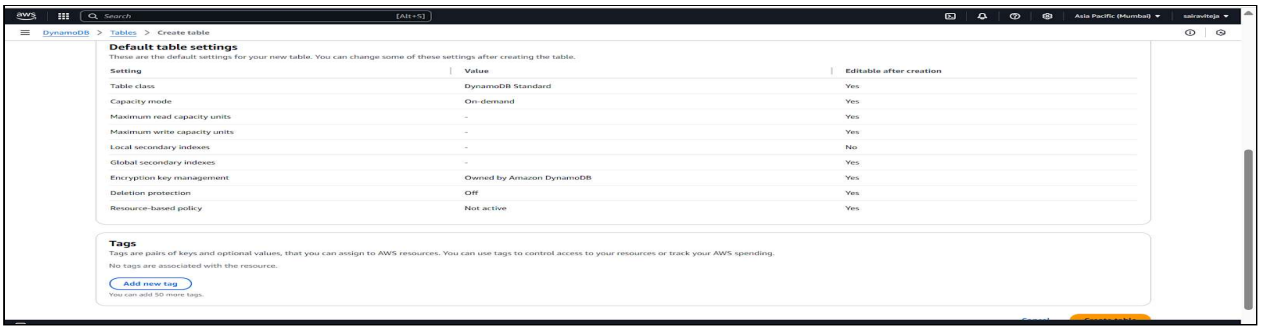
Activity 2.2: Configure Attributes for User Data and Book Requests.





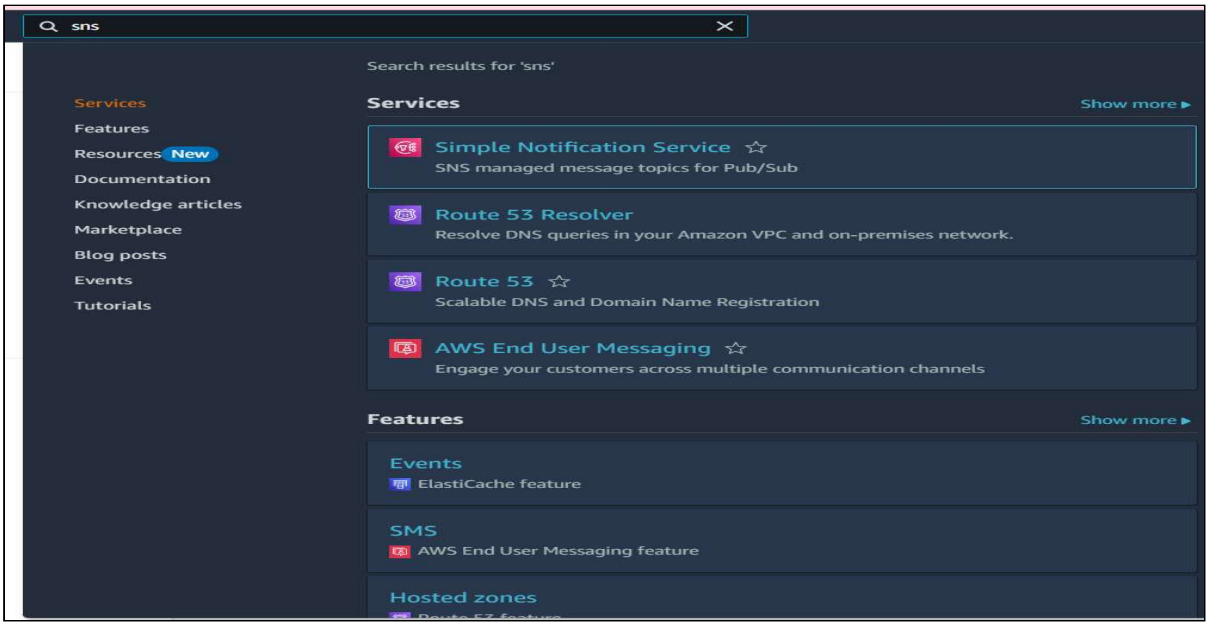




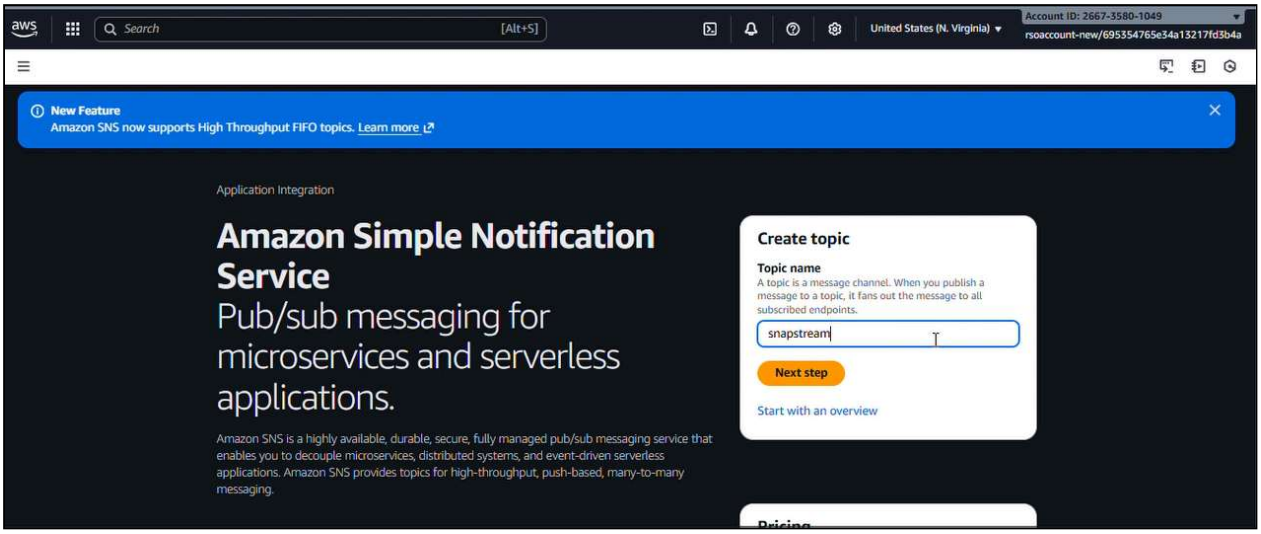


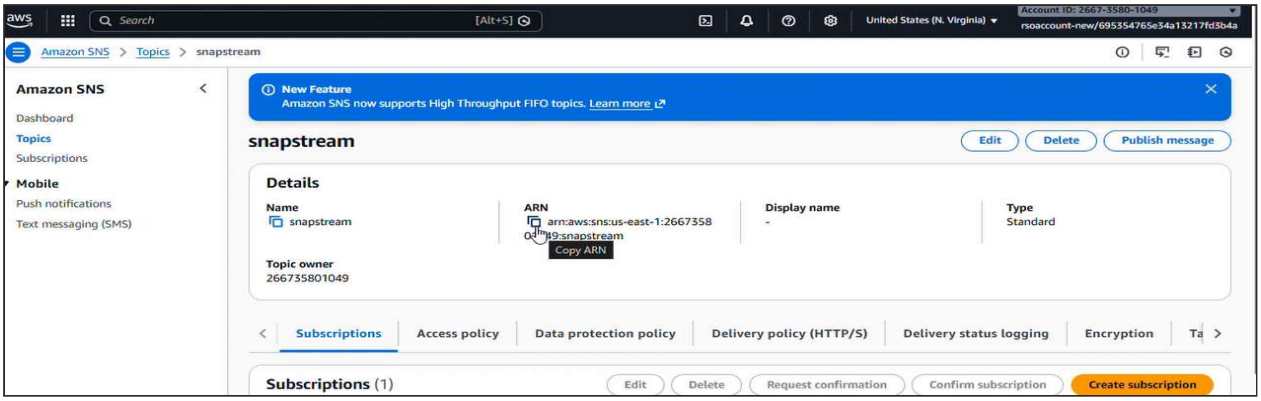
Milestone 3: SNS Notification Setup

Activity 3.1: Create SNS topics for sending email notifications to users and library staff.



Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time

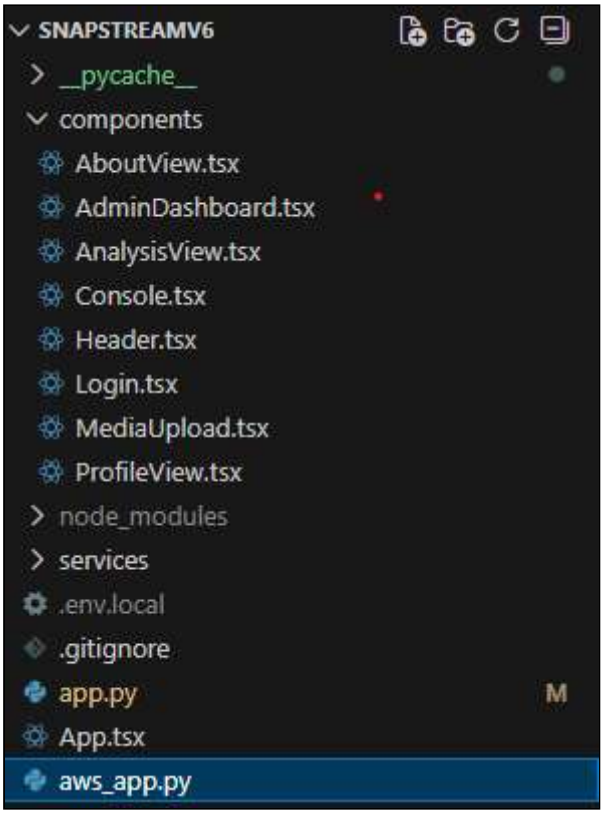




Milestone 4: Backend Development and Application Setup

Activity 4.1:Develop the Backend Using Flask

- **File explorer**



Activity 4.2: Integrate AWS Services Using boto3.

```
aws_app.py > ...
1   from flask import Flask, request, jsonify
2   from flask_cors import CORS
3   import boto3
4   import uuid
5   import json
6   import os
7   from boto3.dynamodb.conditions import Key, Attr
8   from botocore.exceptions import ClientError
```

Description: This project uses Flask for routing, session management, and user authentication with secure password hashing. It integrates AWS services via Boto3 for handling data storage,

notifications, and unique user operations

```
app = Flask(__name__)
```

Description: A new Flask application instance is initialized, and a secret key is set to securely manage user sessions and protect against cookie tampering.

- **DynamoDB Setup**

```
# --- AWS Configuration ---
REGION = 'us-east-1'
SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:604665149129:aws_capstone_topic'

# Initialize AWS Resources
# In EC2, this uses the IAM Role. Locally, it uses ~/.aws/credentials.
dynamodb = boto3.resource('dynamodb', region_name=REGION)
sns = boto3.client('sns', region_name=REGION)

# Table Definitions (Match the 'id' partition keys from your app.py logic)
users_table = dynamodb.Table('users')
media_table = dynamodb.Table('media_items')
```

- **Routes for web pages**

Register User: Collecting registration data, hashes the password, and stores user details in the database.

```
@app.route('/api/register', methods=['POST'])
def register():
    data = request.json
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    user_id = str(uuid.uuid4())
    try:
        cursor.execute(
            "INSERT INTO users (id, name, email, password, role) VALUES (?, ?, ?, ?, ?)",
            (user_id, data['name'], data['email'], data['password'], data['role'])
        )
        conn.commit()
        return jsonify({"id": user_id, "name": data['name'], "email": data['email'], "role": data['role']}), 201
    except sqlite3.IntegrityError:
        return jsonify({"error": "Email already exists"}), 400
    finally:
        conn.close()
```

- **login Route** : Verifies user credentials, increments login count, and redirects to the dashboard on success.

```python
@app.route('/api/login', methods=['POST'])

def login():

    data = request.json

    conn = sqlite3.connect(DB_PATH)

    cursor = conn.cursor()

    # Ensure email comparison is case-insensitive for better UX

    cursor.execute("SELECT id, name, email, password, role FROM users WHERE LOWER(email) = LOWER(?)", (data['email']

    user = cursor.fetchone()

    conn.close()
```

- **Upload media:** users able to upload their objects safe and securely

```python
@app.route('/api/media', methods=['GET'])
def get_media():
    owner_id = request.args.get('owner_id')
    is_admin = request.args.get('is_admin') == 'true'

    if is_admin:
        response = media_table.scan()
    else:
        response = media_table.scan(FilterExpression=Attr('owner_id').eq(owner_id))

    items = response.get('Items', [])
    # Sort by upload_date descending (manual sort since Scan is unordered)
    items.sort(key=lambda x: x.get('upload_date', ''), reverse=True)

    # Format analysis results back to JSON for the frontend
    for item in items:
        if 'analysis_results' in item and isinstance(item['analysis_results'], str):
            item['analysis'] = json.loads(item['analysis_results'])

    return jsonify(items)

@app.route('/api/media', methods=['POST'])
def create_media():
    data = request.json
    media_item = {
        'id': data['id'],
        'owner_id': data.get('owner_id'),
        'name': data['name'],
        'type': data['type'],
        'size': data['size'],
        'status': data['status'],
        'url': data['url'],
        'thumbnail_url': data.get('thumbnailUrl'),
        'profile': data.get('profile'),
        'upload_date': str(uuid.uuid1())
        # Using a timestamp-based ID or current time string
    }
```

```python
@app.route('/api/media', methods=['POST'])
def create_media():
    data = request.json
    media_item = {
        'id': data['id'],
        'owner_id': data.get('owner_id'),
        'name': data['name'],
        'type': data['type'],
        'size': data['size'],
        'status': data['status'],
        'url': data['url'],
        'thumbnail_url': data.get('thumbnailUrl'),
        'profile': data.get('profile'),
        'upload_date': str(uuid.uuid1())
        # Using a timestamp-based ID or current time string
    }

    media_table.put_item(Item=media_item)
    return jsonify({"status": "created"}), 201

@app.route('/api/media/<item_id>/analysis', methods=['PUT'])
def update_analysis(item_id):
    data = request.json
    media_table.update_item(
        Key={'id': item_id},
        UpdateExpression="SET analysis_results = :val, #s = :stat",
        ExpressionAttributeValues={
            ':val': json.dumps(data),
            ':stat': 'completed'
        },
        ExpressionAttributeNames={
            "#s": "status"
            # 'status' is a reserved keyword in DynamoDB
        }
    )
    return jsonify({"status": "updated"})

@app.route('/api/media/<item_id>', methods=['DELETE'])
def delete_media(item_id):
    media_table.delete_item(Key={'id': item_id})
    return jsonify({"status": "deleted"})

@app.route('/api/users', methods=['GET'])
def get_users():
    response = users_table.scan()
    items = response.get('Items', [])
    # Filter out passwords before sending to frontend
    users = [{"id": u['id'], "name": u['name'], "email": u['email'], "role": u['role']} for u in items]
```

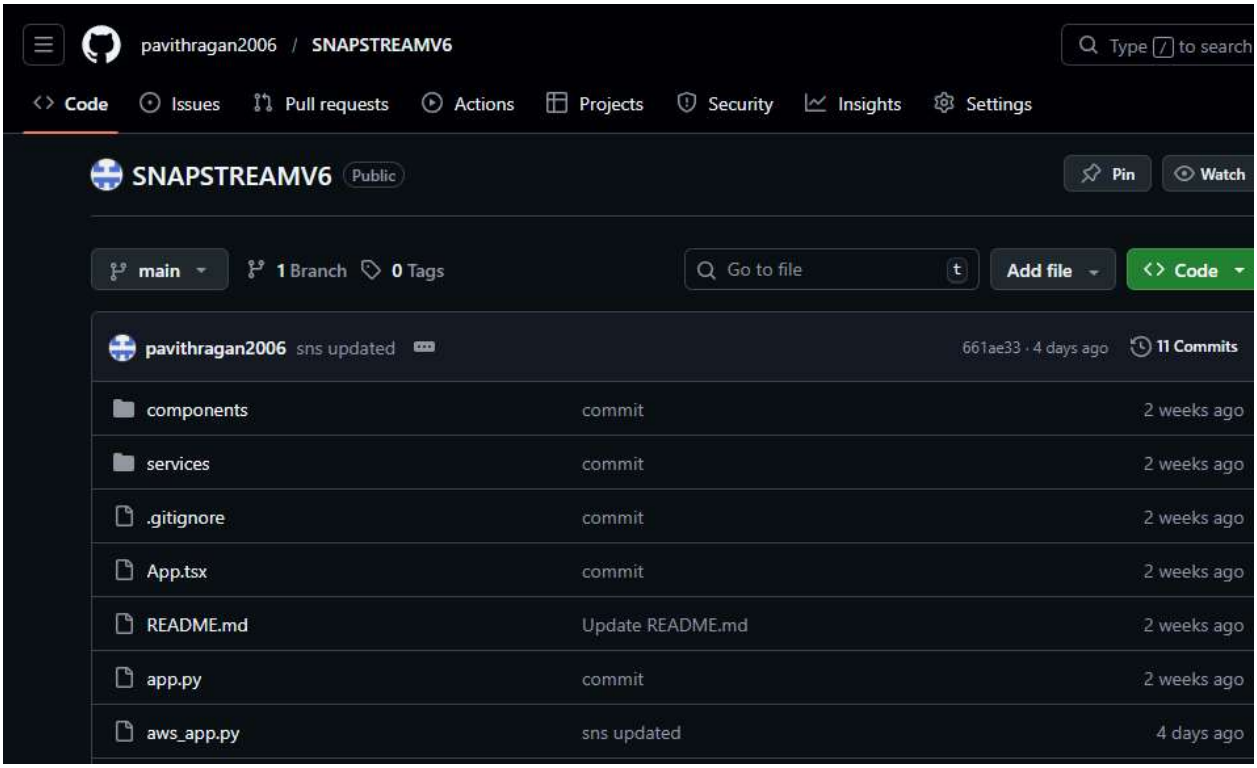
- **Application Entry point:**

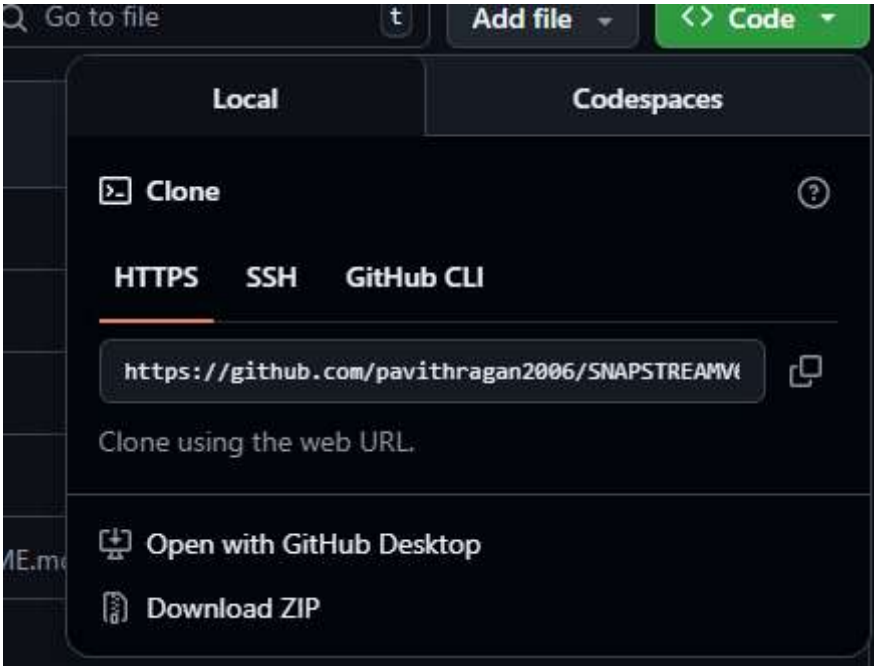
```python
if __name__ == '__main__':

 app.run(port=5000, debug=True)
```

Milestone 5: EC2 Instance Setup

- **Note: Load your Flask app and Html files into GitHub repository.**

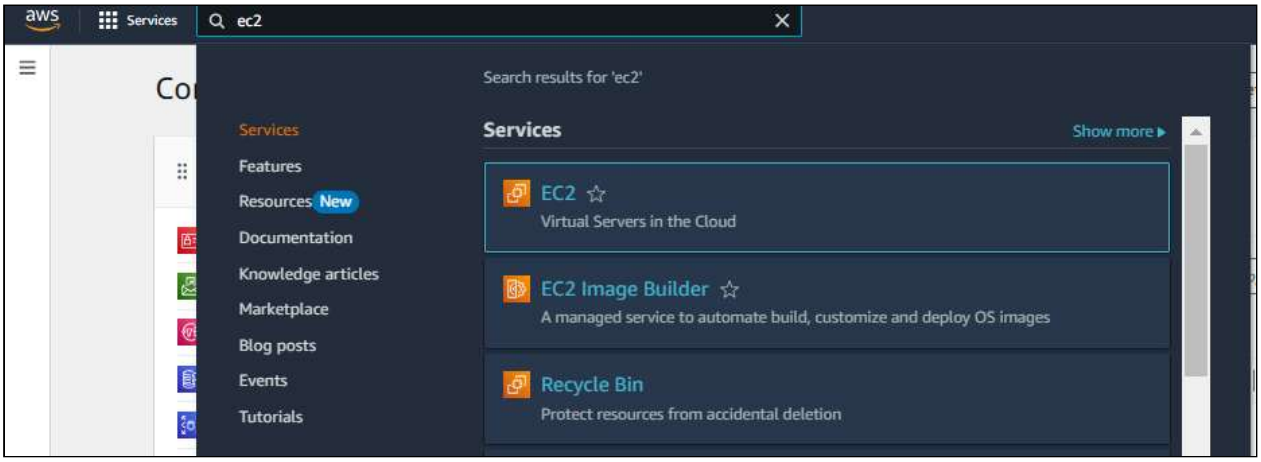




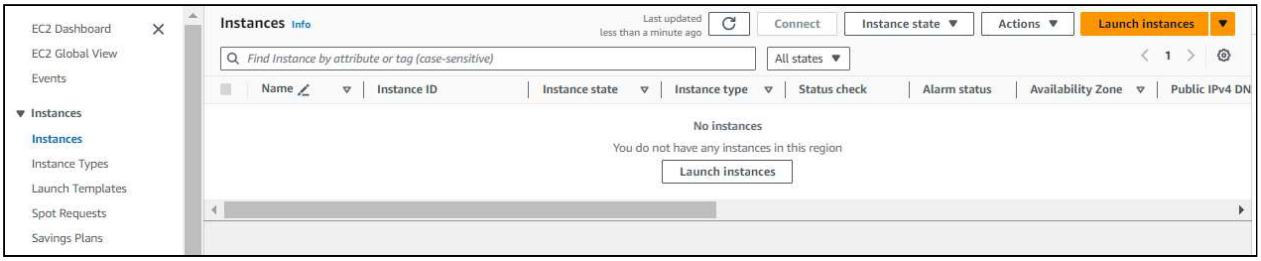
Activity 5.1: Launch an EC2 instance to host the Flask application.

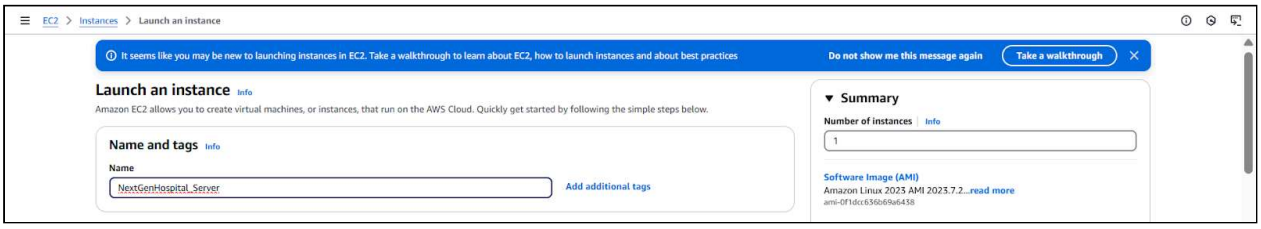
- **Launch EC2 Instance**

  In the AWS Console, navigate to EC2 and launch a new instance.

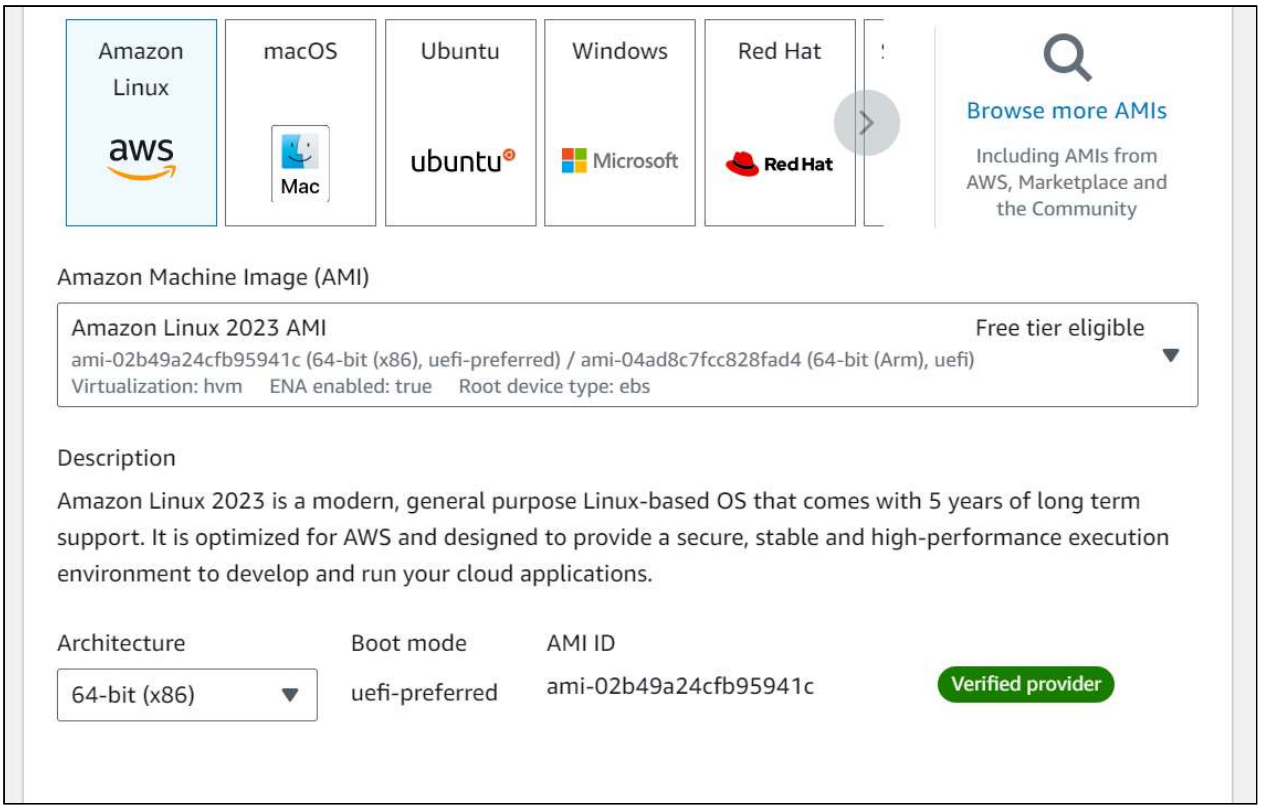


- **Click on Launch instance to launch EC2 instance**

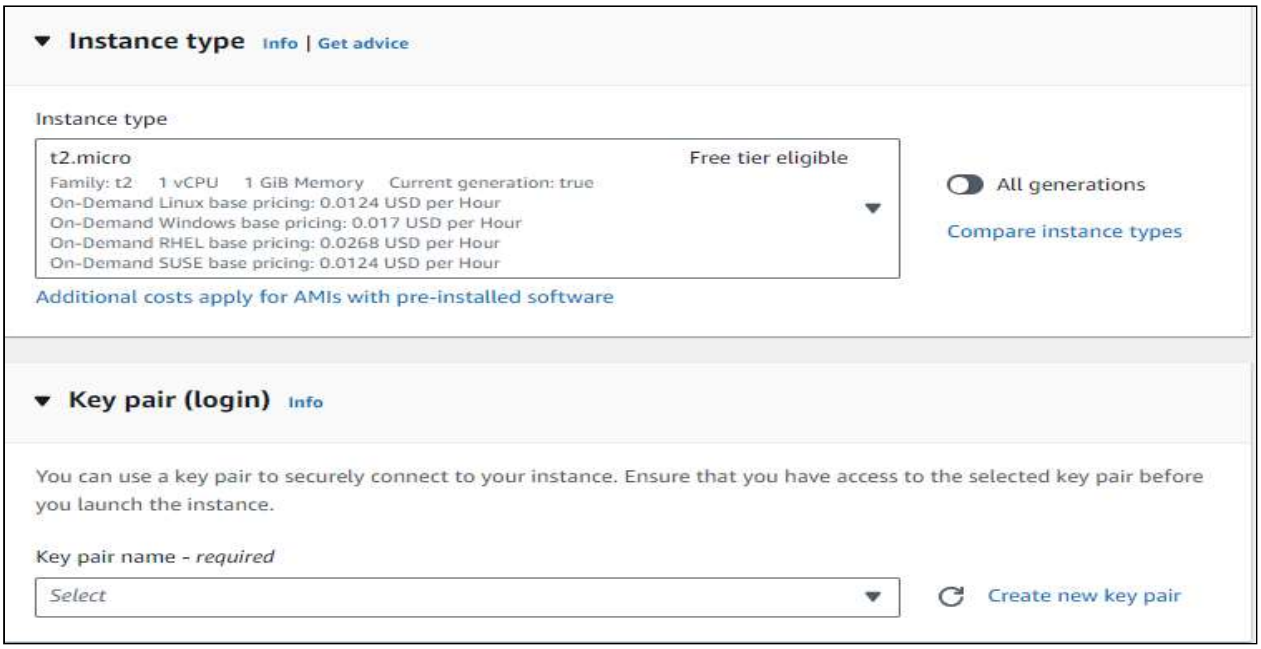




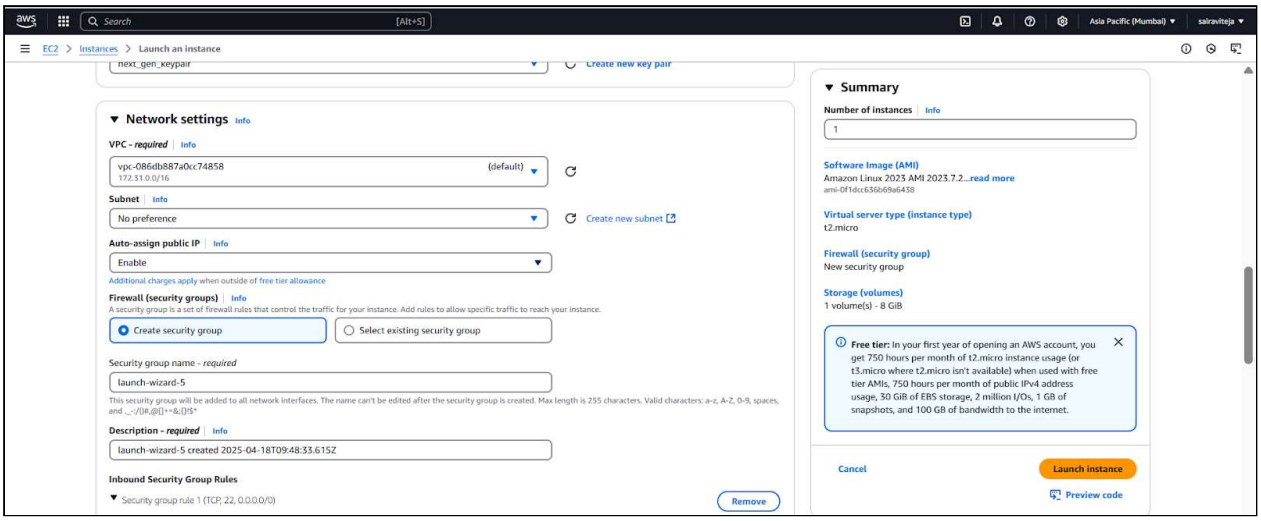
● Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible)

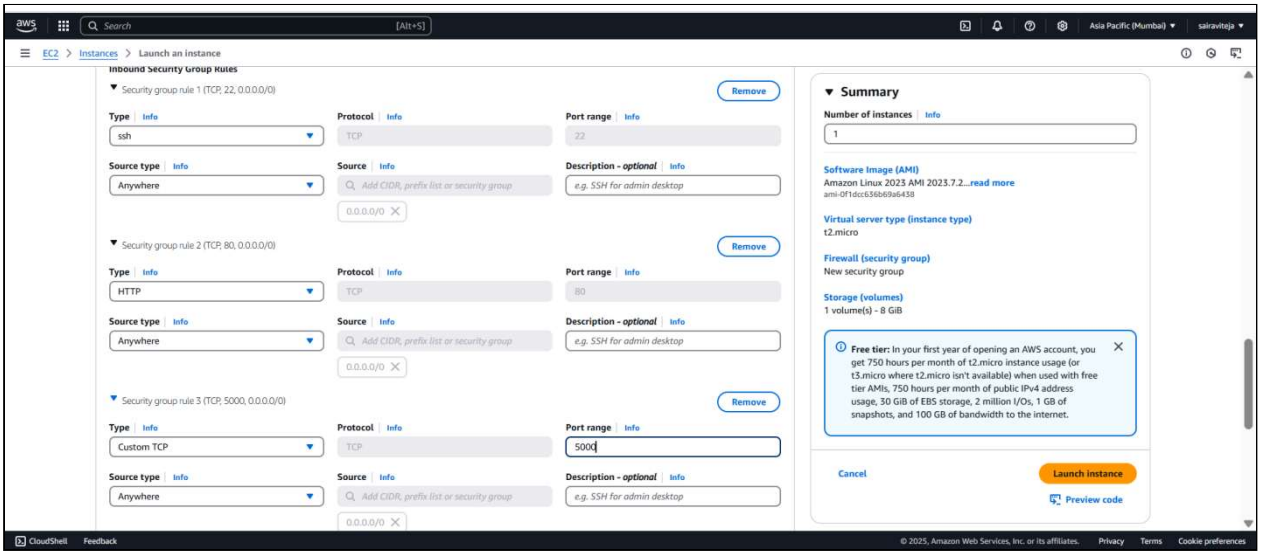


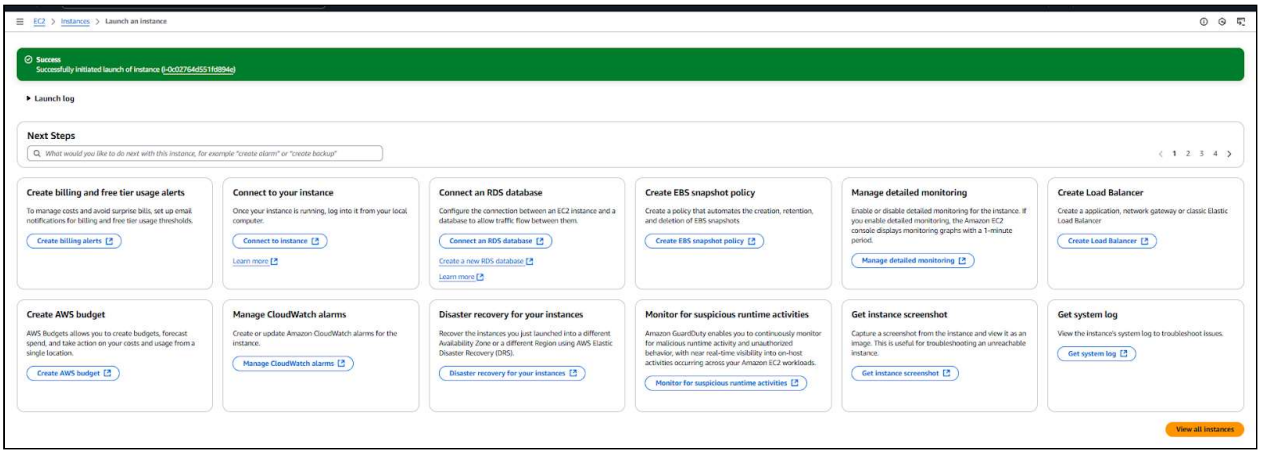
- **Create and download the key pair for Server access.**



Activity 5.2: Configure security groups for HTTP, and SSH access.



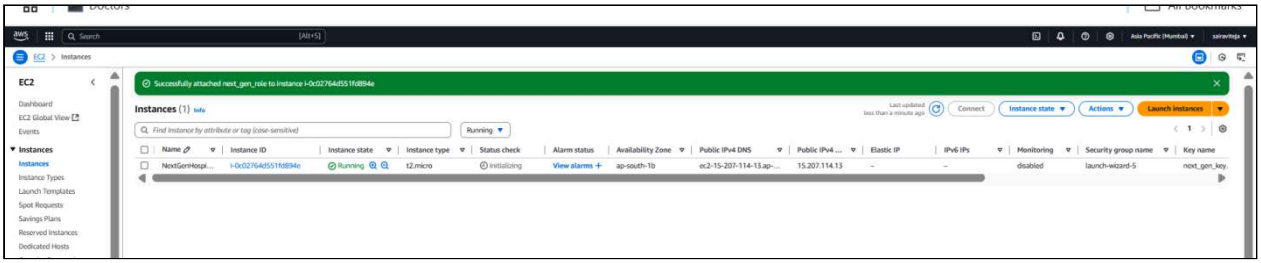


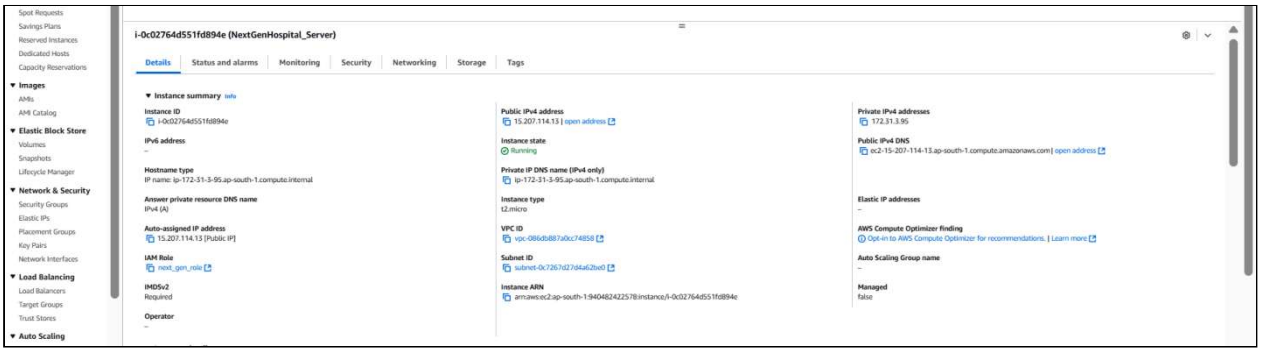


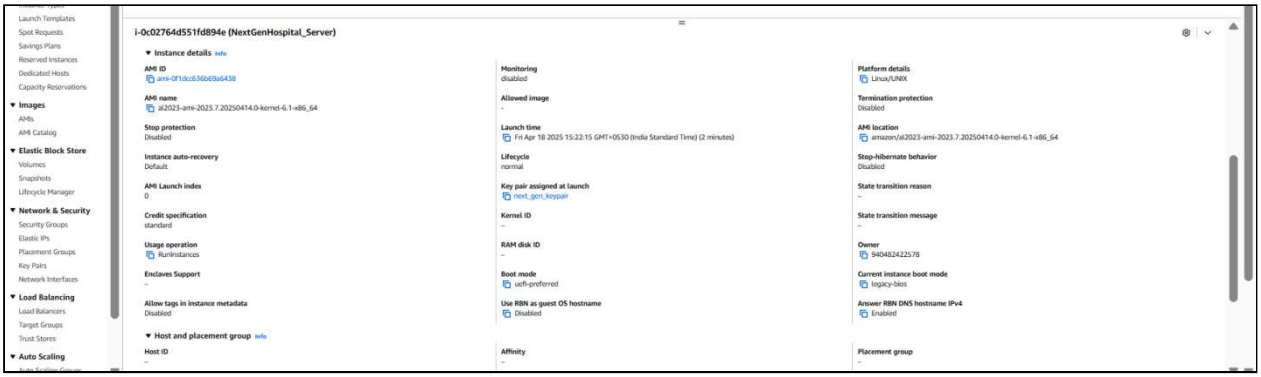
- To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached

to your EC2 instance. You can do this by selecting your instance, clicking onActions, then navigating to Security and

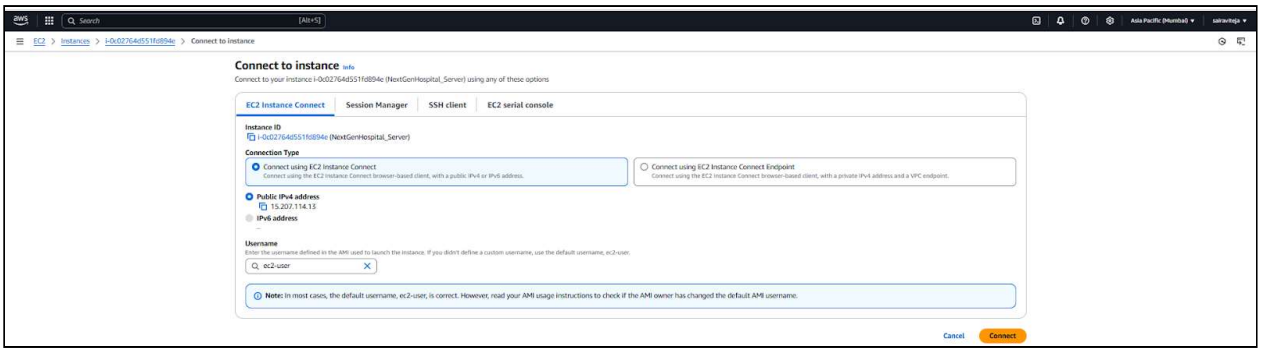
● navigate to the EC2 section in the AWS Management Console. Select the EC2 instance youwish to connect to. At the top of the EC2 Dashboard, click the Connect button. From theconnection methods presented, choose EC2 Instance Connect. Finally, click Connect again,

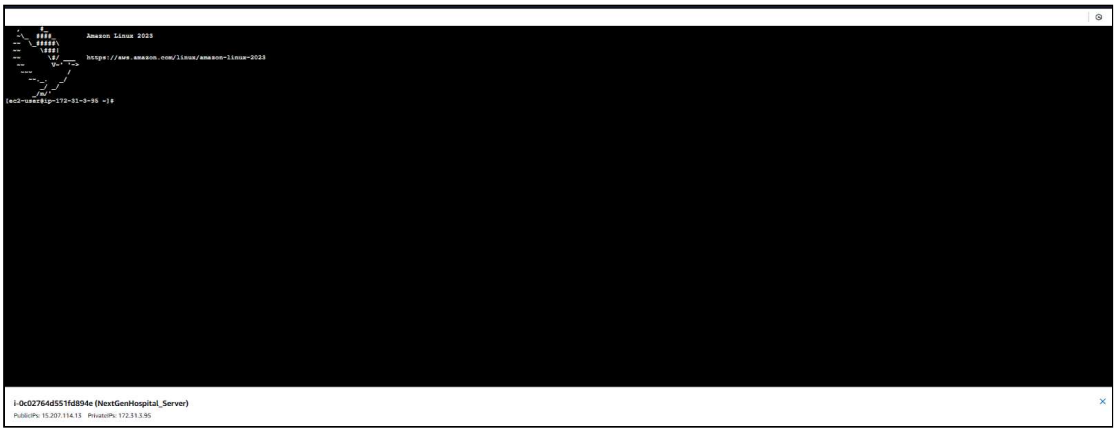






- **Now connect the EC2 with the files**





Milestone 6: Deployment on EC2

Activity 6.1: Install Software on the EC2 Instance

Install Python3, Flask, and Git:

On Amazon Linux 2:

```
sudo yum update -y
sudo yum install python3 git
sudo pip3 install flask boto3
```

Verify Installations:

```
flask --version
git --Version
```

Activity 6.2:Clone Your Flask Project from GitHub

Clone your project repository from GitHub into the EC2 instance using Git.

Run: ‘git clone https://github.com/pavithragan2006/SNAPSTREAMV6.git

Note: change your-github-username and your-repository-name with your credentials

- This will download your project to the EC2 instance.

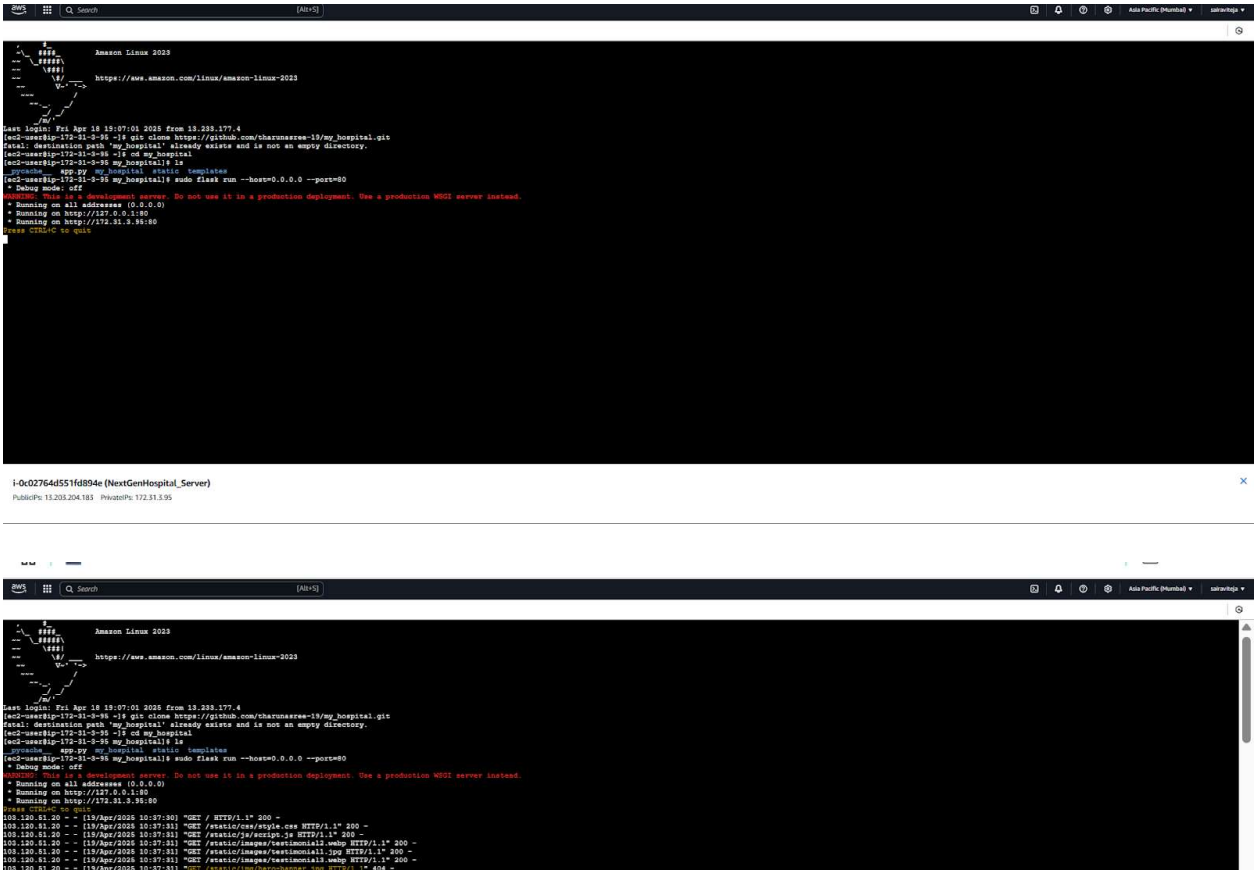
  To navigate to the project directory, run the following command:

  cd snapstream

- Once inside the project directory, configure and run the Flask application by executingthe following command with elevated privileges:

  Run the Flask Application

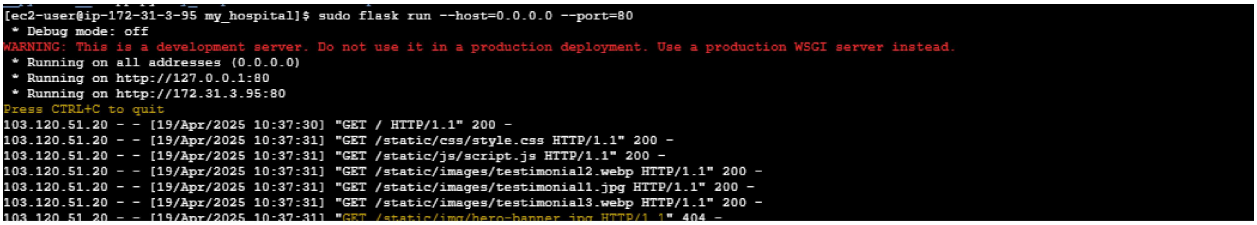
  sudo flask run --host=0.0.0.0 --port=80

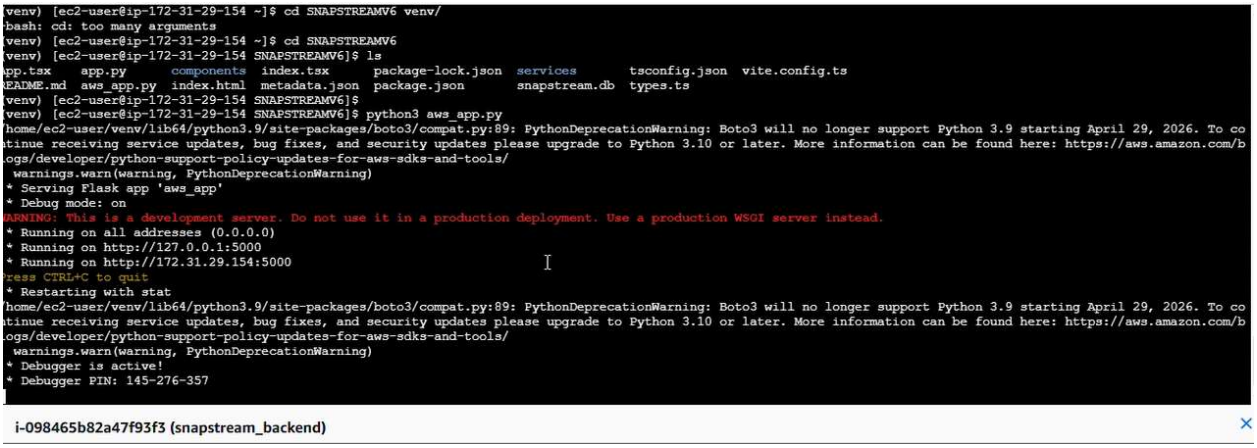


Verify the Flask app is running:

http://your-ec2-public-ip

- Run the Flask app on the EC2 instance





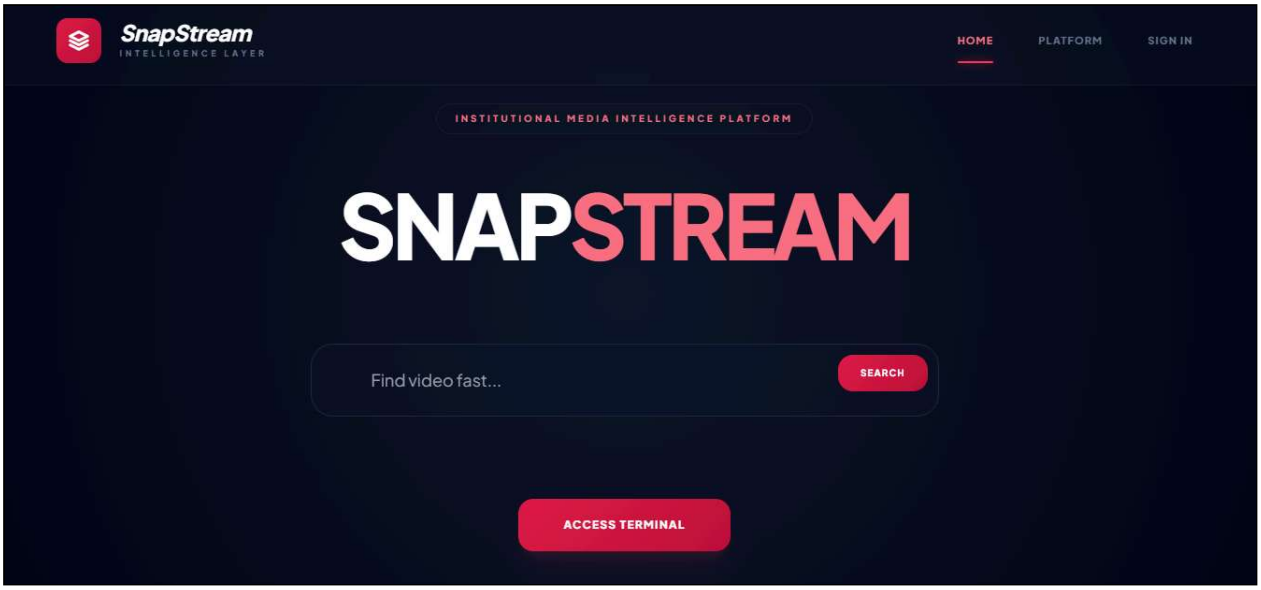
Access the website through:

PublicIPs:http://13.203.204.183/80

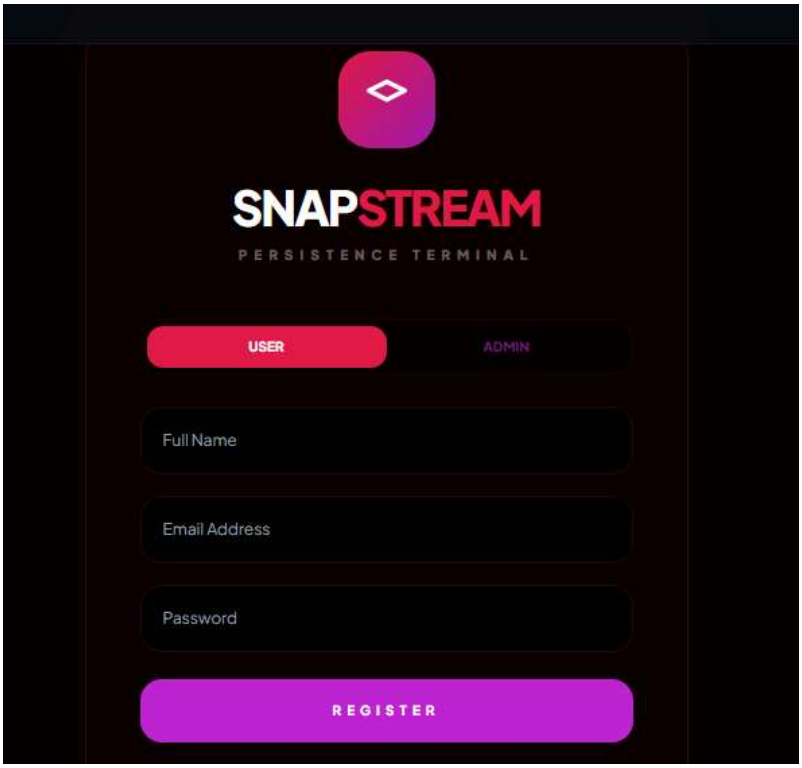
Milestone 7: Testing and Deployment

Activity 7.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

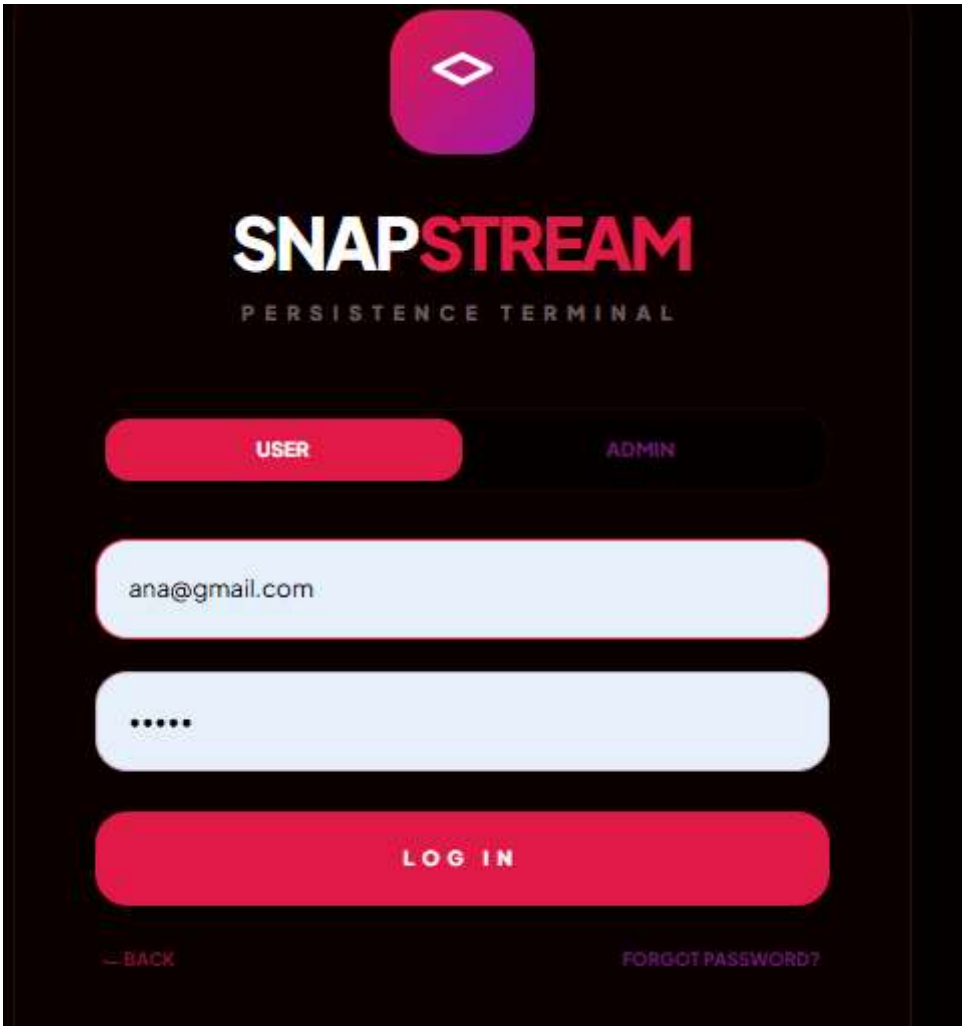
Home page



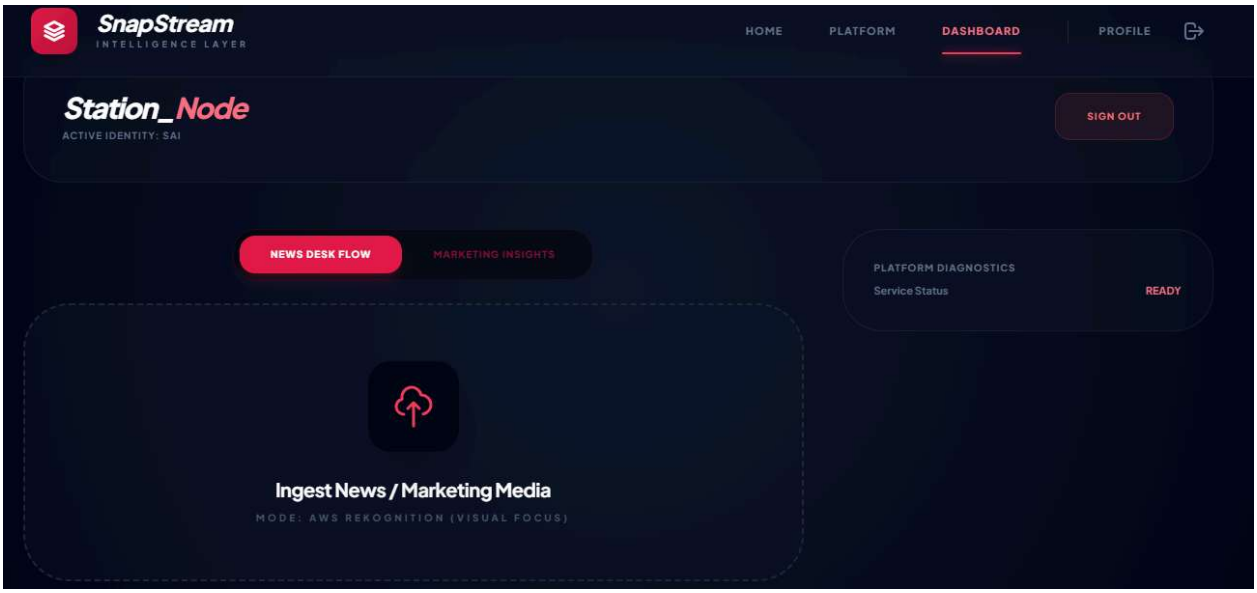
Register Page



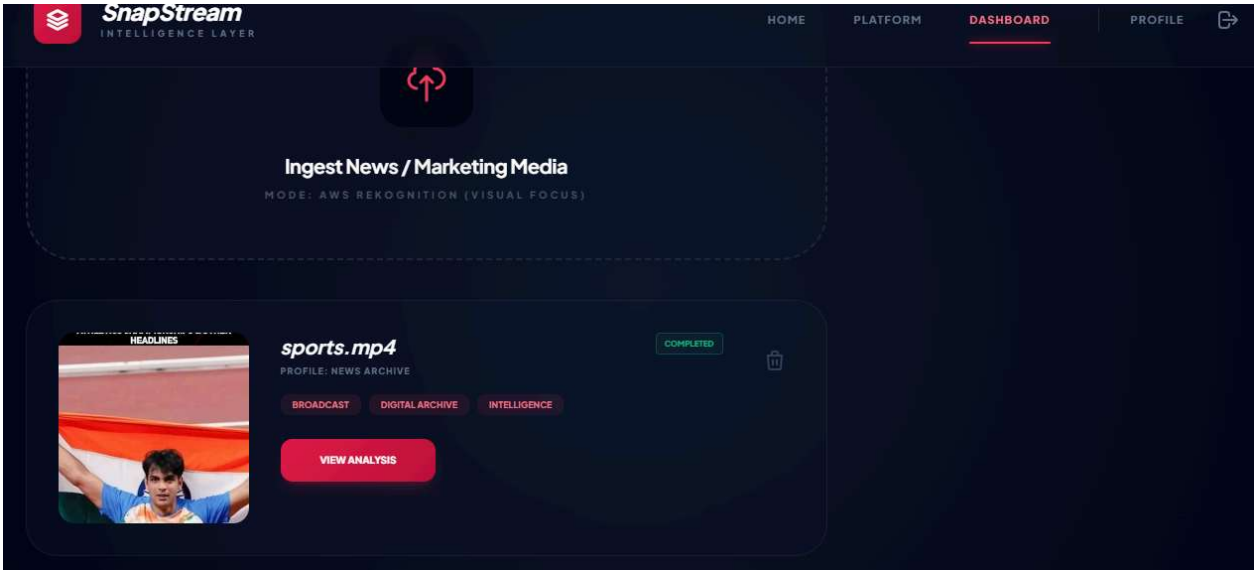
Login page



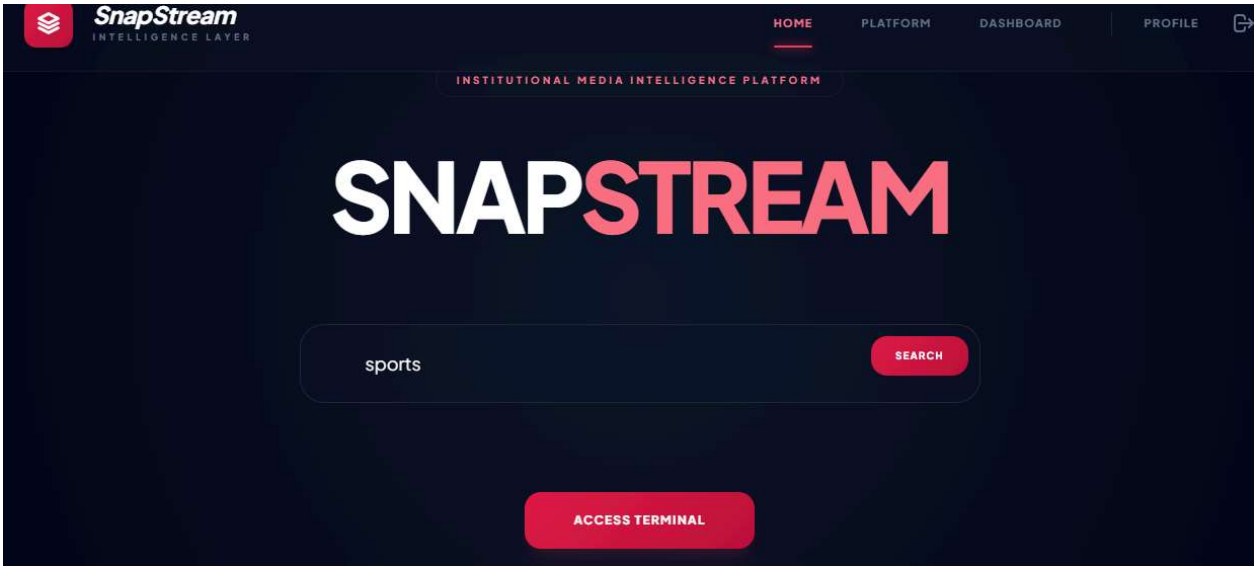
User Dashboard



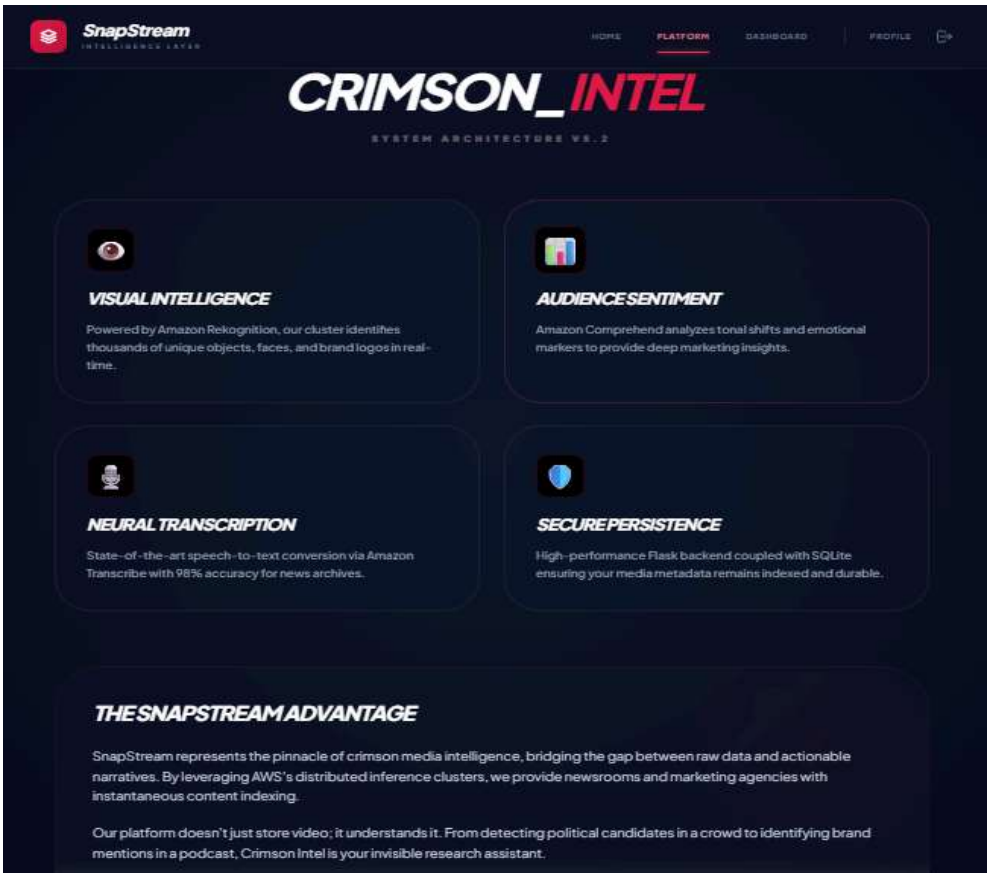
upload media



View Media



about page



CONCLUSION

The SnapStream platform has been successfully implemented as a cloud-native media management and analysis solution powered by AWS services. By integrating scalable and serverless cloud components, the system delivers a secure, efficient, and intelligent framework for handling large volumes of multimedia content. This implementation demonstrates how modern cloud technologies can simplify complex workflows associated with media storage, processing, and analysis.

Through the seamless integration of Amazon S3, AWS Lambda, and AI/ML services such as Amazon Rekognition, Transcribe, and Comprehend, SnapStream automates time-intensive tasks like tagging, transcription, and content analysis. This not only enhances operational productivity but also empowers users with data-driven insights and faster access to valuable media information.

The platform's modular design ensures scalability and cost efficiency, while AWS IAM, DynamoDB, and SNS collectively guarantee security, reliability, and real-time communication. Extensive testing has validated all critical functionalities, from secure uploads to automated analysis pipelines, confirming SnapStream's readiness for production-level deployment.

In conclusion, SnapStream represents a significant step forward in intelligent media management, showcasing the power of AWS cloud infrastructure to transform traditional media workflows into an automated, scalable, and insight-driven ecosystem.