MedTrack: AWS Cloud-Enabled Healthcare Management System

Project Description

In today's fast-evolving healthcare landscape, efficient communication and coordination between doctors and patients are crucial. MedTrack is a cloud-based healthcare management system that streamlines patient-doctor interactions by providing a centralized platform for booking appointments, managing medical histories, and submitting diagnoses. The system leverages Flask for backend development, AWS EC2 for hosting, and DynamoDB for data management. It supports real-time notifications through AWS SNS and secure access control via AWS IAM, ensuring both accessibility and data integrity. MedTrack is designed to improve healthcare accessibility, efficiency, and real-time communication.

Scenarios

Scenario 1: Efficient Appointment Booking System for Patients

AWS EC2 supports multiple concurrent users, allowing patients to log in and book appointments. Flask handles backend operations and integrates with DynamoDB to store real-time data efficiently.

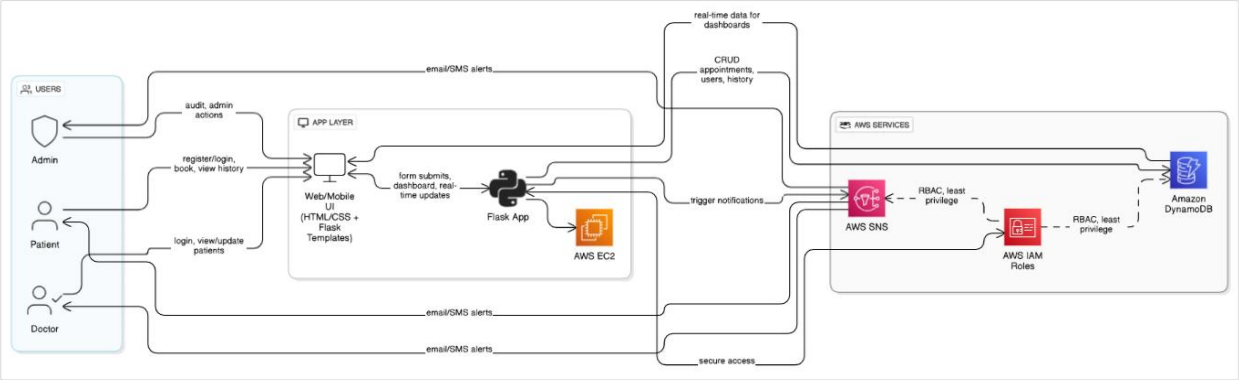
Scenario 2: Secure User Management with IAM

MedTrack uses IAM roles for secure, role-based access control. Patients and doctors receive permissions specific to their roles, ensuring secure access to sensitive data.

Scenario 3: Easy Access to Medical History and Resources

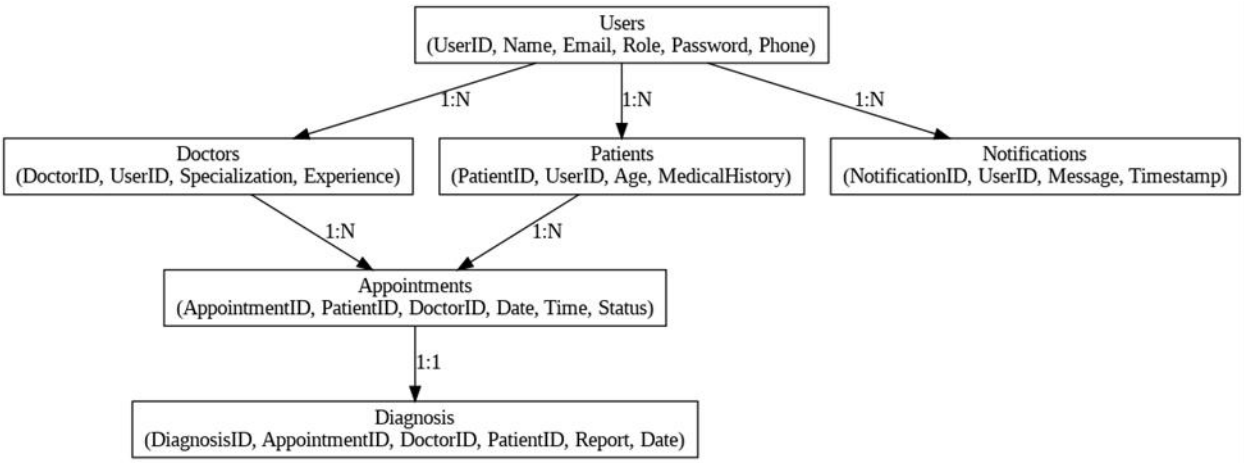
Doctors can retrieve patient records and update diagnoses in real time. Flask and DynamoDB integration enables high-speed access and data updates, ensuring seamless operation during peak usage.

AWS Architecture



- **Flask (Python Framework):** Handles routing and backend logic.
- **Amazon EC2:** Hosts the Flask application.
- **Amazon DynamoDB:** Stores patient data, appointments, and records.
- **AWS SNS:** Sends real-time alerts and appointment confirmations.
- **AWS IAM:** Controls user access and permissions securely.

Entity Relationship (ER) Diagram



The ER diagram illustrates entities such as Users (patients/doctors), Appointments, and their relationships. Key attributes include user ID, name, email, appointment ID, doctor ID, date, and status. This structure supports efficient query processing and normalization.

Pre-requisites

- AWS Account Setup:
  https://docs.aws.amazon.com/accounts/latest/reference/getting-started.html
- IAM Overview:
  https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
- EC2 Tutorial:
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- DynamoDB Introduction:
  https://docs.aws.amazon.com/amazondynamodb/Introduction.html
- SNS Documentation:
  https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- Git Documentation:
  https://git-scm.com/doc
- VS Code:
  https://code.visualstudio.com/download

Project Workflow

Milestone 1: Web Application Development and Setup

- Set up the Flask app with routing and templates.
- Use local Python lists/dictionaries for initial testing.
- Integrate AWS services (DynamoDB, SNS) using boto3.

Milestone 2: AWS Account Setup

- Access AWS via Troven Labs.
- Avoid personal AWS account usage to prevent billing issues.

Milestone 3: DynamoDB Database Creation and Setup

- Create a Users table (Primary key: Email).
- Create an Appointments table (Primary key: appointment_id).

Milestone 4: SNS Notification Setup

- Create an SNS topic.
- Subscribe to users/admin via email.
- Confirm subscriptions and note Topic ARN.

Milestone 5: IAM Role Setup

- Create IAM Role (e.g., flask dynamodb sns).
- Attach policies: AmazonDynamoDBFullAccess, AmazonSNSFullAccess.

Milestone 6: EC2 Instance Setup

- Launch EC2 instance (Amazon Linux 2/Ubuntu, t2.micro).
- Assign IAM Role and key pair.
- Configure security groups for HTTP/SSH.

Milestone 7: Deployment on EC2

- Install Python3, Flask, Git.
- Clone the GitHub repo.

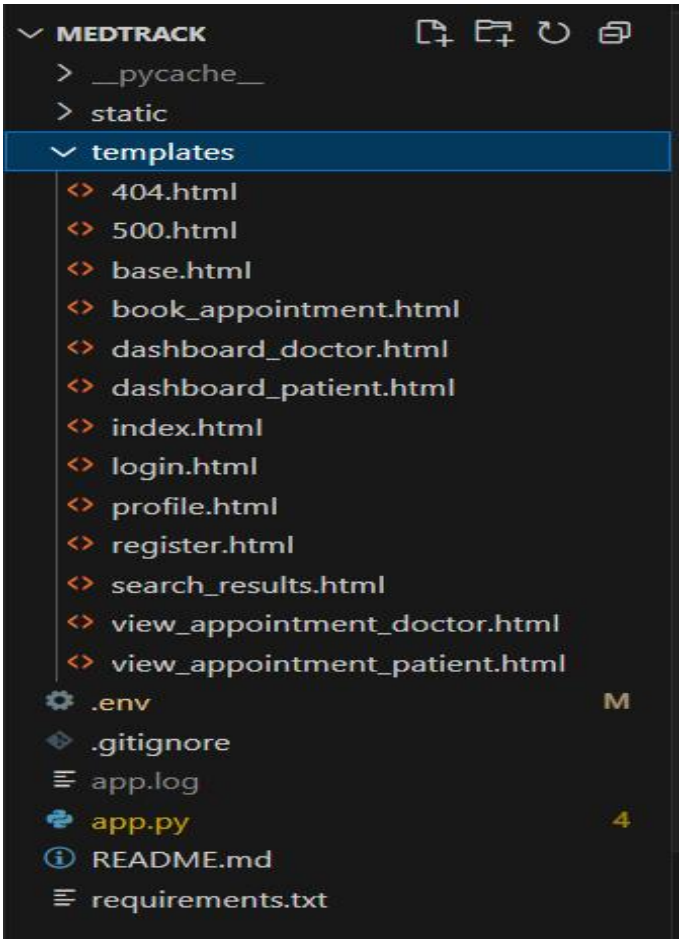
- Run Flask app: sudo flask run --host=0.0.0.0 --port=5000

Milestone 8: Testing and Deployment

- Verify registration, login, appointment booking, and SNS notifications.

Milestone 1: Web Application Development and Setup

- Set up Flask app with routing and templates.



- Use local Python lists/dictionaries for initial testing.

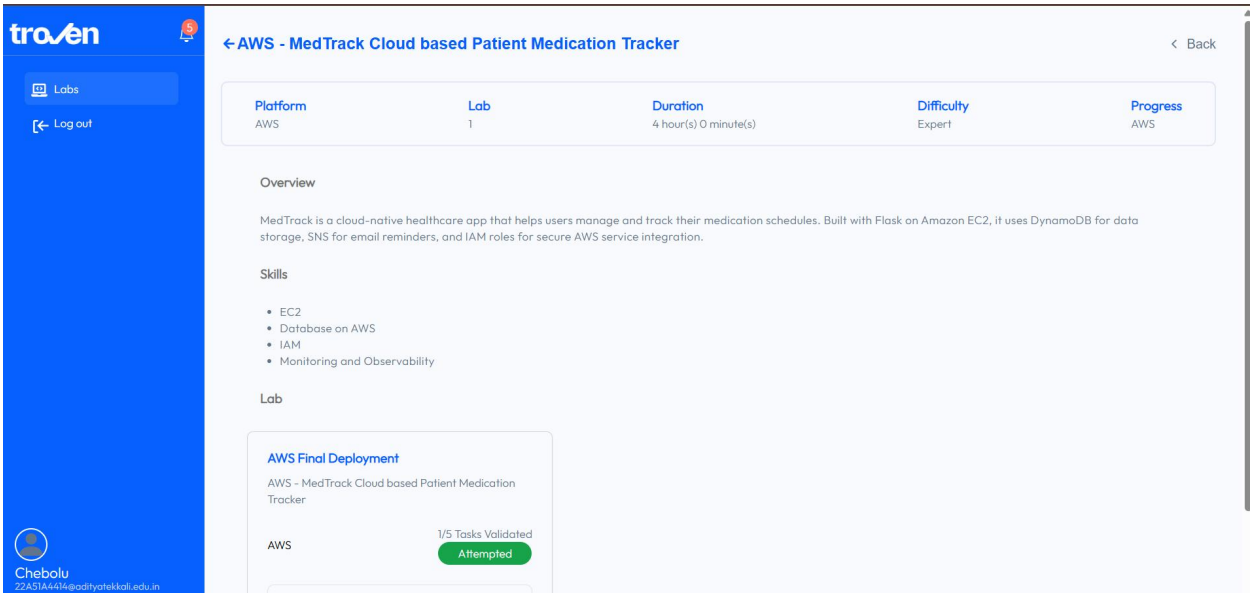
```python
# --- Mock Data Stores ---
users = []
appointments = []

# --- Register a User ---
def register_user(email, name, role, password):
    for user in users:
        if user['email'] == email:
            return "User already exists."
    users.append({
        "email": email,
        "name": name,
        "role": role,
        "password": password  # In real app, hash it!
    })
    return "User registered successfully."


# --- Login User ---
def login_user(email, password):
    for user in users:
        if user['email'] == email and user['password'] == password:
            return f"Welcome {user['name']} ({user['role']})"
    return "Invalid credentials."


# --- Book Appointment ---
def book_appointment(patient_email, doctor_email, date, time):
    appointment_id = len(appointments) + 1
    appointments.append({
        "appointment_id": appointment_id,
        "patient": patient_email,
        "doctor": doctor_email,
        "date": date,
        "time": time,
        "status": "pending"
```

- Integrate AWS services (DynamoDB, SNS) using boto3.

```python
from flask import Flask, request, session, redirect, url_for, render_template, flash
import boto3
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import logging
import os
import uuid
from dotenv import load_dotenv
```

Milestone 2: AWS Account Setup

- Access AWS via Troven Labs.

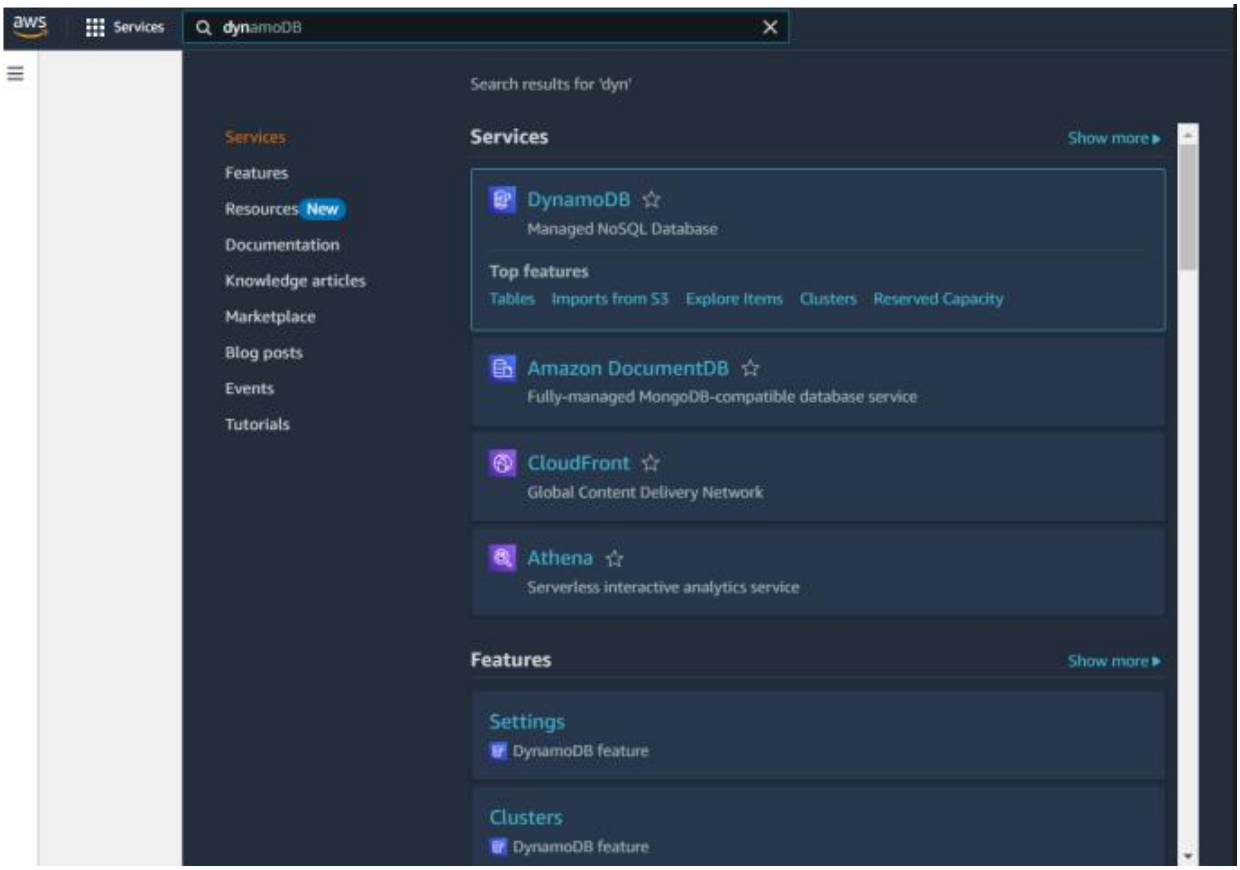


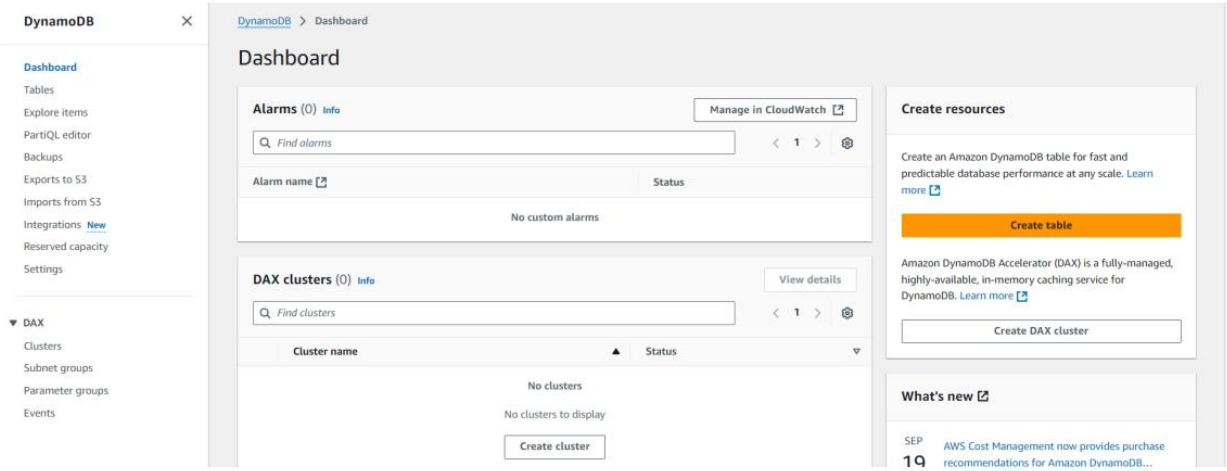
- Avoid personal AWS account usage to prevent billing issues.

Milestone 3: DynamoDB Database Creation and Setup

Activity 3.1:Navigate to the DynamoDB

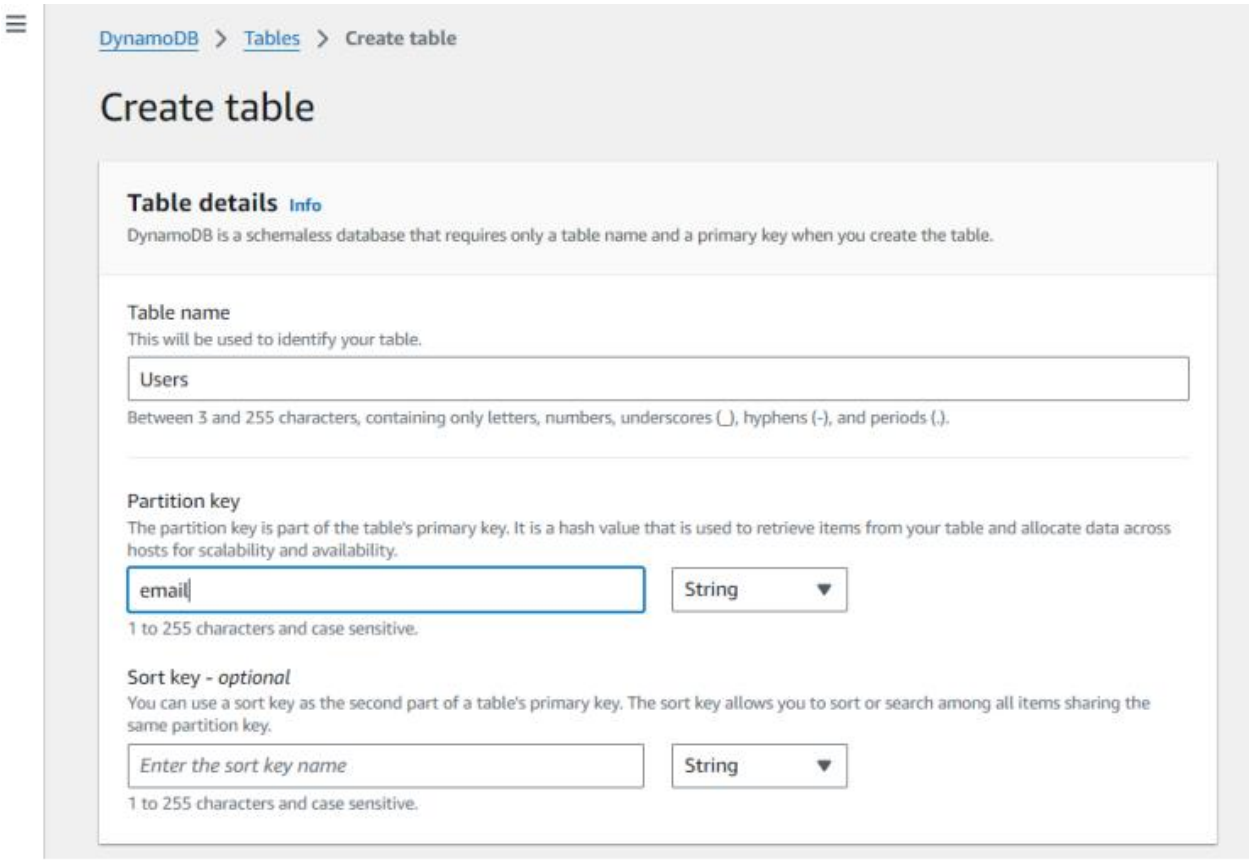
○ **In the AWS Console, navigate to DynamoDB and click on Create tables**



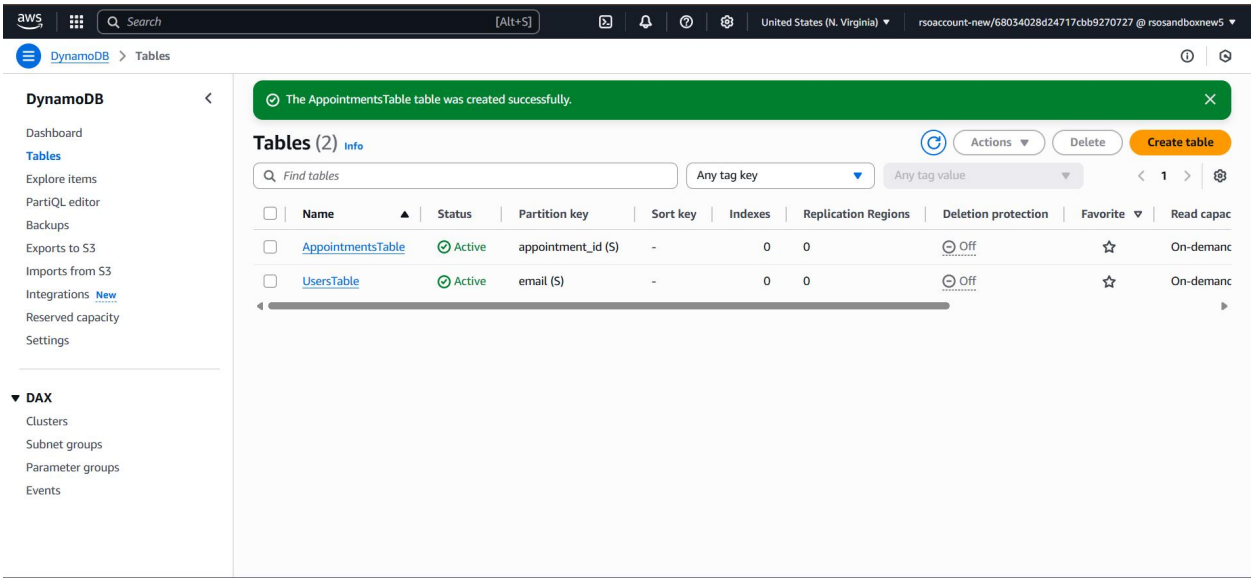


Activity 3.2:Create a DynamoDB table for the Create Users table (Primary key: Email).

○ **Create Users table with partition key “Email” with type String and click on create tables.**



Activity 3.3:Create Appointments table (Primary key: appointment_id).

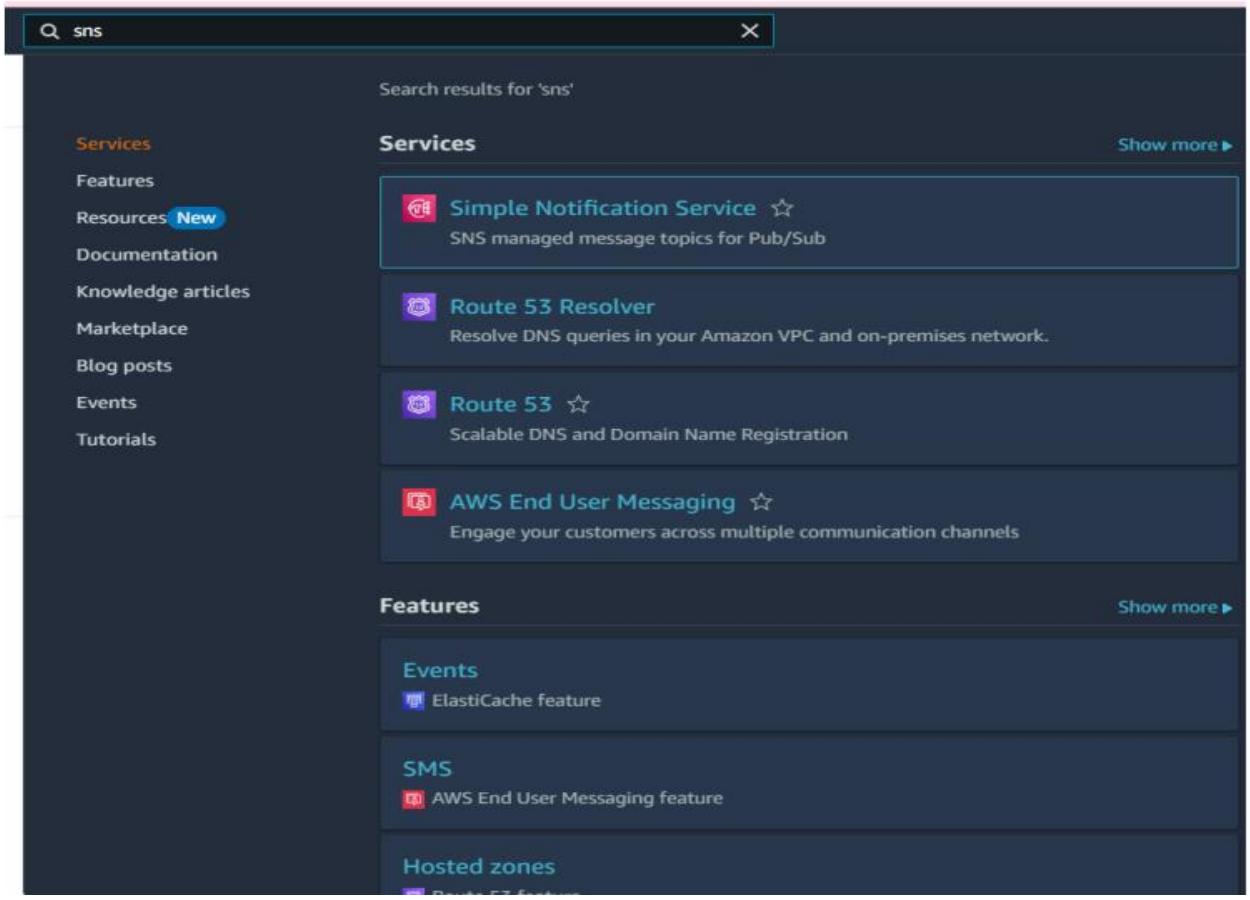


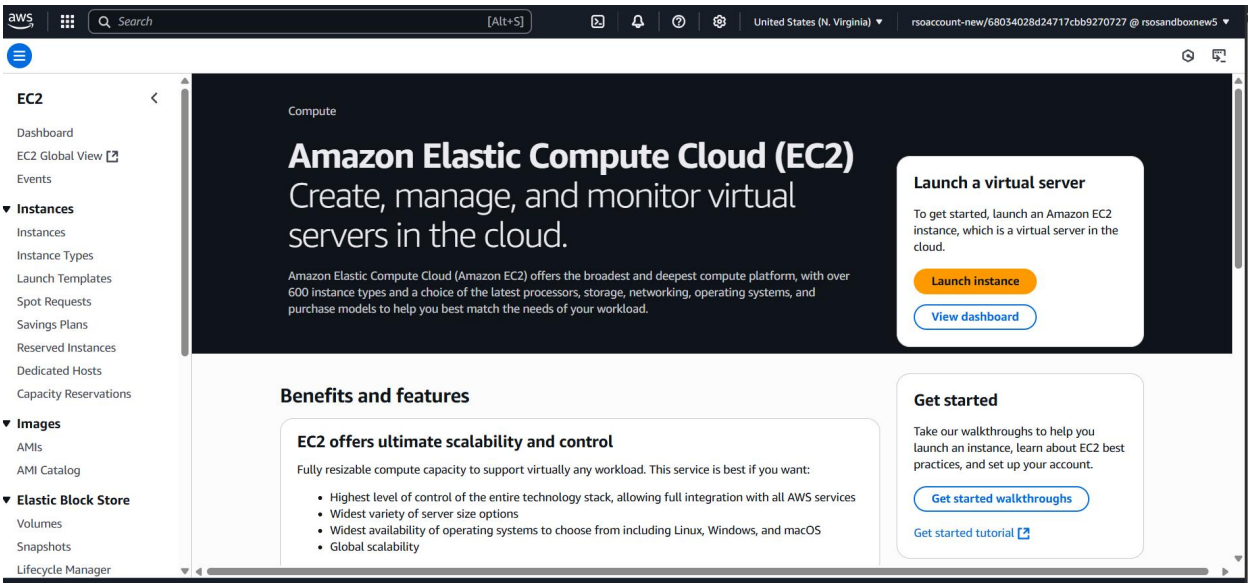
Follow the same steps to create an Appointments table with Email as the primary key for booking diagnosis data.

Milestone 4: SNS Notification Setup

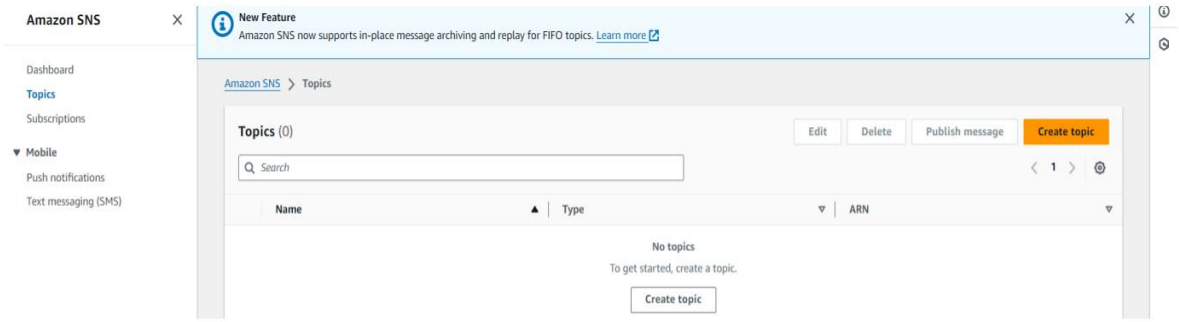
- **Activity 4.1: Create SNS topics for sending email notifications to users and doctor prescriptions.**

○ In the AWS Console, search for SNS and navigate to the SNS Dashboard.

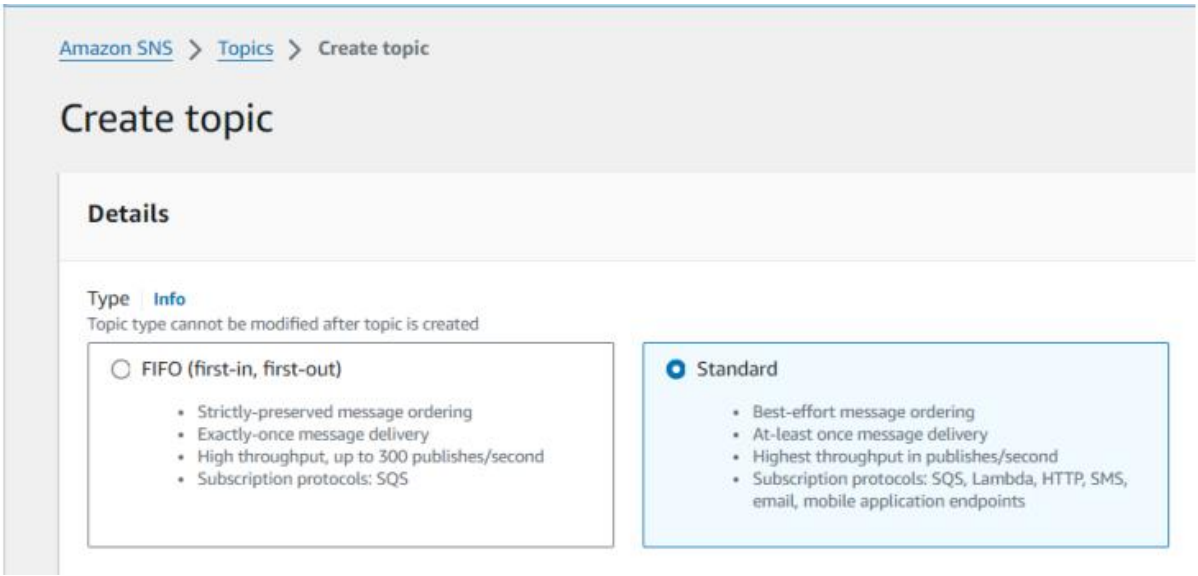




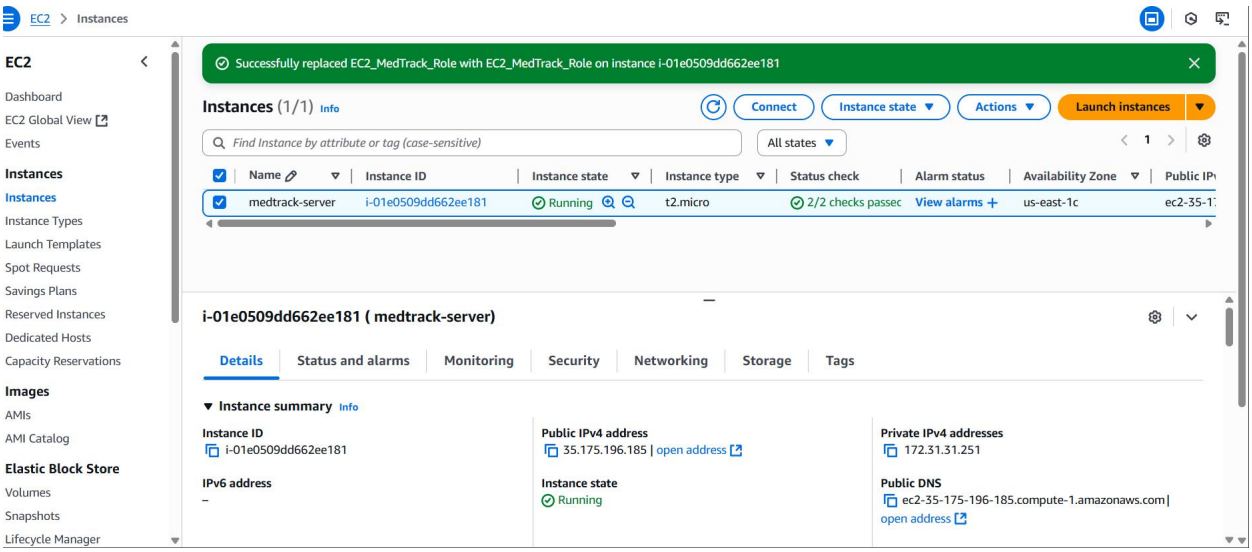
○ Click on Create Topic and choose a name for the topic.



○ Choose Standard type for general notification use cases, and click on Create Topic.

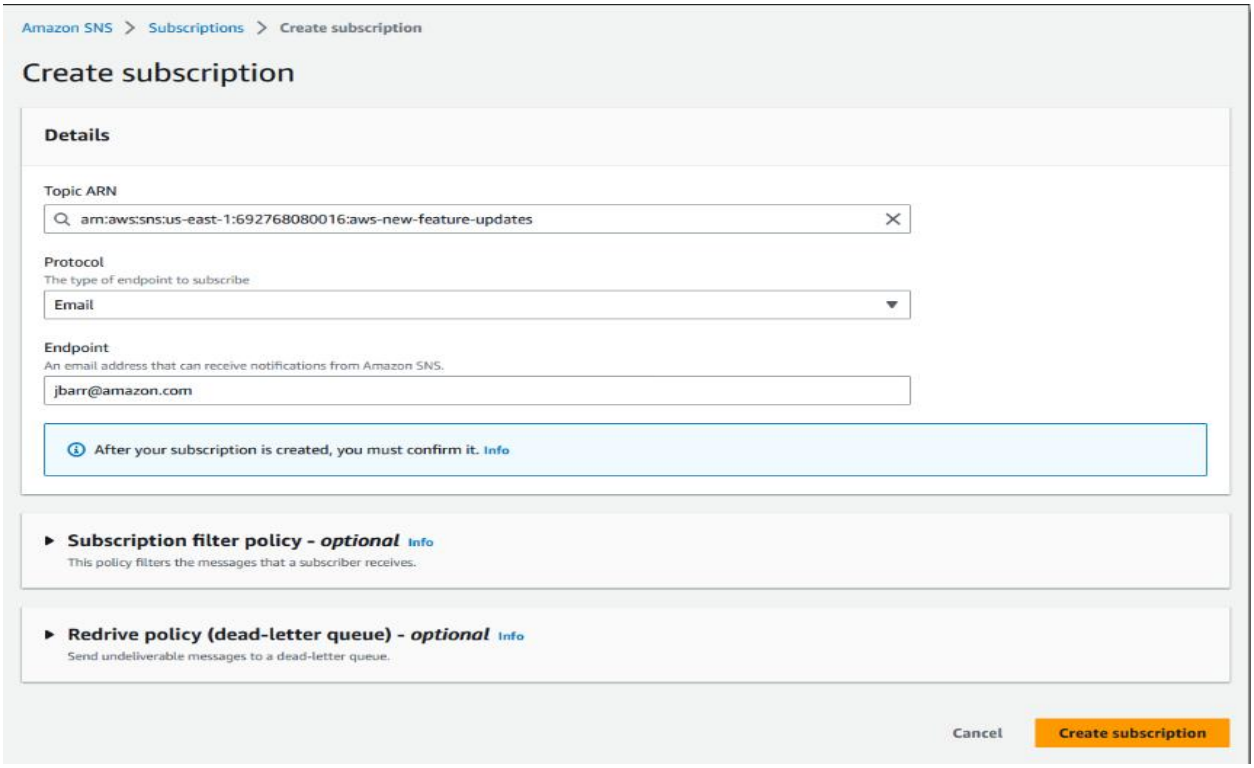


○ Configure the SNS topic and note down the Topic ARN.

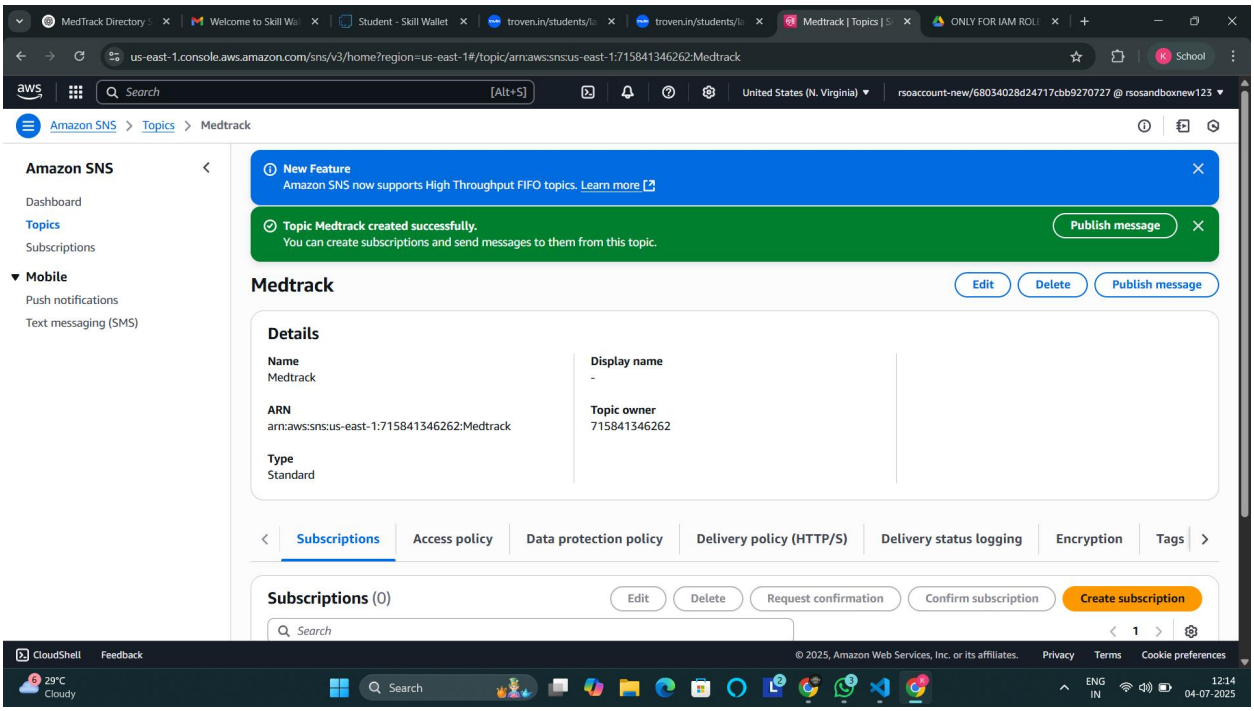


● Activity 4.2: Subscribe users and Doctors to relevant SNS topics to receive real-time notifications when a book appointment is made.

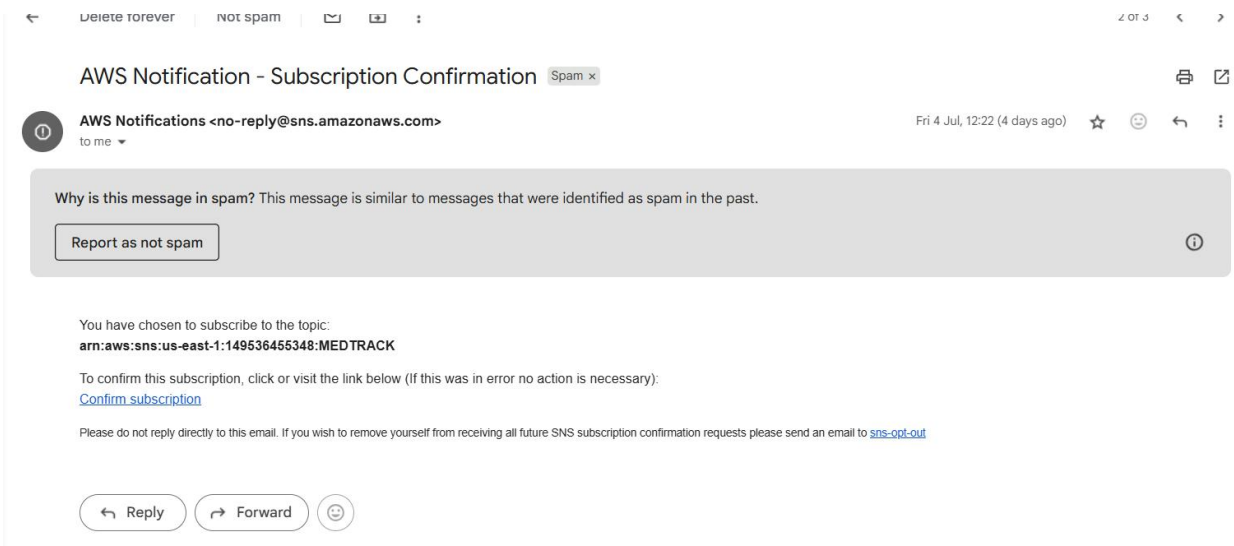
○ Subscribe users (or admin staff) to this topic via Email. When an appointment is made, notifications will be sent to the subscribed emails.

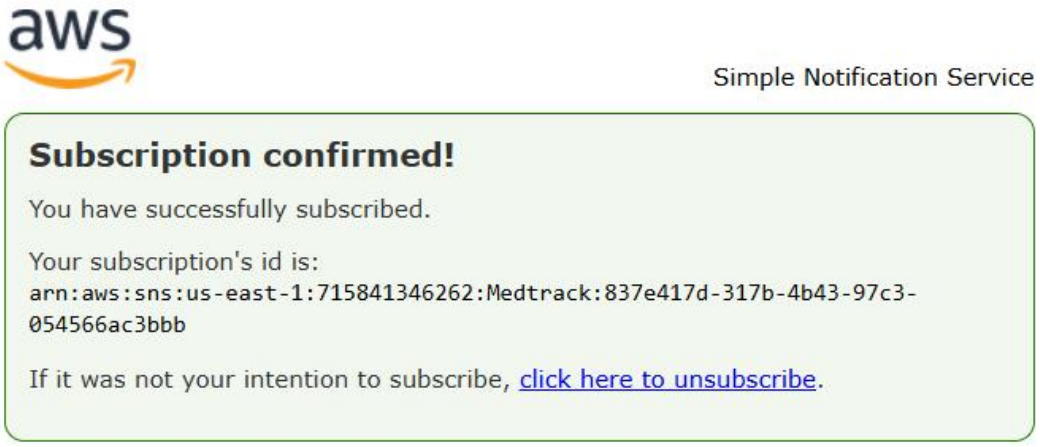


After the subscription request for the mail confirmation.

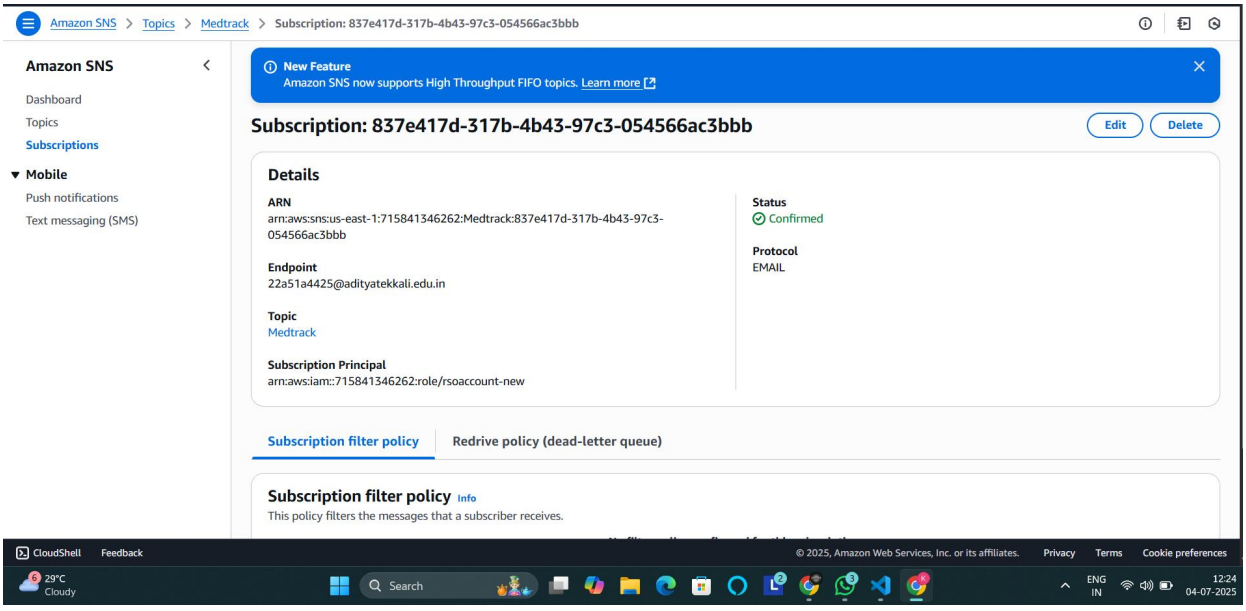


Navigate to the subscribed Email account and click on the confirm subscription in the AWS Notification- Subscription Confirmation email.



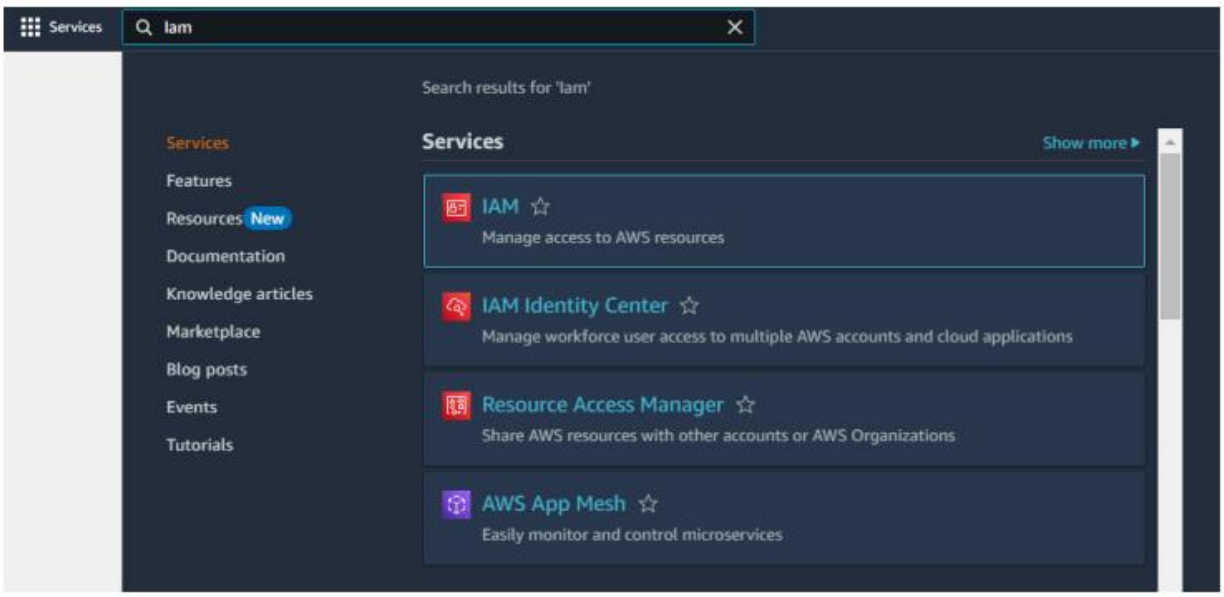


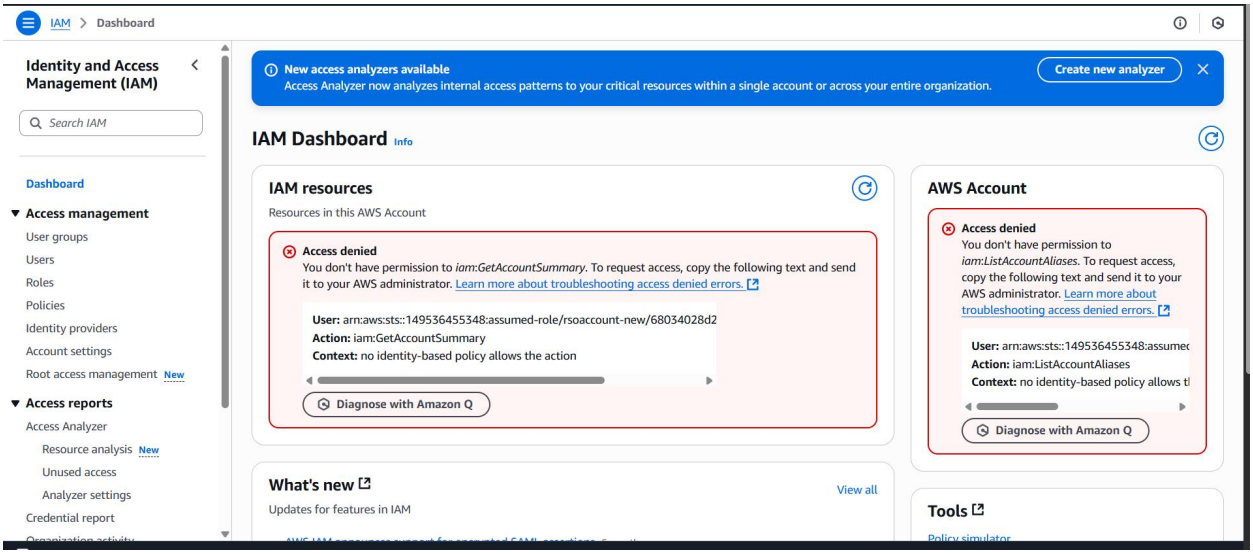
- Successfully done with the SNS mail subscription and setup, now store the ARN link.



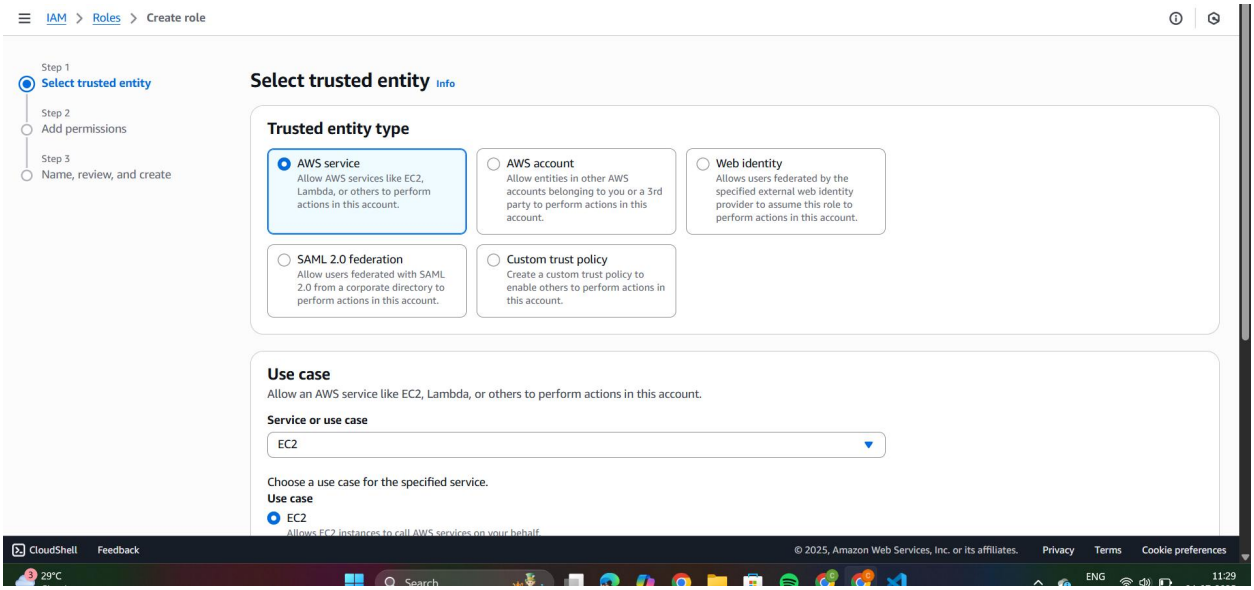
Milestone 5: IAM Role Setup

- **Activity 5.1:Create IAM Role. ○ In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.**

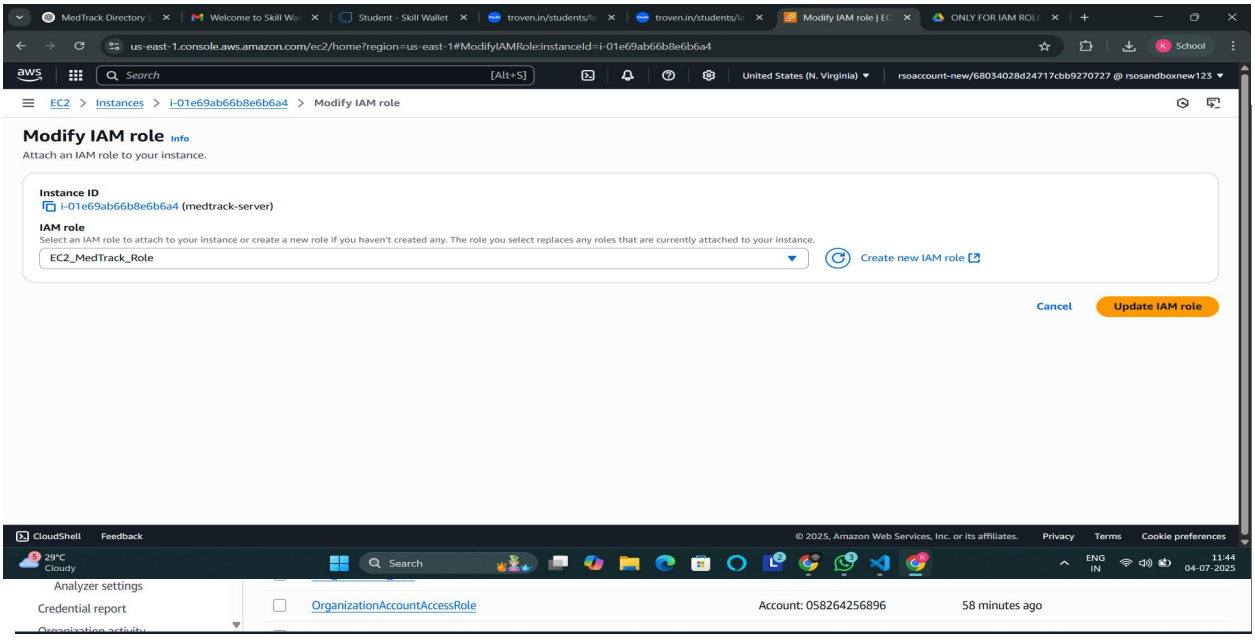




- **Create IAM Roles:**

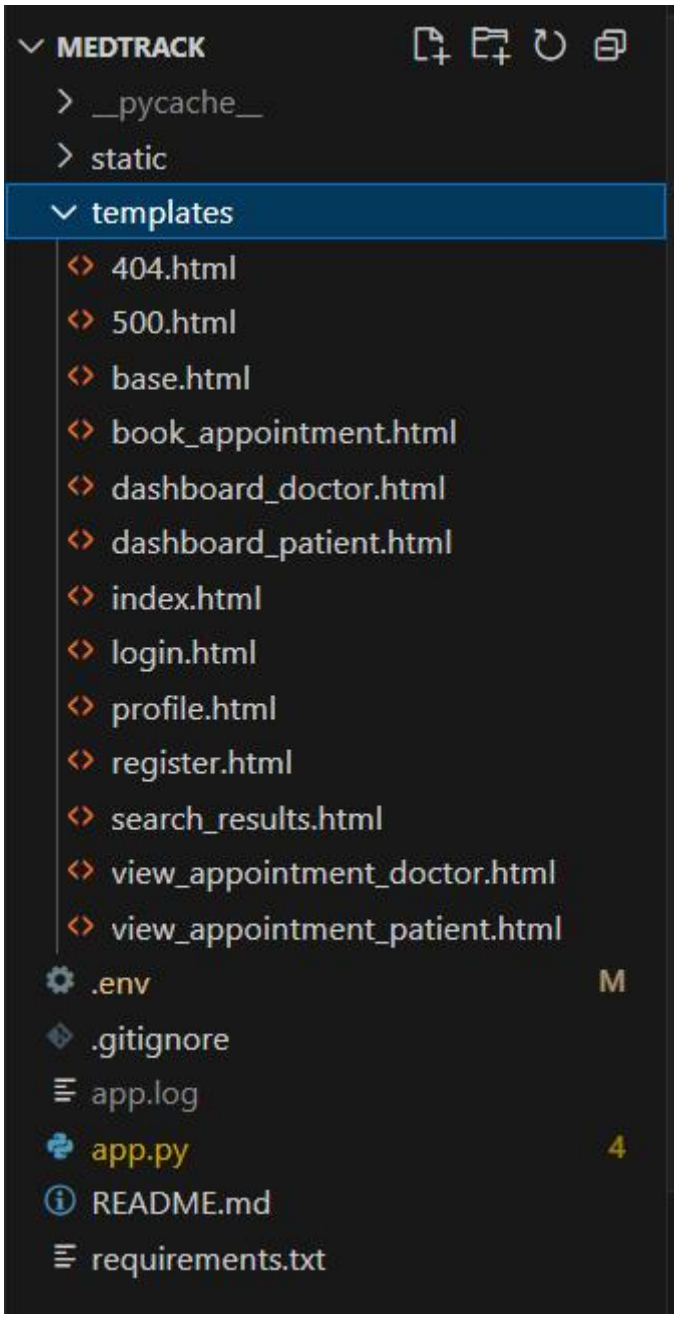


- **Attach policies: AmazonDynamoDBFullAccess, AmazonSNSFullAccess.**



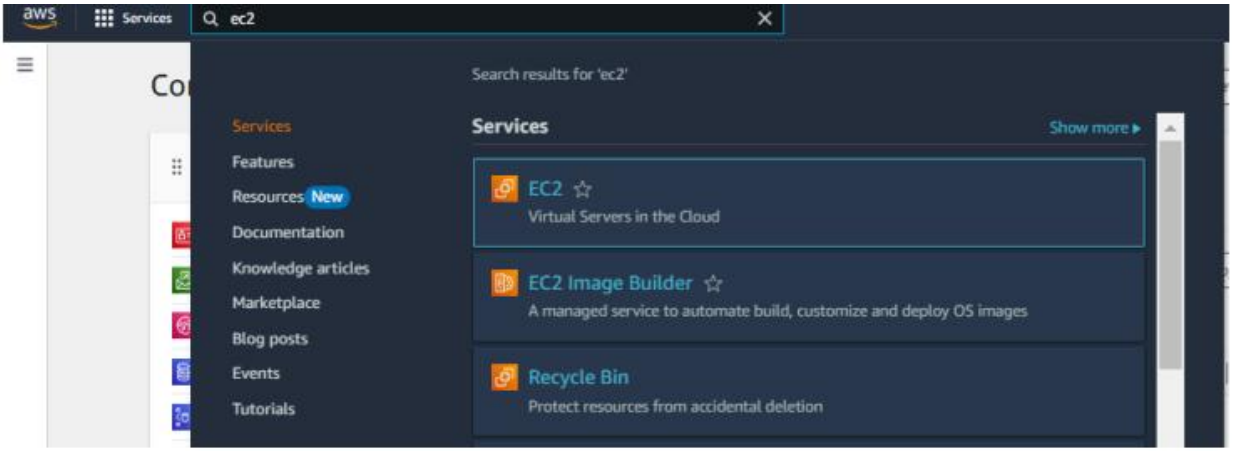
Milestone 6: EC2 Instance Setup

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

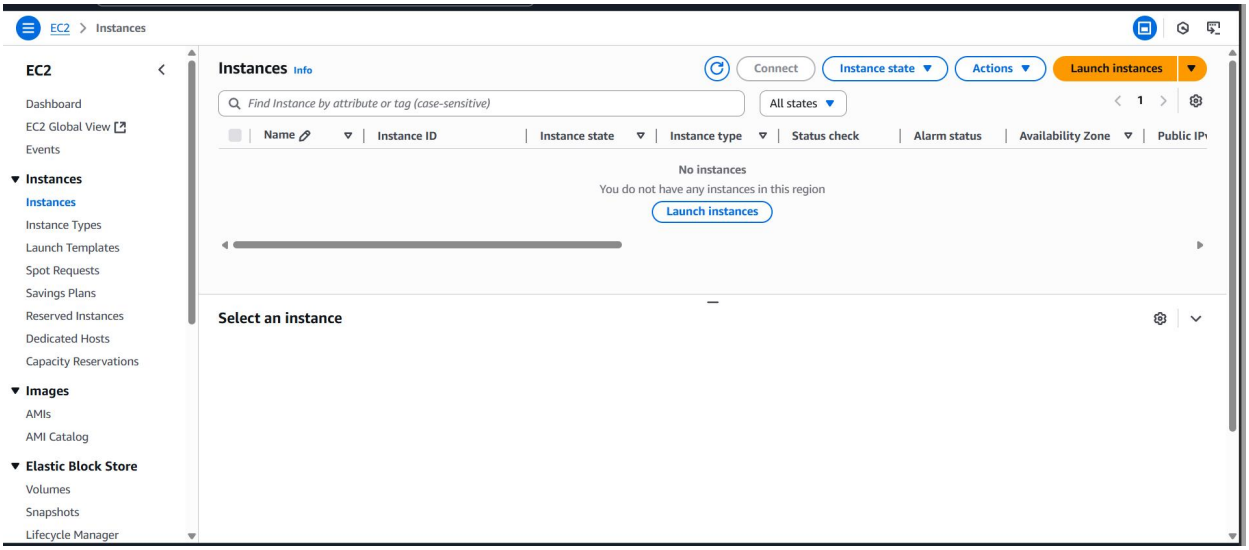


Launch an EC2 instance to host the Flask application.

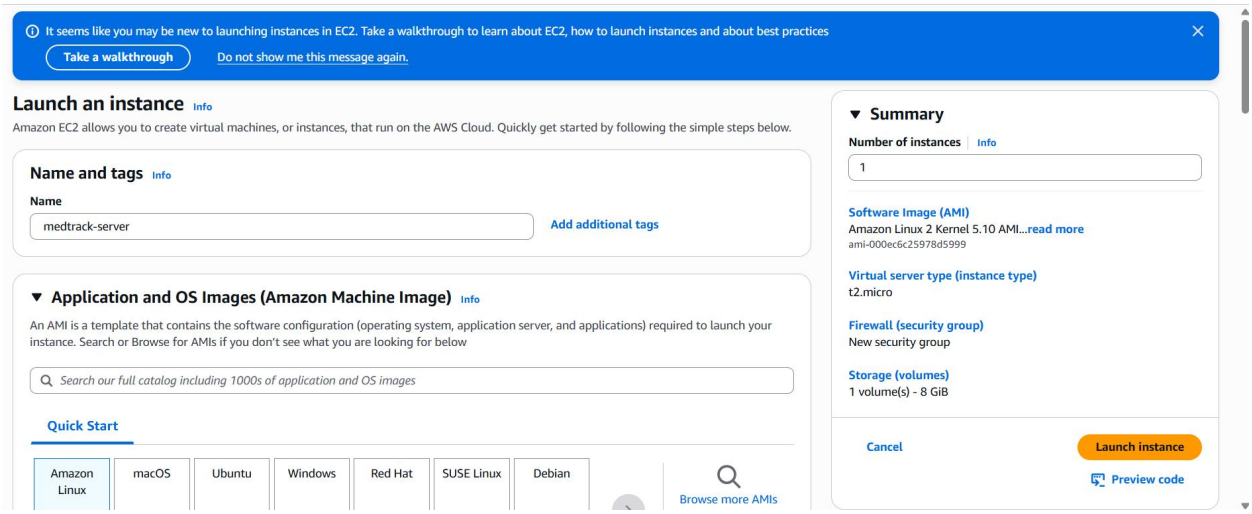
○ In the AWS Console, navigate to EC2 and launch a new instance.

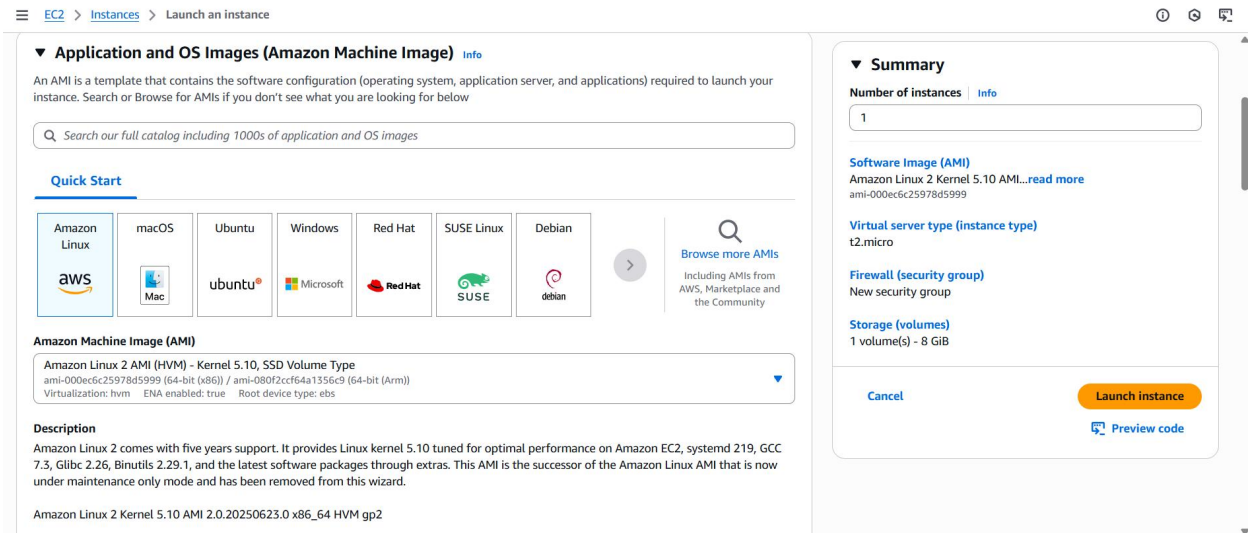


- **Click on Launch instance to launch EC2 instance**

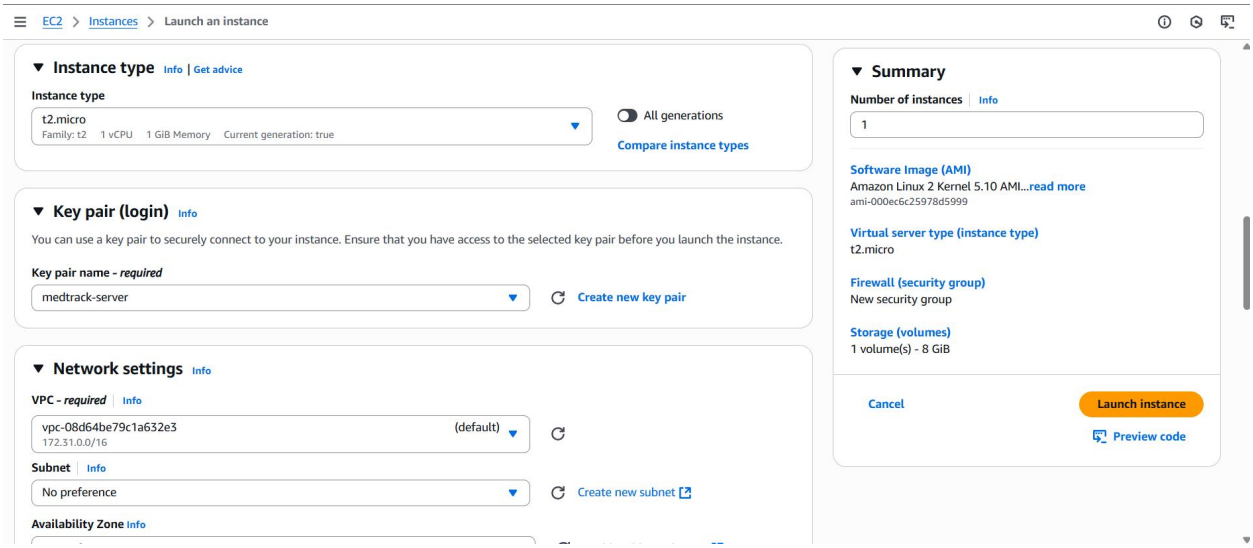


- **Launch EC2 instance (Amazon Linux 2/Ubuntu, t2.micro).**

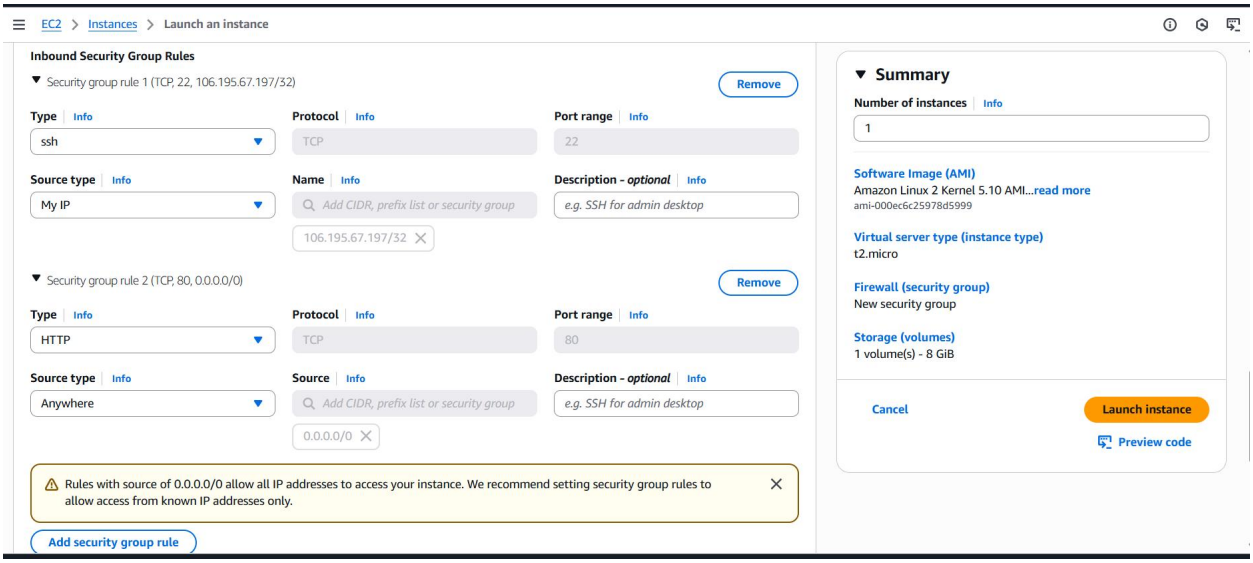




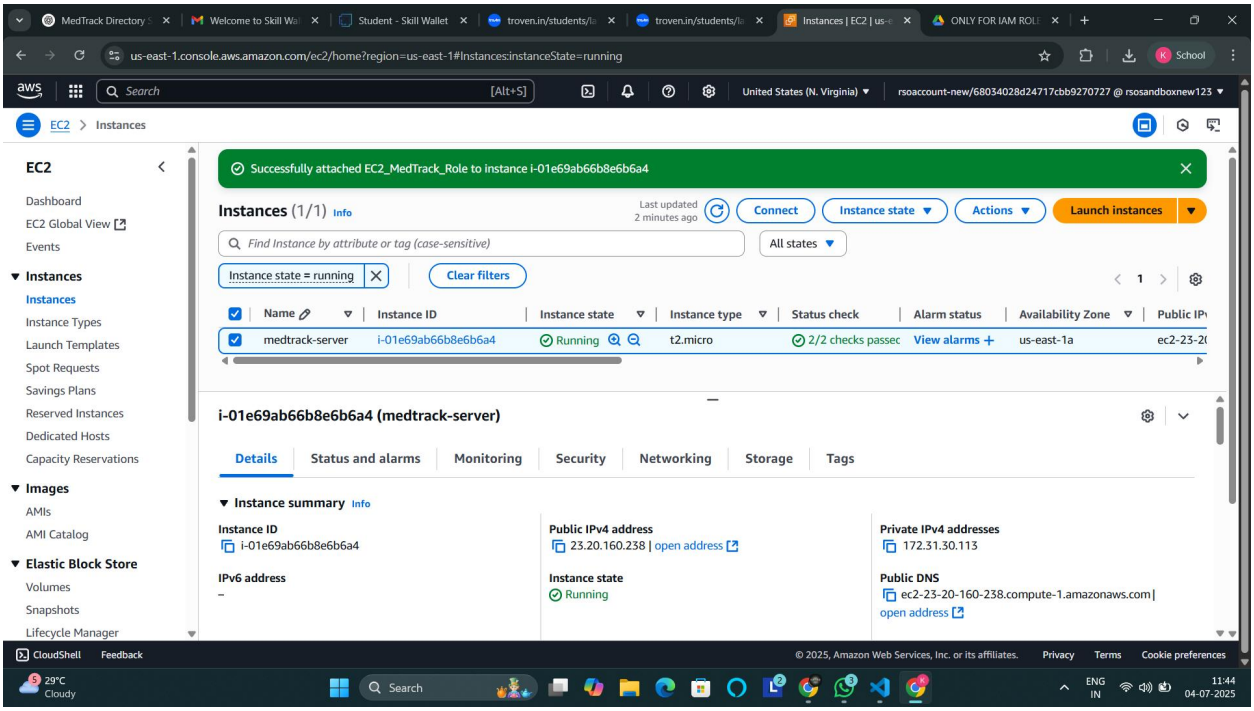
- **Assign an IAM Role and key pair.**



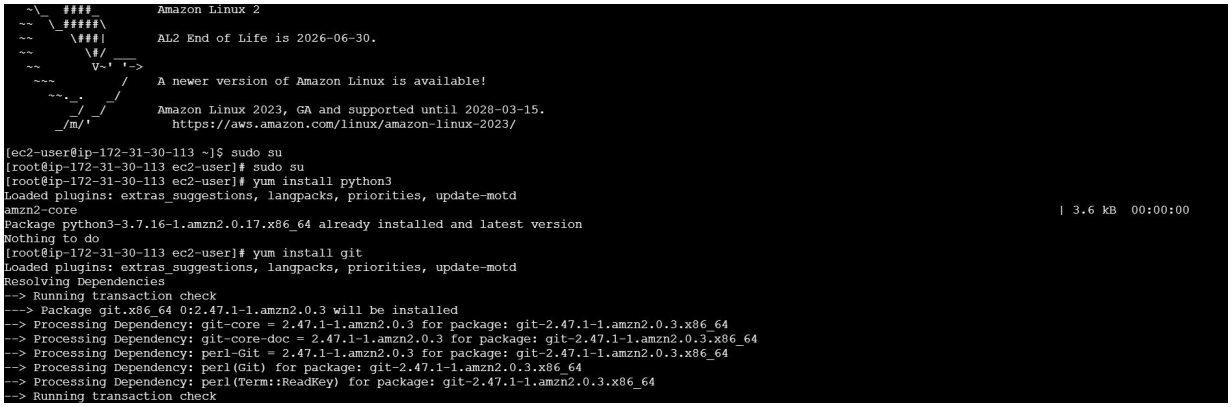
- **Configure security groups for HTTP/SSH.**



● 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 on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, navigate to the EC2 section in the AWS Management Console. Select the EC2 instance you wish to connect to. At the top of the EC2 Dashboard, click the Connect button. From the connection methods presented, choose EC2 Instance Connect. Finally, click Connect again, and a new browser-based terminal will open, allowing you to access your EC2 instance directly from your browser.



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



Milestone 7: Deployment on EC2

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

Clone Your Flask Project from GitHub:

Run: git clone https://github.com/pavanichebolu/MEDTRACK.git

Note: change your-github-username and your-repository-name with your credentials. here: 'git clone https://github.com/pavanichebolu/MEDTRACK.git'

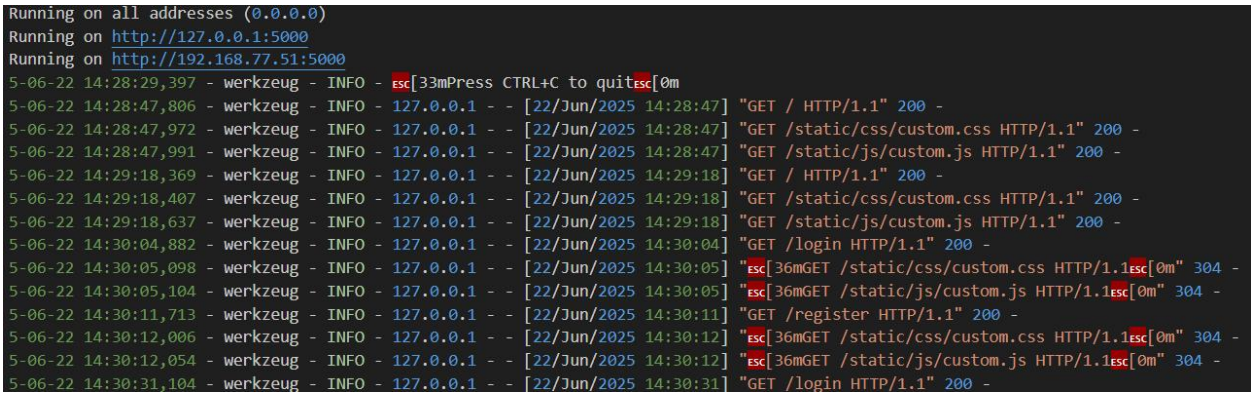
- This will download your project to the EC2 instance.
- To navigate to the project directory, run the following
- command: cd Medtrack.

Once inside the project directory, configure and run the Flask application by executing the following command with elevated privileges: Run the Flask Application.

- Run Flask app: sudo flask run --host=0.0.0.0 --port=5000

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://192.168.191.91:5000

Milestone 8: Testing and Deployment

- Verify registration, login, appointment booking, and SNS notifications.

Flask Application Structure & Code

App Initialization:

- Setup routes: register, login, dashboard, book appointment, view appointment, search, profile.

```python
@app.route('/')
def index():
    if is_logged_in():
        return redirect(url_for('dashboard'))
    return render_template('index.html')
```

- Connect to DynamoDB and SNS using boto3 with the correct region and ARN.

Routes:

- **Register:** Register the user, hash the password, and store it in DynamoDB.

```python
@app.route('/register', methods=['GET', 'POST'])
def register():
    if is_logged_in():
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        required_fields = ['name', 'email', 'password', 'confirm_password', 'age', 'gender', 'role']
        for field in required_fields:
            if not request.form.get(field):
                flash(f'Please enter {field}', 'danger')
                return render_template('register.html')
        if request.form['password'] != request.form['confirm_password']:
            flash('Passwords do not match', 'danger')
            return render_template('register.html')

        email = request.form['email'].lower()
        existing = user_table.get_item(Key={'email': email}).get('Item')
        if existing:
            flash('Email already registered', 'danger')
            return render_template('register.html')

        user_data = {
            'email': email,
            'name': request.form['name'],
            'password': generate_password_hash(request.form['password']),
            'age': request.form['age'],
            'gender': request.form['gender'],
            'role': request.form['role'].lower(),
            'created_at': datetime.utcnow().isoformat()
        }
        user_table.put_item(Item=user_data)
```

- **Login:** Authenticate and update login count.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if is_logged_in():
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        email = request.form.get('email', '').lower()
        password = request.form.get('password', '')
        role = request.form.get('role', '').lower()

        if not email or not password or not role:
            flash('All fields are required', 'danger')
            return render_template('login.html')

        user = user_table.get_item(Key={'email': email}).get('Item')
        if user and user['role'] == role and check_password_hash(user['password'], password):
            session['email'] = email
            session['role'] = role
            session['name'] = user.get('name', '')
            flash('Login successful!', 'success')
            return redirect(url_for('dashboard'))
        flash('Invalid email, password, or role', 'danger')
    return render_template('login.html')
```

- **Logout:** End session.

```python
@app.route('/logout')
def logout():
    session.clear()
    flash('Logged out successfully', 'success')
    return redirect(url_for('login'))
```

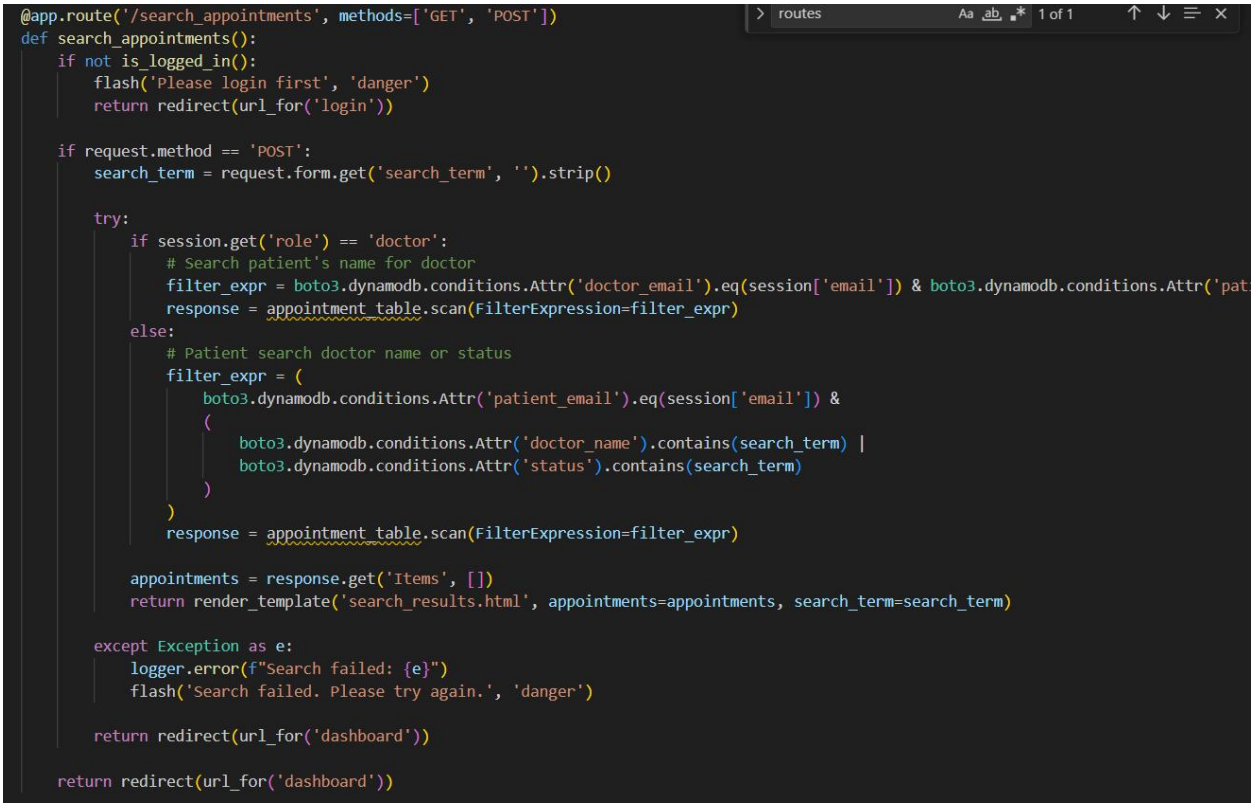
- **Book Appointment:** Collect and store appointment details, trigger SNS.

```python
@app.route('/book_appointment', methods=['GET', 'POST'])
def book_appointment():
    if not is_logged_in() or session.get('role') != 'patient':
        flash('Only patients can book appointments', 'danger')
        return redirect(url_for('login'))

    if request.method == 'POST':
        doctor_email = request.form.get('doctor_email')
        symptoms = request.form.get('symptoms')
        appointment_date = request.form.get('appointment_date')
        patient_email = session.get('email')

        if not doctor_email or not symptoms or not appointment_date:
            flash('Please fill all fields', 'danger')
            return redirect(url_for('book_appointment'))

        # Get doctor and patient info
        doctor = user_table.get_item(Key={'email': doctor_email}).get('Item')
        patient = user_table.get_item(Key={'email': patient_email}).get('Item')

        appointment_id = str(uuid.uuid4())
        appointment_item = {
            'appointment_id': appointment_id,
            'doctor_email': doctor_email,
            'doctor_name': doctor.get('name', 'Doctor'),
            'patient_email': patient_email,
            'patient_name': patient.get('name', 'Patient'),
            'symptoms': symptoms,
            'status': 'pending',
            'appointment_date': appointment_date,
            'created_at': datetime.utcnow().isoformat()
        }
```

- **View Appointments:** Retrieve data from DynamoDB.

```python
@app.route('/view_appointment/<appointment_id>', methods=['GET', 'POST'])
def view_appointment(appointment_id):
    if not is_logged_in():
        flash('Please login first', 'danger')
        return redirect(url_for('login'))

    try:
        response = appointment_table.get_item(Key={'appointment_id': appointment_id})
        appointment = response.get('Item')
        if not appointment:
            flash('Appointment not found', 'danger')
            return redirect(url_for('dashboard'))

        # Authorization check
        if session.get('role') == 'doctor' and appointment['doctor_email'] != session.get('email'):
            flash('Unauthorized access', 'danger')
            return redirect(url_for('dashboard'))
        if session.get('role') == 'patient' and appointment['patient_email'] != session.get('email'):
            flash('Unauthorized access', 'danger')
            return redirect(url_for('dashboard'))

        if request.method == 'POST' and session.get('role') == 'doctor':
            diagnosis = request.form.get('diagnosis')
            treatment_plan = request.form.get('treatment_plan')
            prescription = request.form.get('prescription')

            appointment_table.update_item(
                Key={'appointment_id': appointment_id},
                UpdateExpression="SET diagnosis = :d, treatment_plan = :t, prescription = :p, #s = :s, updated_at = :u",
                ExpressionAttributeValues={
                    ':d': diagnosis,
                    ':t': treatment_plan,
                    ':p': prescription,
                    ':s': 'completed',
                    ':u': datetime.utcnow().isoformat()
                },
```

- **Search:** Filter appointments.



Profile: View/edit personal data:

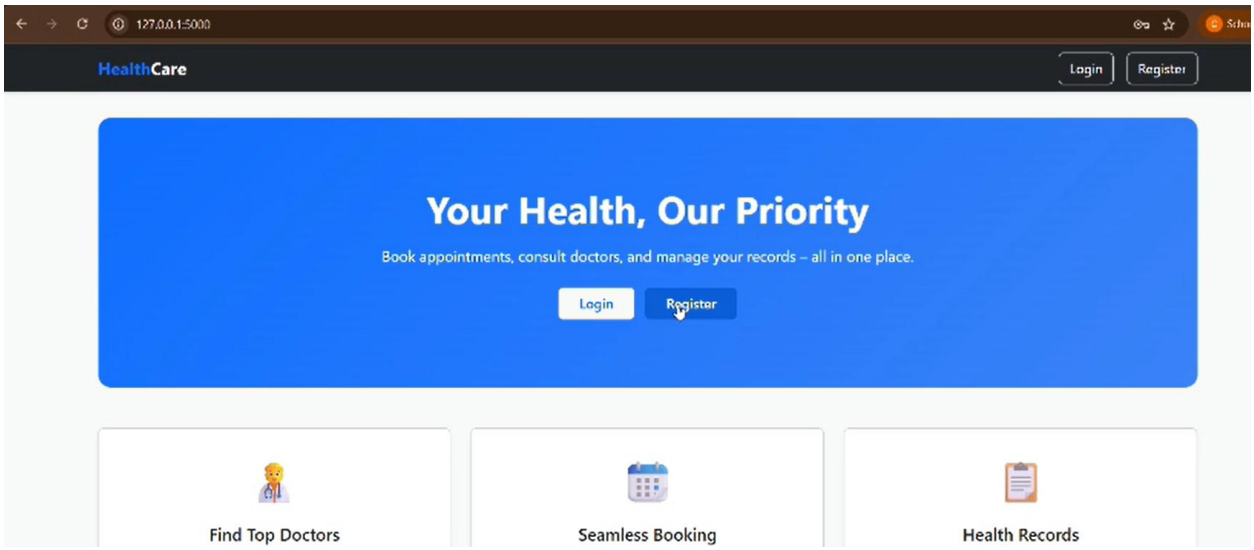
```python
@app.route('/profile', methods=['GET', 'POST'])
def profile():
    if not is_logged_in():
        flash('Please login first', 'danger')
        return redirect(url_for('login'))
    email = session.get('email')
    user = user_table.get_item(Key={'email': email}).get('Item')
    if not user:
        flash('User not found', 'danger')
        return redirect(url_for('logout'))

    if request.method == 'POST':
        name = request.form.get('name', user.get('name'))
        age = request.form.get('age', user.get('age'))
        gender = request.form.get('gender', user.get('gender'))
        update_expression = "SET #name = :name, age = :age, gender = :gender"
        expr_values = {':name': name, ':age': age, ':gender': gender}
        expr_names = {'#name': 'name'}

        # If doctor, allow specialization update
        if user['role'] == 'doctor' and 'specialization' in request.form:
            update_expression += ", specialization = :spec"
            expr_values[':spec'] = request.form['specialization']

        user_table.update_item(
            Key={'email': email},
            UpdateExpression=update_expression,
            ExpressionAttributeValues=expr_values,
            ExpressionAttributeNames=expr_names
        )
        session['name'] = name
        flash('Profile updated', 'success')
        return redirect(url_for('profile'))

    return render_template('profile.html', user=user)
```

Deployment Code:

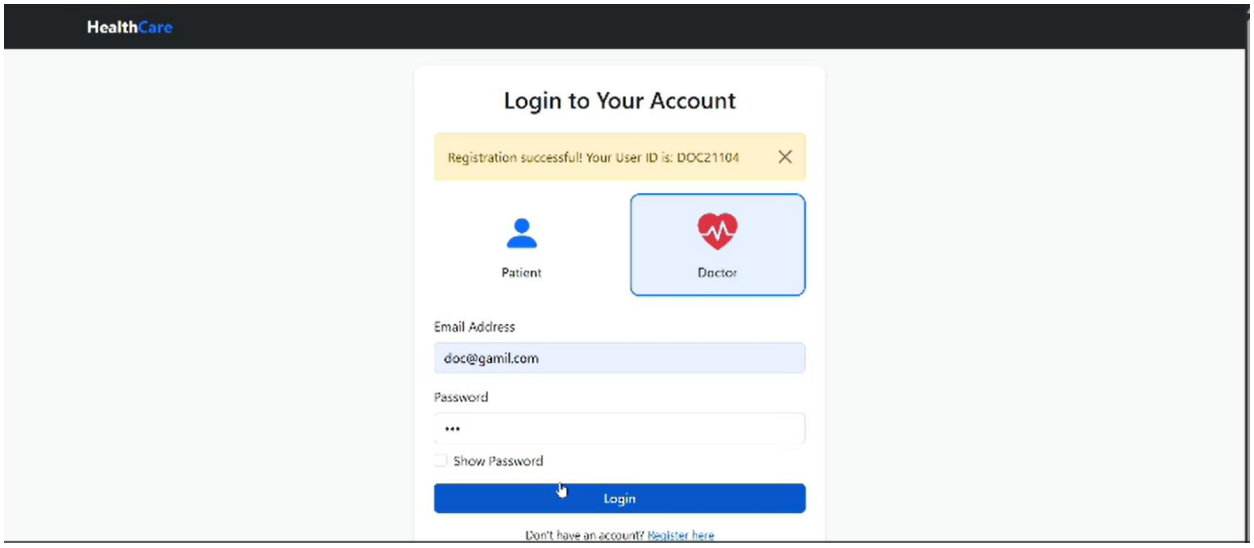
```python
if __name__ == '__main__':
    port = int(os.environ.get('PORT', 5000))
    debug_mode = os.environ.get('FLASK_ENV', '') == 'development'
    app.run(host='0.0.0.0', port=port, debug=debug_mode)
```

Functional Testing Summary

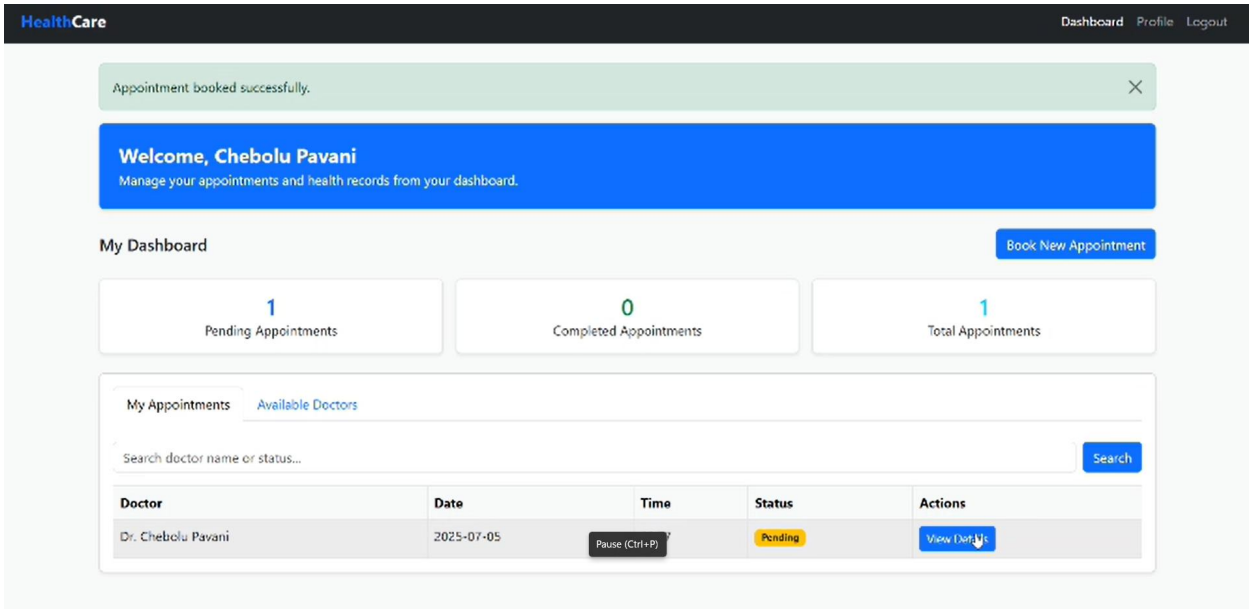
- **Home Page:** Entry point with navigation and responsive design.



- **Doctor & Patient Registration:** Collects and validates credentials.

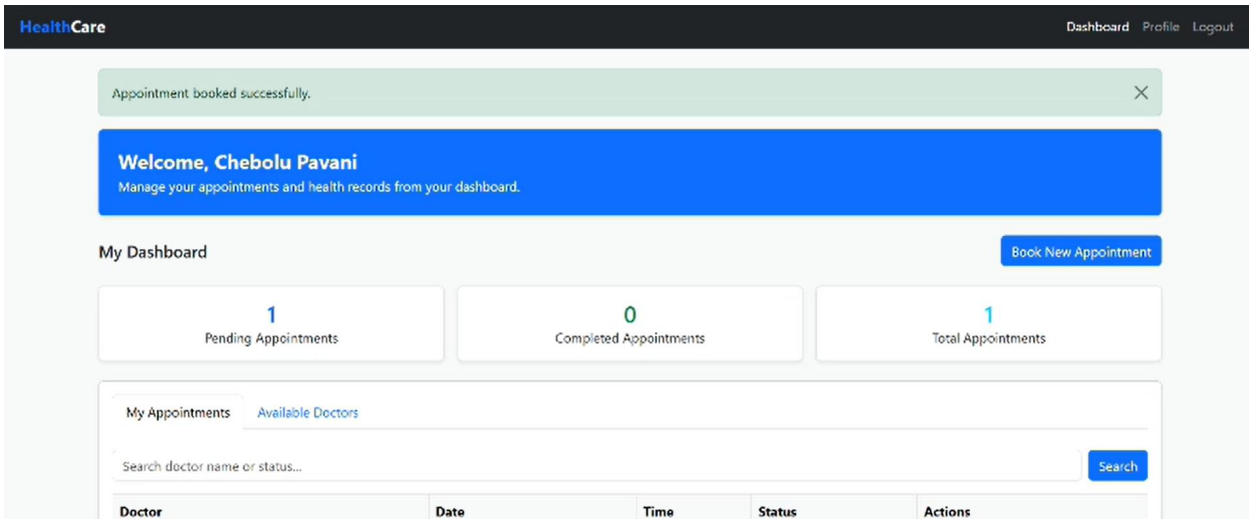


- **Login Pages:** Secures access and redirects to dashboards.



- **Dashboards:** Role-based UIs for managing appointments.

➔ **Doctor Dashboard**



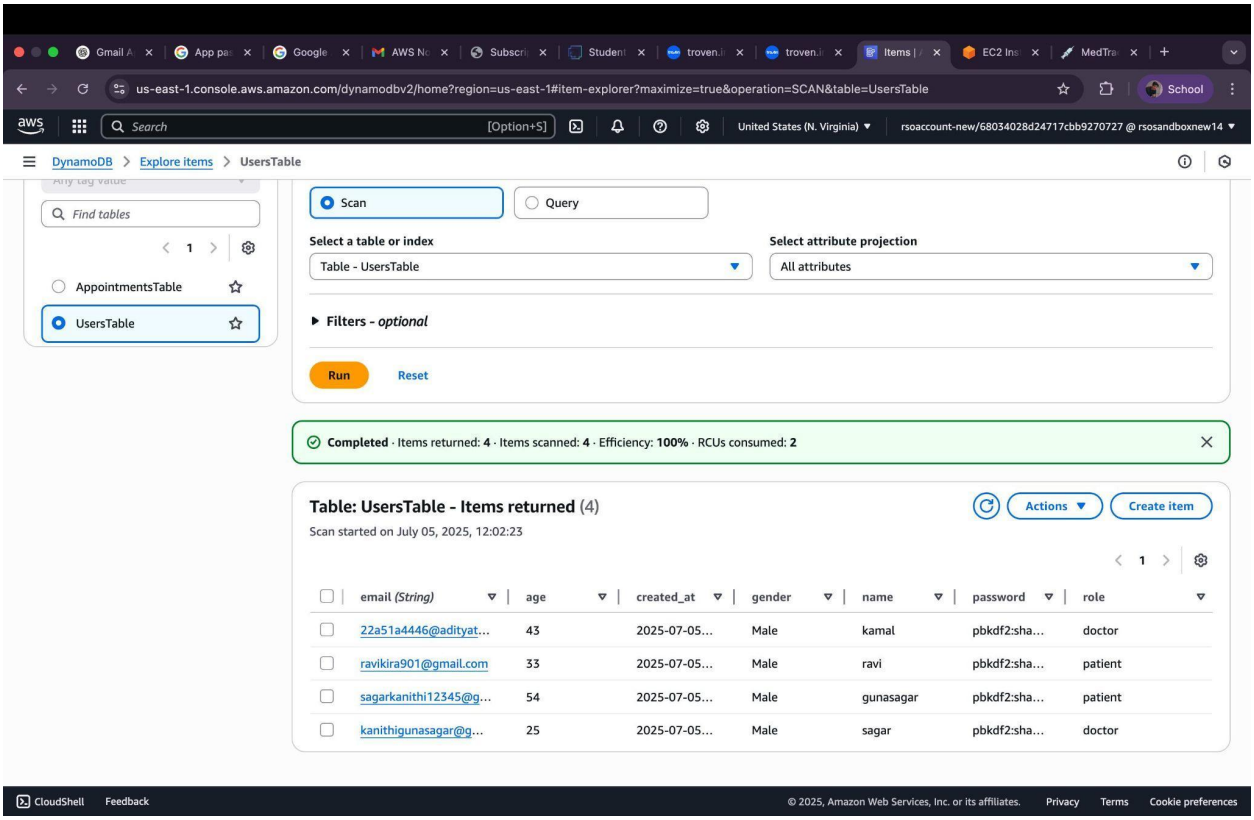
➔ **Patient Dashboard**

- **Search Feature:** Enables real-time filtering by status/date.

Database Updates

Users Table:

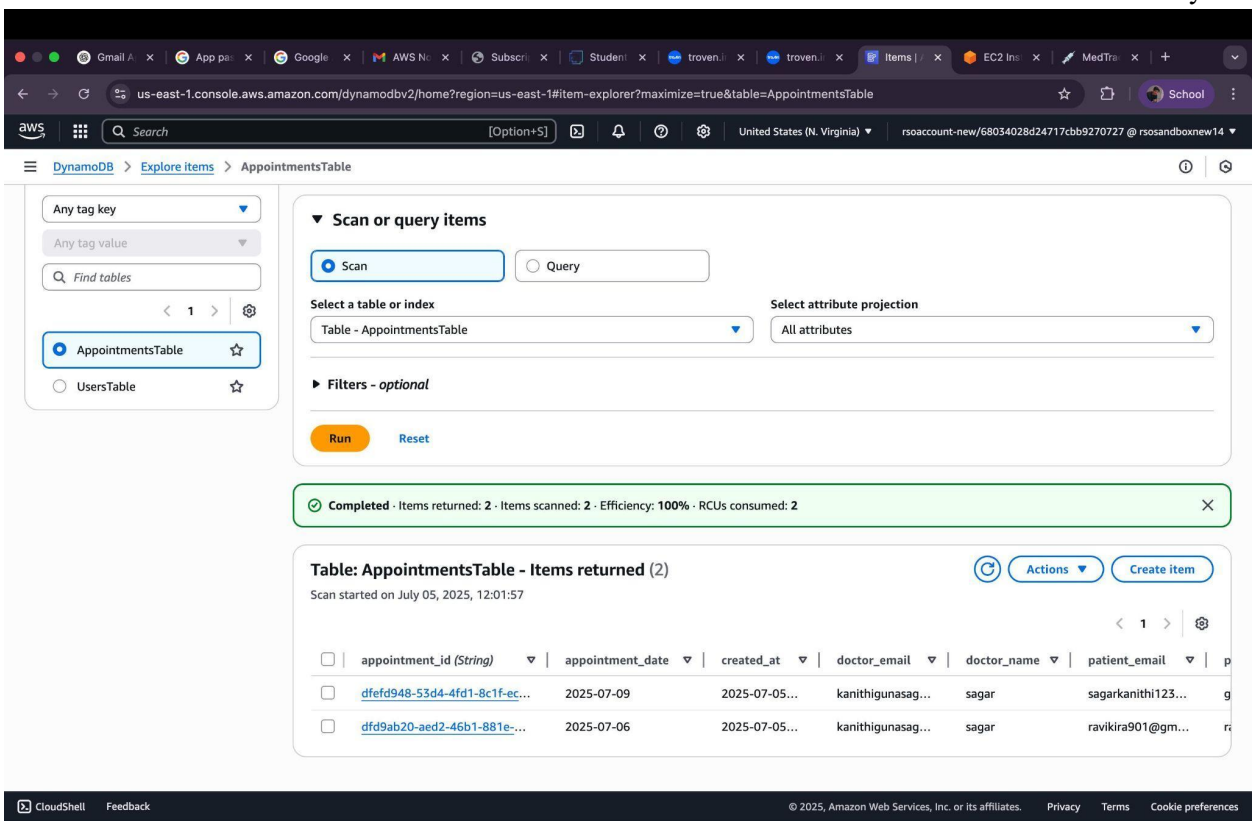
- Add a new user (doctor/patient)
- Update Profile
- Track active/inactive status.



Appointments Table:

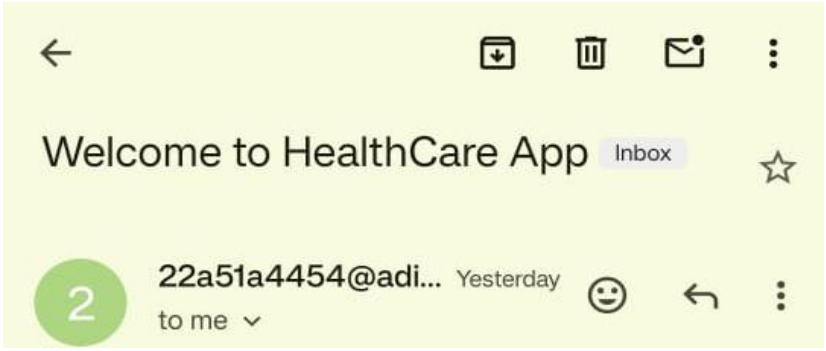
- Create new appointment
- Update Status

- Maintain history



Notifications

- Email confirmation to patients upon booking.



- Email alerts to doctors/admins upon new appointment.

Conclusion

MedTrack successfully demonstrates how cloud-native technologies can modernize healthcare delivery. Integrating Flask with AWS services such as EC2, DynamoDB, SNS, and IAM ensures secure, scalable, and responsive operations. MedTrack enhances patient care, optimizes appointment workflows, and supports reliable doctor-patient communication. This project stands as a powerful model of how technology can bridge operational gaps in real-world healthcare systems.