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 enabling diagnosis submissions. To address these challenges, the project utilizes Flask for backend development, AWS EC2 for hosting, and DynamoDB for managing data. MedTrack allows patients to register, log in, book appointments, and submit diagnosis reports online. The system ensures real-time notifications, enhancing communication between doctors and patients regarding appointments and medical submissions. Additionally, AWS Identity and Access Management (IAM) is employed to ensure secure access control to AWS resources, allowing only authorized users to access sensitive data. This cloud-based solution improves accessibility and efficiency in healthcare services for all users.

Scenarios:

Scenario 1: Efficient Appointment Booking System for Patients

In the MedTrack system, AWS EC2 provides a reliable infrastructure to manage multiple patients accessing the platform simultaneously. For example, a patient can log in, navigate to the appointment booking section, and easily submit a request for an appointment. Flask handles backend operations, efficiently retrieving and processing user data in real-time. The cloud-based architecture allows the platform to handle a high volume of appointment requests during peak periods, ensuring smooth operation without delays.

Scenario 2: Secure User Management with IAM

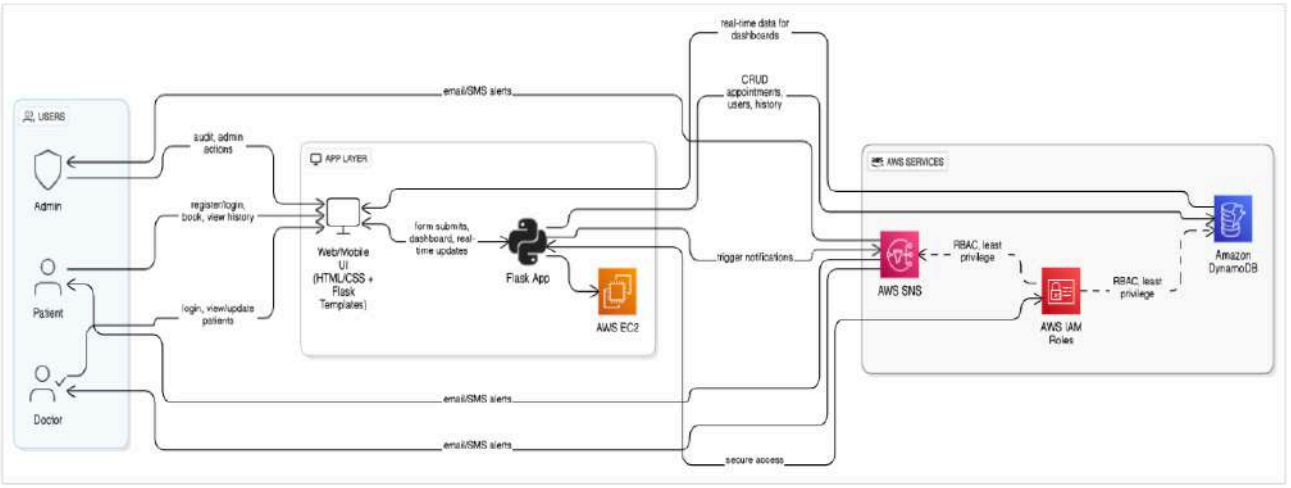
MedTrack utilizes AWS IAM to manage user permissions and ensure secure access to the system. For instance, when a new patient registers, an IAM user is created with specific roles and permissions to access only the features relevant to them. Doctors have their own IAM configurations, allowing them access to patient records and appointment details while maintaining strict security protocols. This setup ensures that sensitive data is accessible only to authorized users.

Scenario 3: Easy Access to Medical History and Resources

The MedTrack system provides doctors and patients with easy access to medical histories and relevant resources. For example, a patient logs in to view their medical history and upcoming appointments. Doctors diagnose conditions and prescribe necessary medications to their patients. Flask manages real-time data fetching from DynamoDB, while EC2 hosting ensures the platform performs seamlessly even when multiple users access it simultaneously, offering a smooth and uninterrupted user experience.

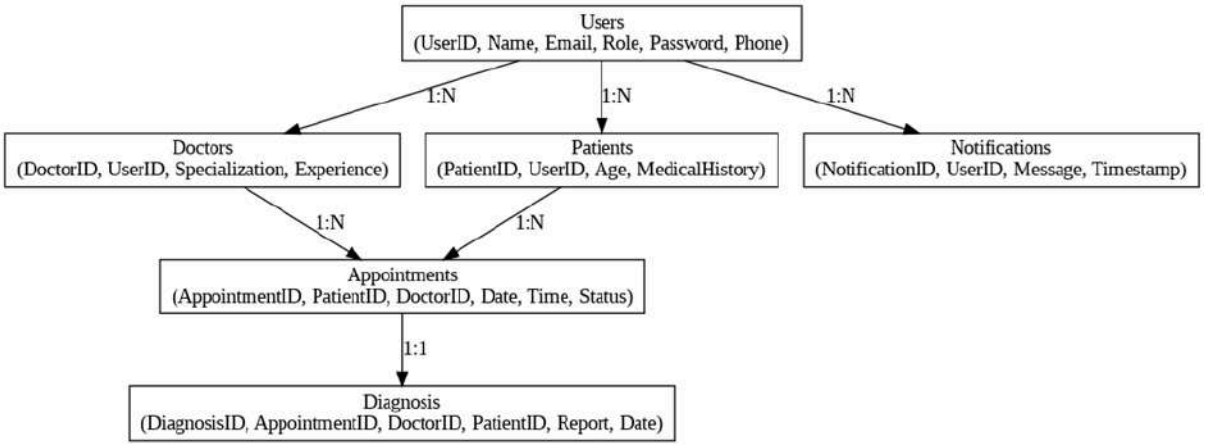
Architecture

This AWS-based architecture powers a scalable and secure web application using Amazon EC2 for hosting the backend, with a lightweight framework like Flask handling core logic. Application data is stored in Amazon DynamoDB, ensuring fast, reliable access, while user access is managed through AWS IAM for secure authentication and control. Real-time alerts and system notifications are enabled via Amazon SNS, enhancing communication and user engagement.



Entity Relationship (ER) Diagram

An ER (Entity-Relationship) diagram visually represents the logical structure of a database by defining entities, their attributes, and the relationships between them. It helps organize data efficiently by illustrating how different components of the system interact and relate. This structured approach supports effective database normalization, data integrity, and simplified query design.



Pre-requisites

- AWS Account Setup:
  https://docs.aws.amazon.com/accounts/latest/reference/getting-started.html
- AWS IAM (Identity and Access Management):
  https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
- AWS EC2 (Elastic Compute Cloud):
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- AWS DynamoDB:
  https://docs.aws.amazon.com/amazondynamodb/Introduction.html
- Amazon SNS:
  https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- Git Documentation:
  https://git-scm.com/doc
- VS Code Installation: (download the VS Code using the below link or you can get that in Microsoft store)
  https://code.visualstudio.com/download

Project WorkFlow

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 book request notifications.

Activity 3.2: Subscribe users and library staff to SNS email notifications.

Milestone 4. Backend Development and Application Setup

Activity 4.1: Develop the Backend Using Flask.

Activity 4.2: Integrate AWS Services Using boto3.

Milestone 5. IAM Role Setup

Activity 5.1: Create IAM Role

Activity 5.2: Attach Policies

Milestone 6. EC2 Instance Setup

Activity 6.1: Launch an EC2 instance to host the Flask application.

Activity 6.2: Configure security groups for HTTP, and SSH access.

Milestone 7. Deployment on EC2

Activity 7.1: Upload Flask Files

Activity 7.2: Run the Flask App

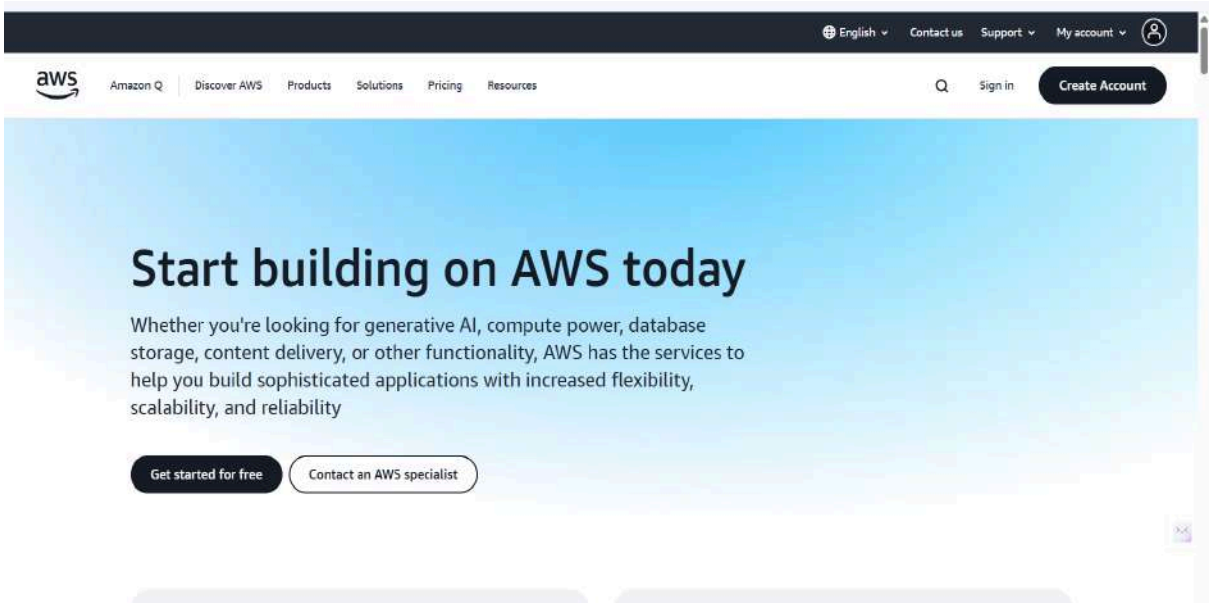
Milestone 8. Testing and Deployment

Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

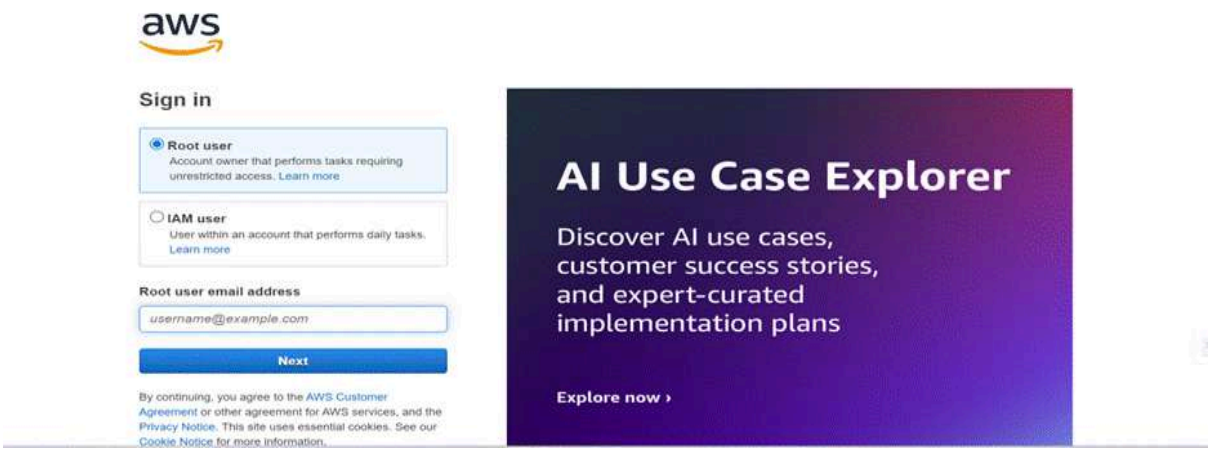
Milestone 1. AWS Account Setup and Login

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

- Sign up for an AWS account and configure billing settings.



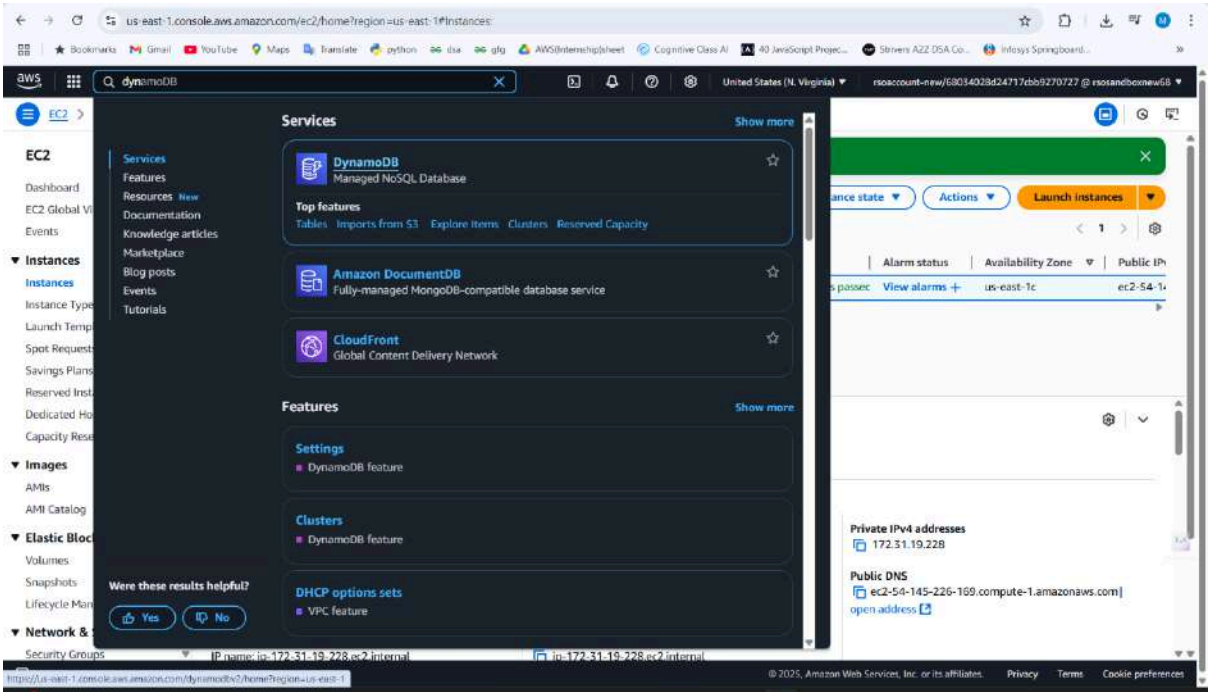
- **Activity 1.2:** Log in to the AWS Management Console
  - After setting up your account, log in to the AWS Management Console.

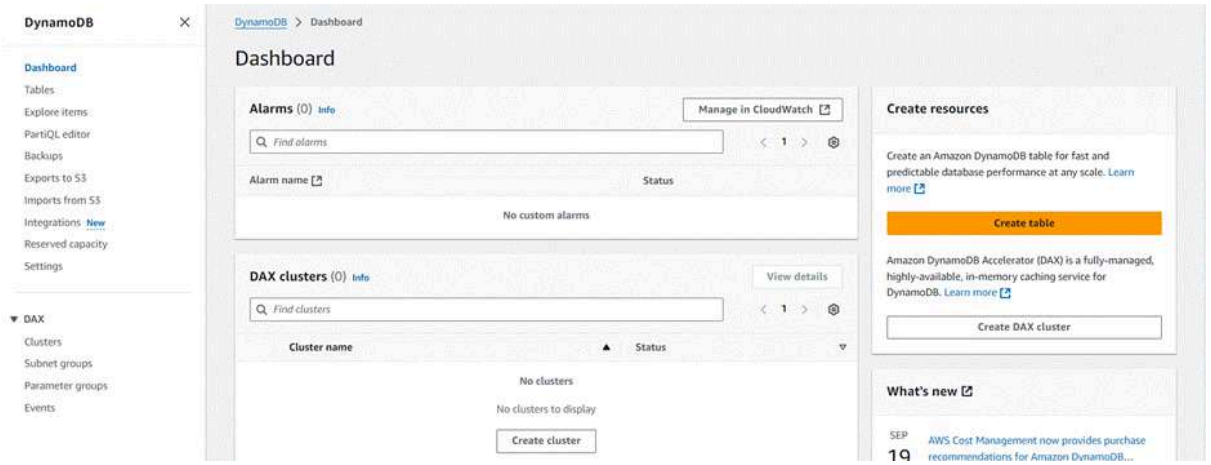


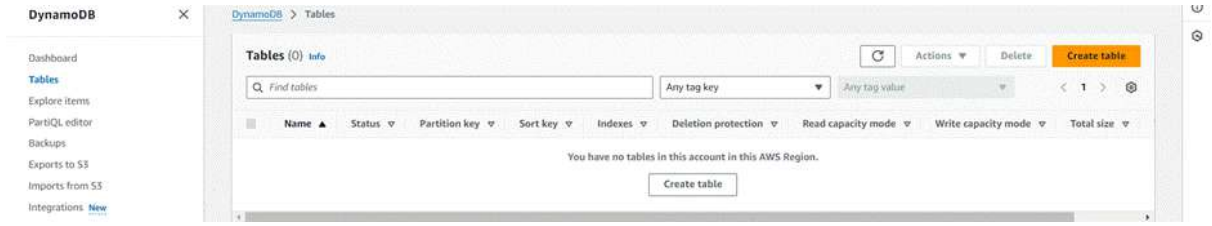
Milestone 2. DynamoDB Database Creation and Setup

Activity 2.1: Create a DynamoDB Table.

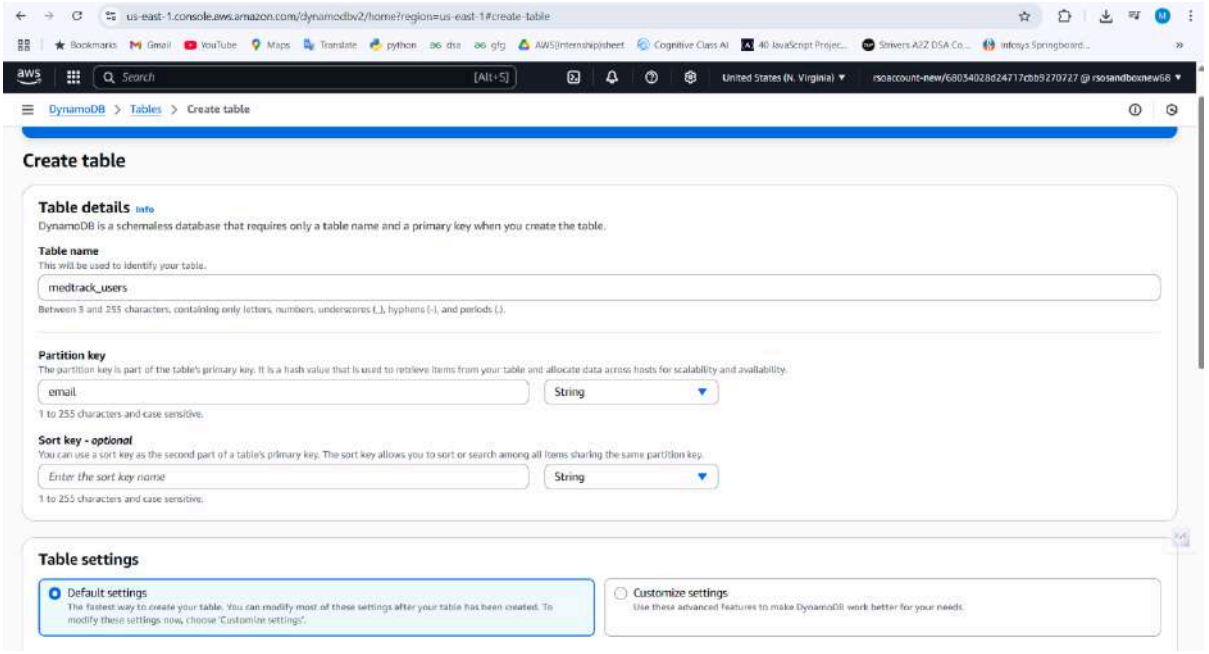
- In the AWS Console, navigate to DynamoDB and click on create tables.

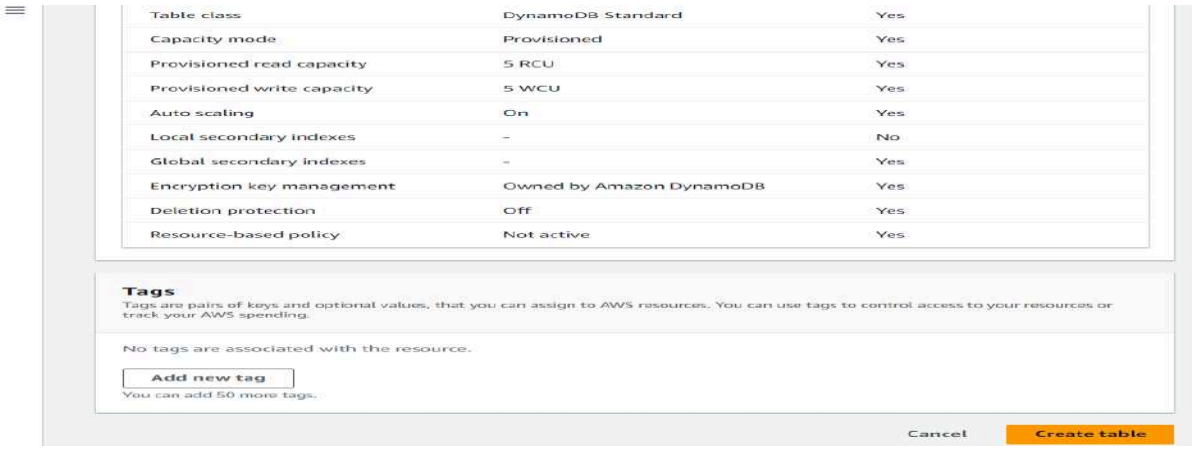




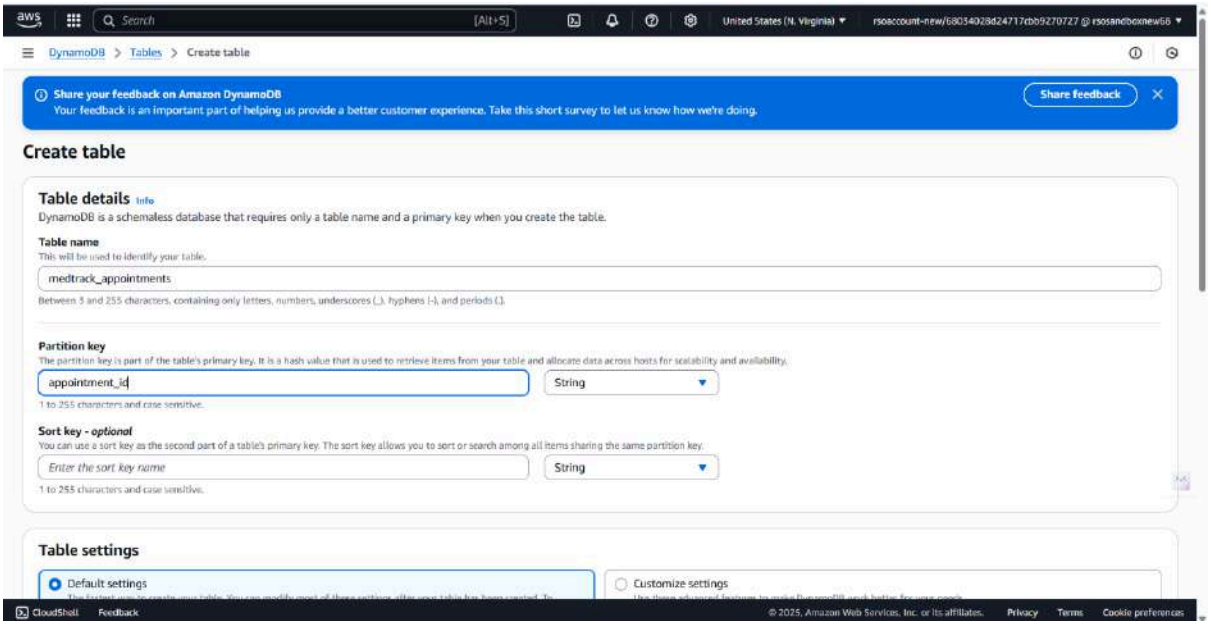


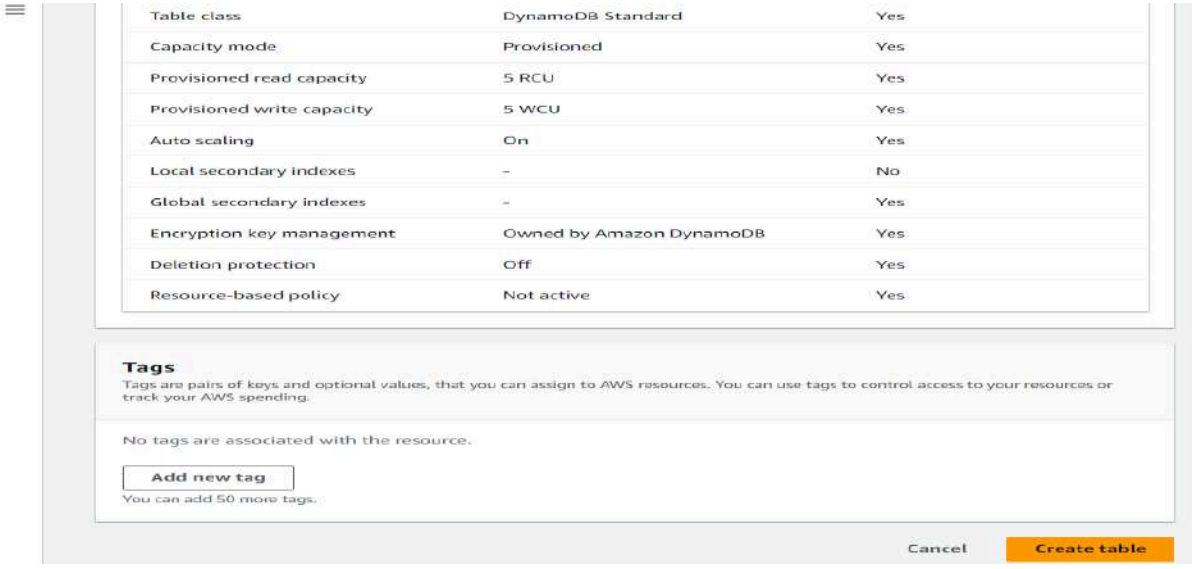
- **Activity 2.2**: Create a DynamoDB table for storing registration details and book requests.
    - Create medtrack_users table with partition key “Email” with type String and click on create tables.



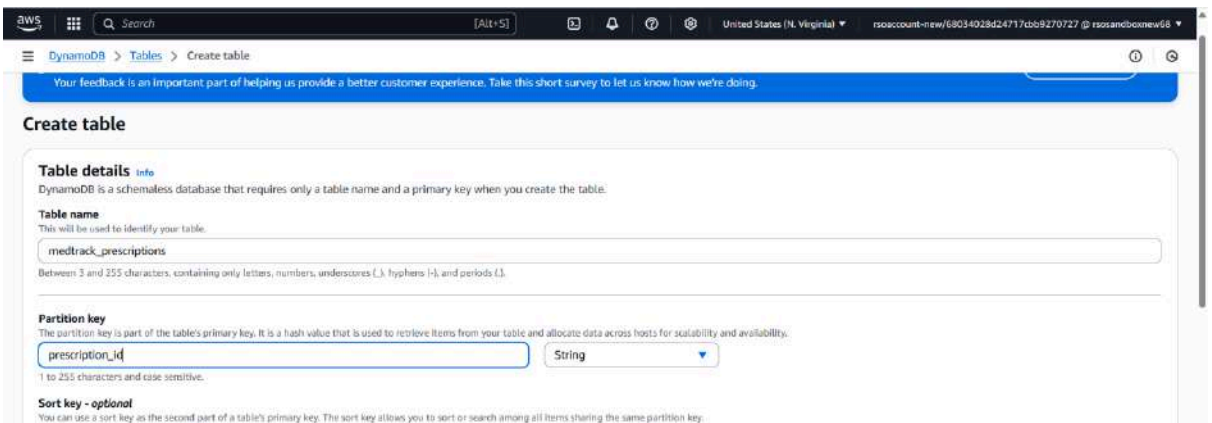


- Create medtrack_appointments Table with partition key “appointment_id” with type String and click on create tables.

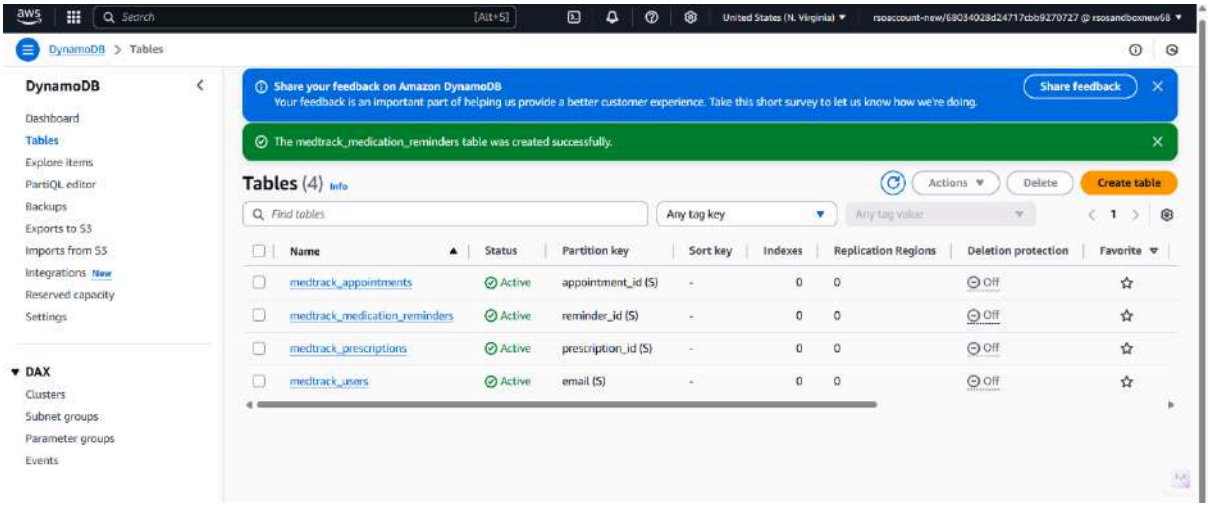




- Create medtrack_prescriptions Table with partition key “prescription_id” with type String and click on create tables.



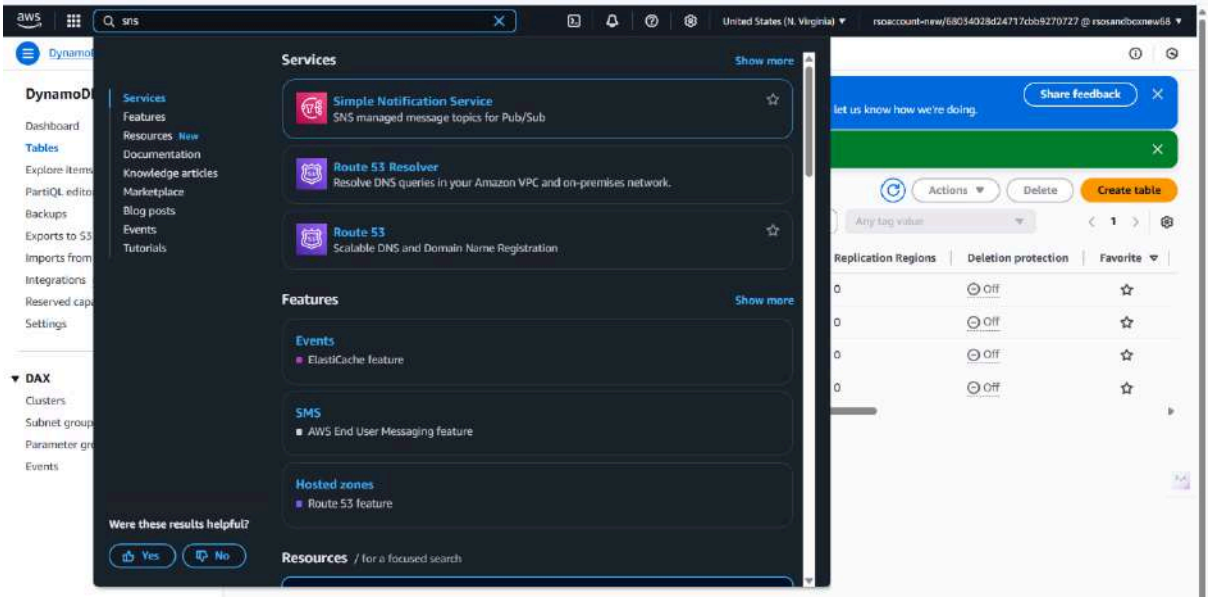
- Similarly Create medtrack_medication_remainders Table with partition key “reminder_id” with type String and click on create tables.

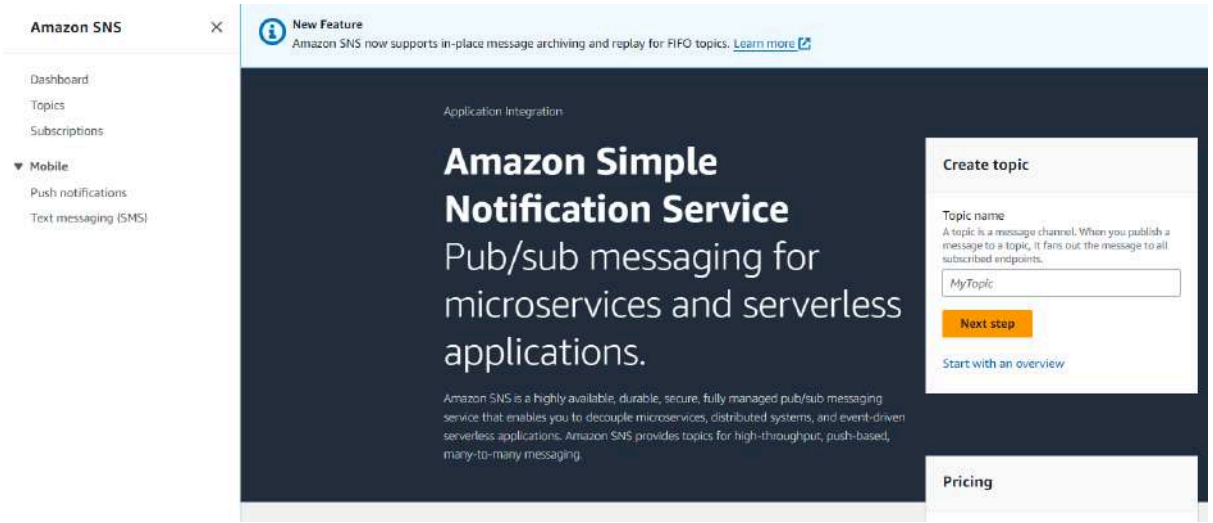


Milestone 3. SNS Notification Setup

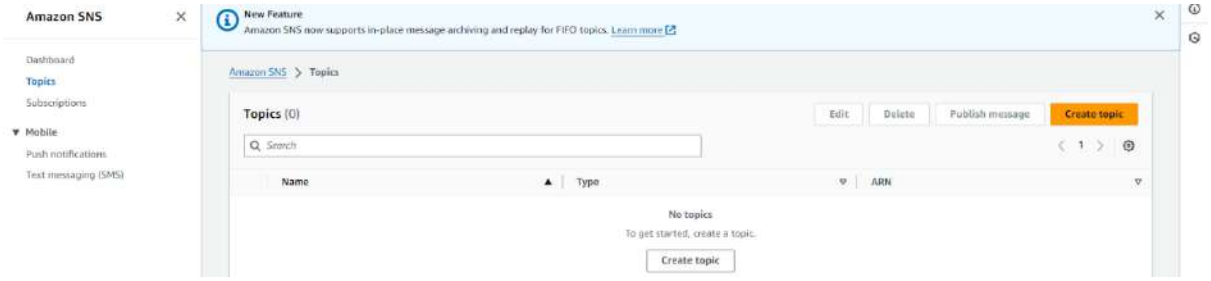
Activity 3.1: Create SNS topics for book request notifications.

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

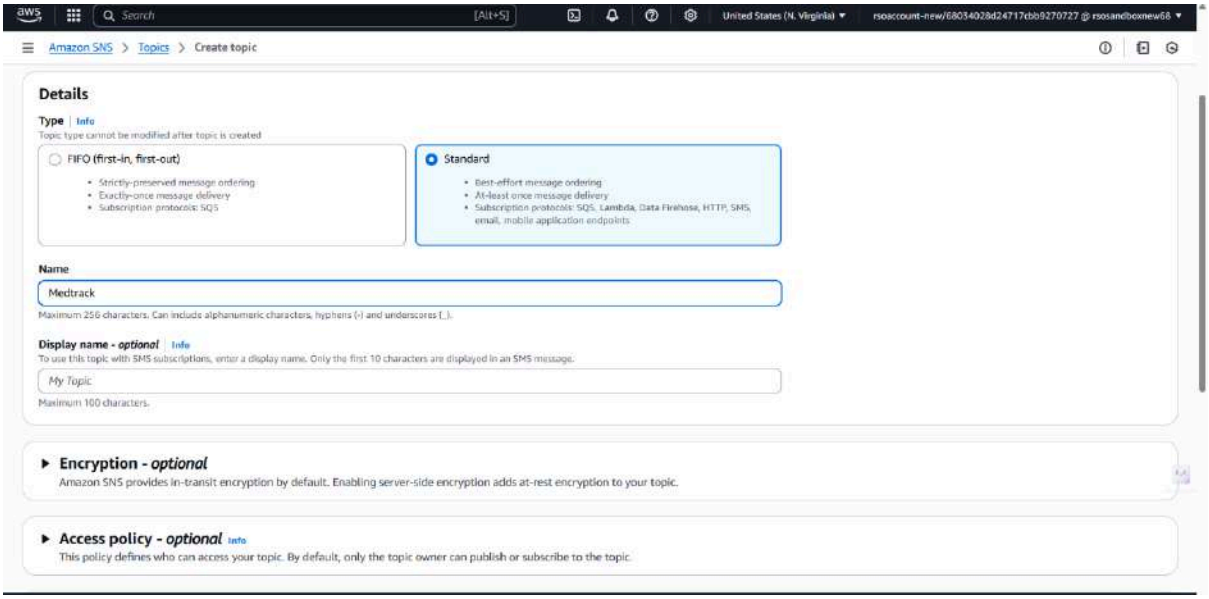


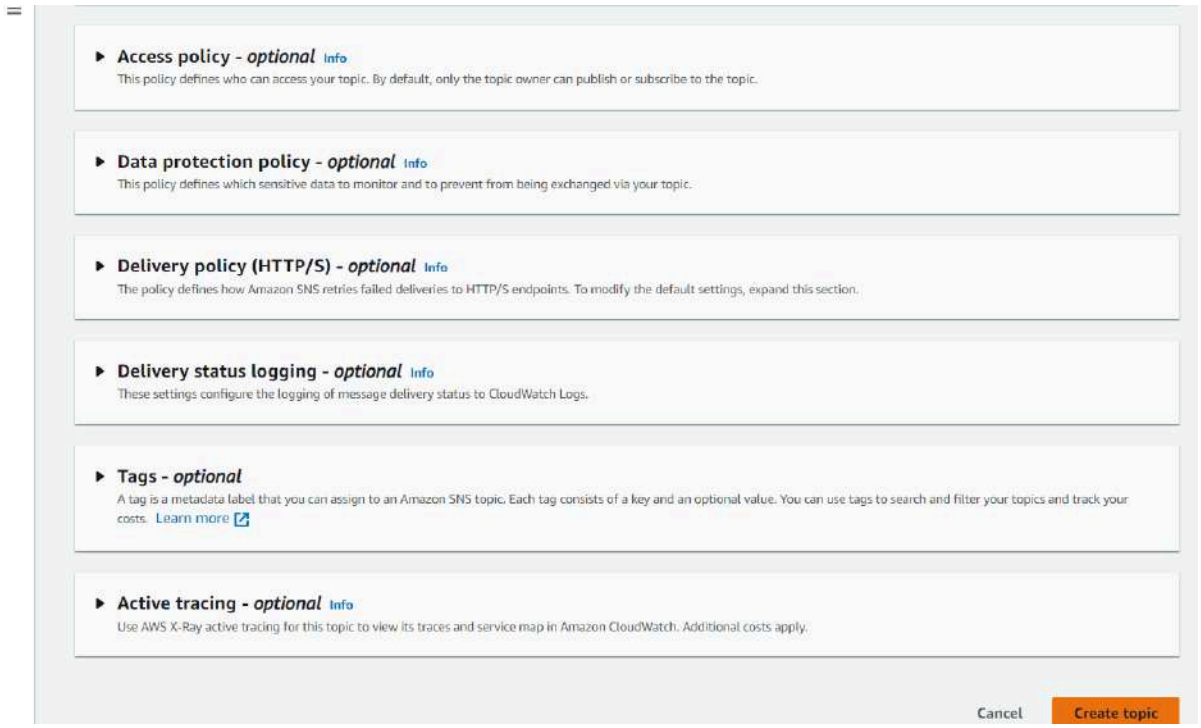


- Click on **Create Topic** and choose a name for the topic.

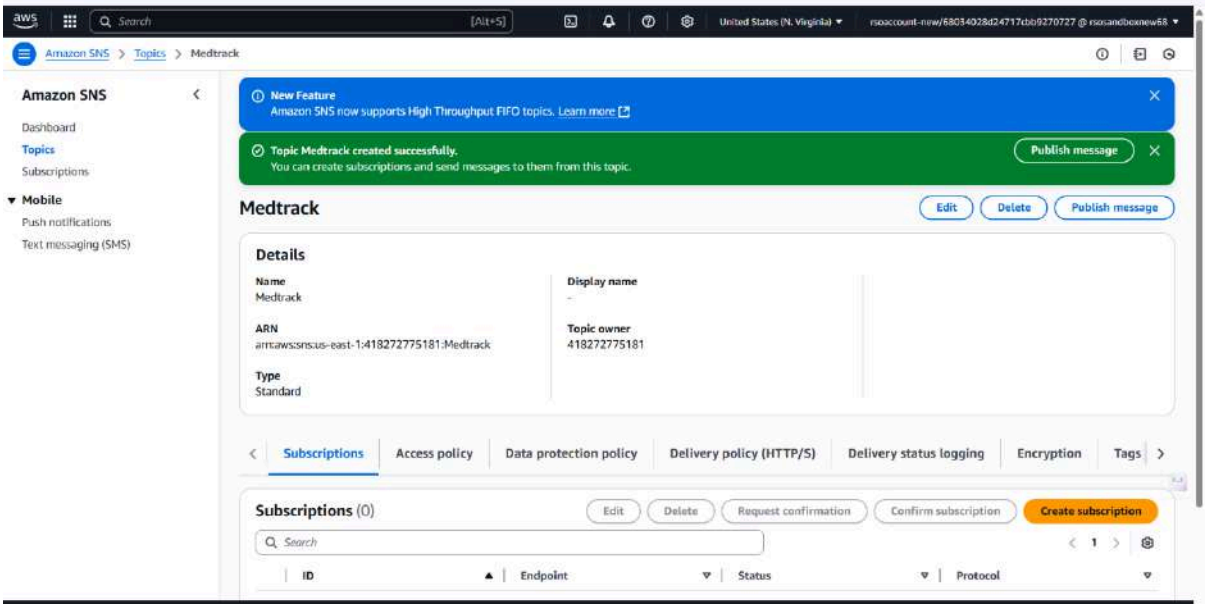


- Choose Standard type for general notification use cases and Click on Create Topic.

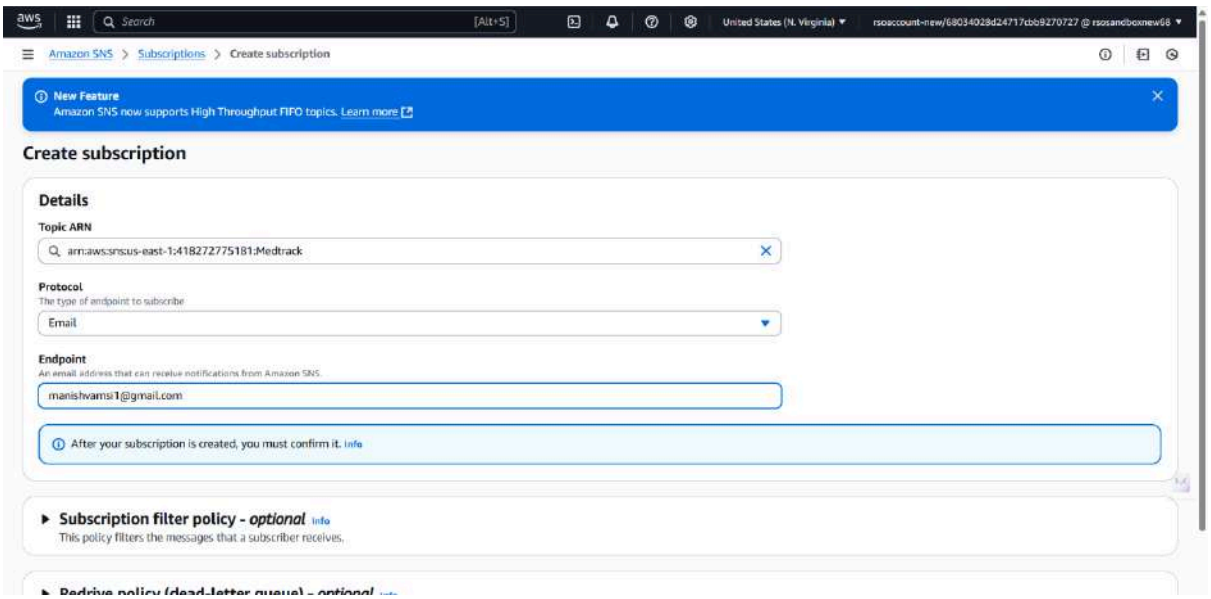




- Configure the SNS topic and note down the **Topic ARN**.

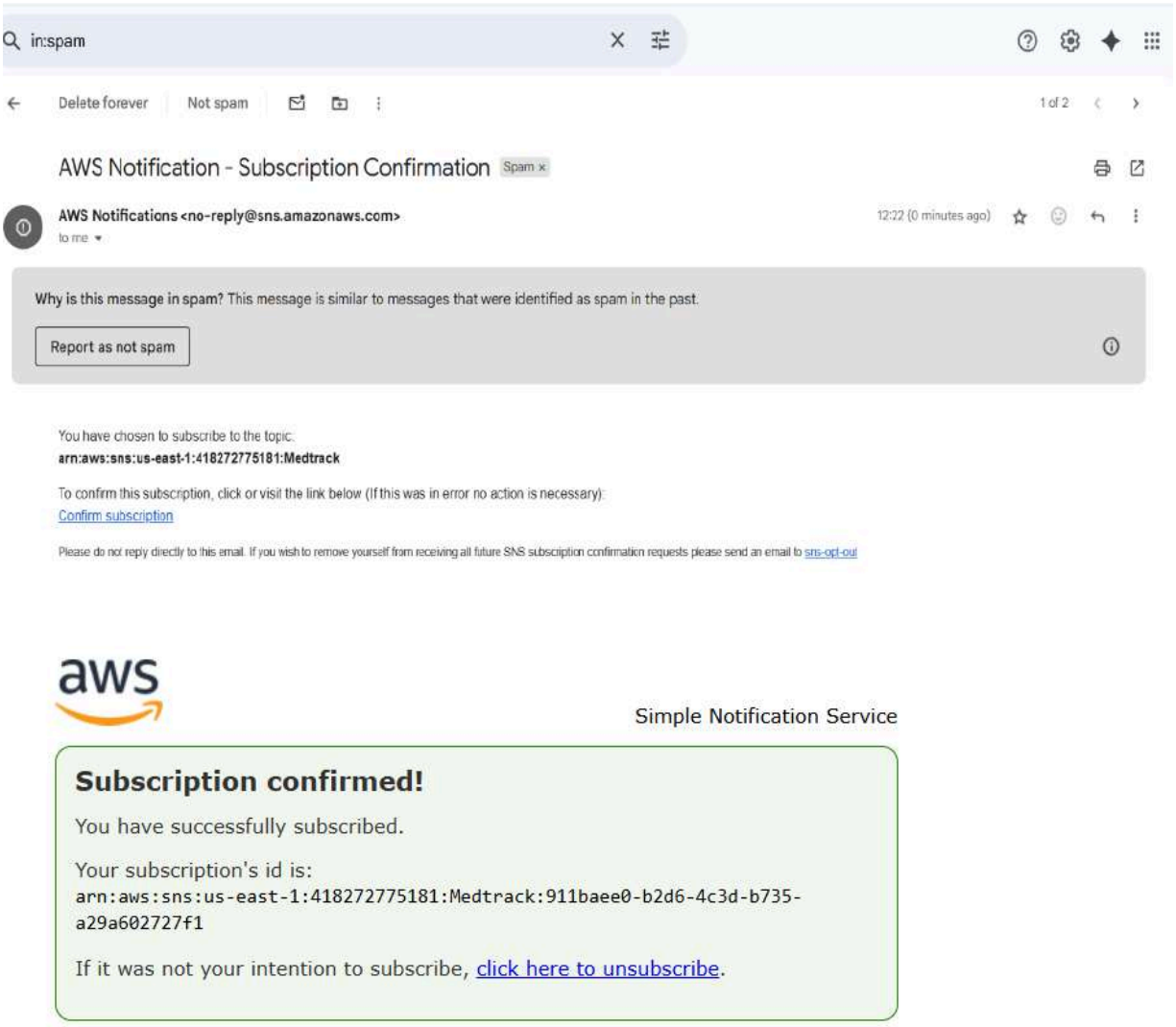


- **Activity 3.2**: Subscribe users and staff to relevant SNS topics to receive real-time notifications when a book request is made.
    - Subscribe users (or admin staff) to this topic via email. When a book request is made, notifications will be sent to the subscribed emails**.**



- After subscription request for the mail confirmation

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

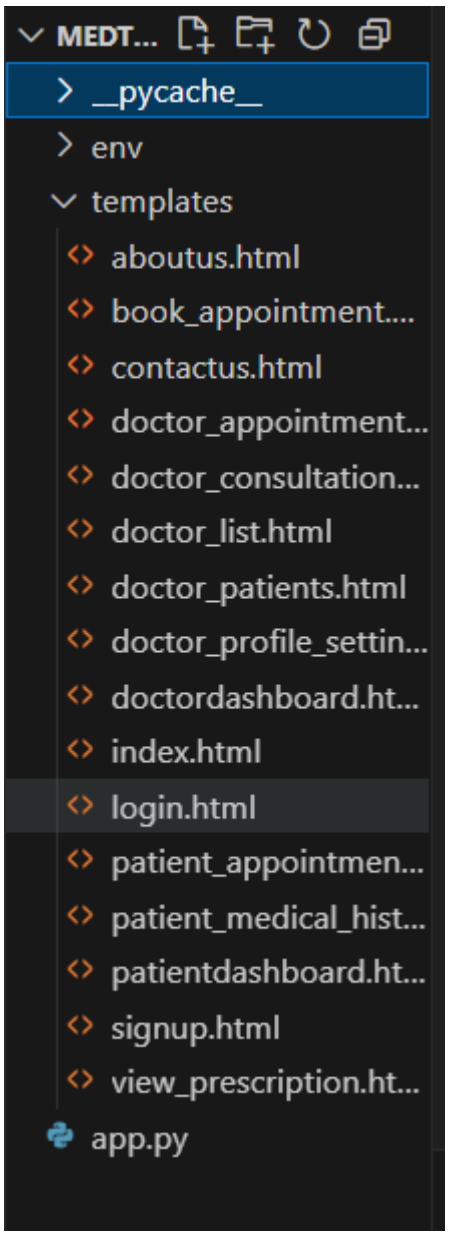


- Successfully done with the SNS mail subscription and setup, store the ARN link.

Milestone 4. Backend Development and Application Setup

Activity 4.1: Develop the Backend Using Flask.

- File Explorer Structure



- **Description of the code:**
  - **Flask App Initialization:**

```
from flask import Flask, render_template, request, redirect, url_for, session, flash
from datetime import datetime, timedelta
from functools import wraps
import os
import boto3
from botocore.exceptions import ClientError
```

Description: import essential libraries including Flask utilities for routing, Boto3 for DynamoDB operations, SMTP and email modules for sending mails, and Bcrypt for password hashing and verification.

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

- **Dynamodb Setup:**

```
try:
    # boto3 will automatically use credentials from an attached IAM Role on EC2
    # or from environment variables (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY) / AWS CLI config locally.
    dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
    sns_client = boto3.client('sns', region_name=AWS_REGION)

    users_table = dynamodb.Table(USERS_TABLE)
    appointments_table = dynamodb.Table(APPOINTMENTS_TABLE)
    medical_records_table = dynamodb.Table(MEDICAL_RECORDS_TABLE)

    # Test DynamoDB connection by attempting to access a table (e.g., describe table)
    users_table.load()
    appointments_table.load()
    medical_records_table.load()
    print(f"Successfully connected to AWS DynamoDB in region {AWS_REGION} and SNS.")
```

Description: Initialize the DynamoDB resource for the ap-south-1 region and set up access to the Users and Requests tables for storing user details and book requests.

- **SNS Connection**

```
message = (f"Appointment cancelled: Patient {patient_email}'s appointment "
           f"with Dr. {appointment['doctor_name']} on {appointment['date']} "
           f"at {appointment['time']} has been cancelled.")
try:
    sns_client.publish(TopicArn=SNS_TOPIC_ARN, Message=message, Subject="Medtrack Appointment Cancelled")
    logger.info(f"SNS notification sent for appointment cancellation: {appointment_id}.")
except Exception as sns_e:
    logger.error(f"Failed to send SNS notification for cancellation: {sns_e}")
```

```
    message = (f"New appointment booked: Patient {patient_name} ({patient_email}) "
               f"with Dr. {doctor_name} on {appointment_date} at {appointment_time} "
               f"for reason: {reason}.")
    try:
        sns_client.publish(TopicArn=SNS_TOPIC_ARN, Message=message, Subject="New Medtrack Appointment")
        logger.info(f"SNS notification sent for new appointment: {new_appointment['appointment_id']}.")
    except Exception as sns_e:
        logger.error(f"Failed to send SNS notification for appointment booking: {sns_e}")
```

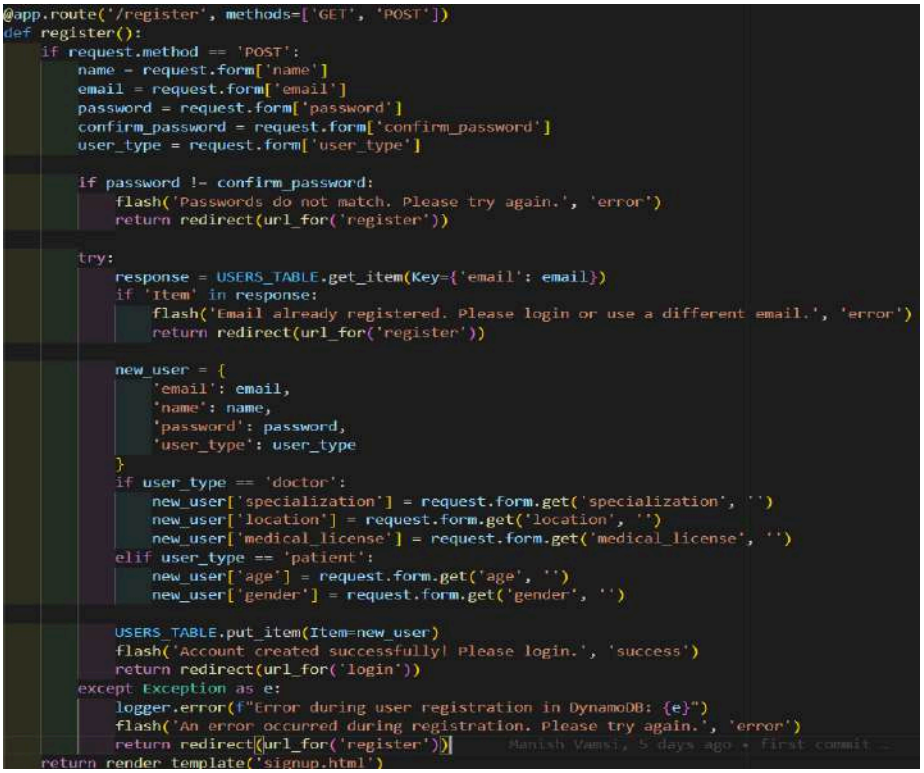
```
message = (f"Your appointment with Dr. {appointment['doctor_name']} "
           f"on {appointment['date']} at {appointment['time']} has been updated to: {new_status}.")
try:
    sns_client.publish(TopicArn=SNS_TOPIC_ARN, Message=message, Subject="Medtrack Appointment Update")
    logger.info(f"SNS notification sent for appointment status update: {appointment_id}.")
except Exception as sns_e:
    logger.error(f"Failed to send SNS notification for status update: {sns_e}")
```

Description: Configure SNS to send notifications when a book request is submitted. Paste your stored ARN link in the sns_topic_arn space, along with the region_name where the SNS topic is created.

- **Routes for Web Pages:**
- **Index Route:**

```python
@app.route('/')
def index():
    return render_template('index.html')
```

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



```python
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
        password = request.form['password']
        confirm_password = request.form['confirm_password']
        user_type = request.form['user_type']

        if password != confirm_password:
            flash('Passwords do not match. Please try again.', 'error')
            return redirect(url_for('register'))

        try:
            response = USERS_TABLE.get_item(Key={'email': email})
            if 'Item' in response:
                flash('Email already registered. Please login or use a different email.', 'error')
                return redirect(url_for('register'))

            new_user = {
                'email': email,
                'name': name,
                'password': password,
                'user_type': user_type
            }
            if user_type == 'doctor':
                new_user['specialization'] = request.form.get('specialization', '')
                new_user['location'] = request.form.get('location', '')
                new_user['medical_license'] = request.form.get('medical_license', '')
            elif user_type == 'patient':
                new_user['age'] = request.form.get('age', '')
                new_user['gender'] = request.form.get('gender', '')

            USERS_TABLE.put_item(Item=new_user)
            flash('Account created successfully! Please login.', 'success')
            return redirect(url_for('login'))
        except Exception as e:
            logger.error(f"Error during user registration in DynamoDB: {e}")
            flash('An error occurred during registration. Please try again.', 'error')
            return redirect(url_for('register'))
    return render_template('signup.html')
```

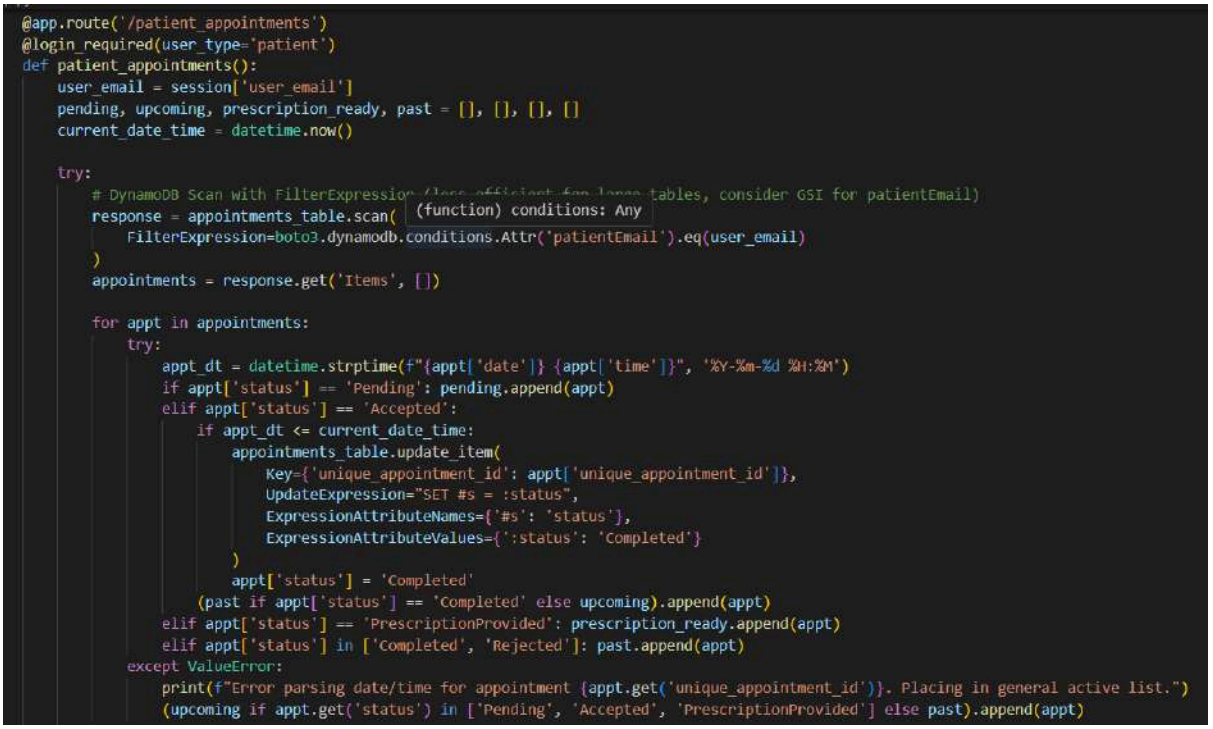
- **Login Route (GET/POST):** The **login route** handles user authentication by verifying credentials stored in **DynamoDB**. Upon successful login, it increments the **login count** and redirects the user to their dashboard.

```python
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if users_table is None:
        flash('Database connection not established. Cannot sign up.', 'error')
        return render_template('signup.html', error='Database connection error.')
    if request.method == 'POST':
        name, email, password, confirm_password, user_type = request.form['signupName'], request.form['signupEmail'], request.form['signupPassword']
        if not all([name, email, password, confirm_password, user_type]) or password != confirm_password:
            return render_template('signup.html', error='All fields are required and passwords must match.')
        try:
            response = users_table.get_item(Key={'email': email})
            if response.get('Item'):
                return render_template('signup.html', error='Email already registered. Please login or use a different email.')

            new_user_document = {'email': email, 'password': password, 'user_type': user_type, 'name': name}
            if user_type == 'patient':
                dob, gender = request.form.get('dateOfBirth'), request.form.get('gender')
                if not all([dob, gender]): return render_template('signup.html', error='Patient details (Date of Birth, Gender) are required.')
                new_user_document['patient_profile'] = {'dateOfBirth': dob, 'gender': gender}
            elif user_type == 'doctor':
                specialization, license = request.form.get('specialization'), request.form.get('licenseNumber')
                if not all([specialization, license]): return render_template('signup.html', error='Doctor details (Specialization, License Number)
                new_user_document['doctor_profile'] = {'specialization': specialization, 'licenseNumber': license, 'experience': 0, 'achievements':
            else: return render_template('signup.html', error='Invalid user type selected.')

            users_table.put_item(Item=new_user_document)
            flash(f'Registration successful as {user_type.capitalize()}! Please login.', 'success')
            return redirect(url_for('login'))
        except ClientError as e:
            print(f"DynamoDB Error during signup: {e}")
            return render_template('signup.html', error='Database error during signup.')
    return render_template('signup.html', error=request.args.get('error'))
```

- **Logout Route:** The logout functionality allows users to securely end their session, clearing any session data and redirecting them to the login page.

```python
# --- Logout Route ---
@app.route('/logout')
def logout():
    session.clear()
    print("User logged out. Session cleared.")
    return redirect(url_for('index'))
```

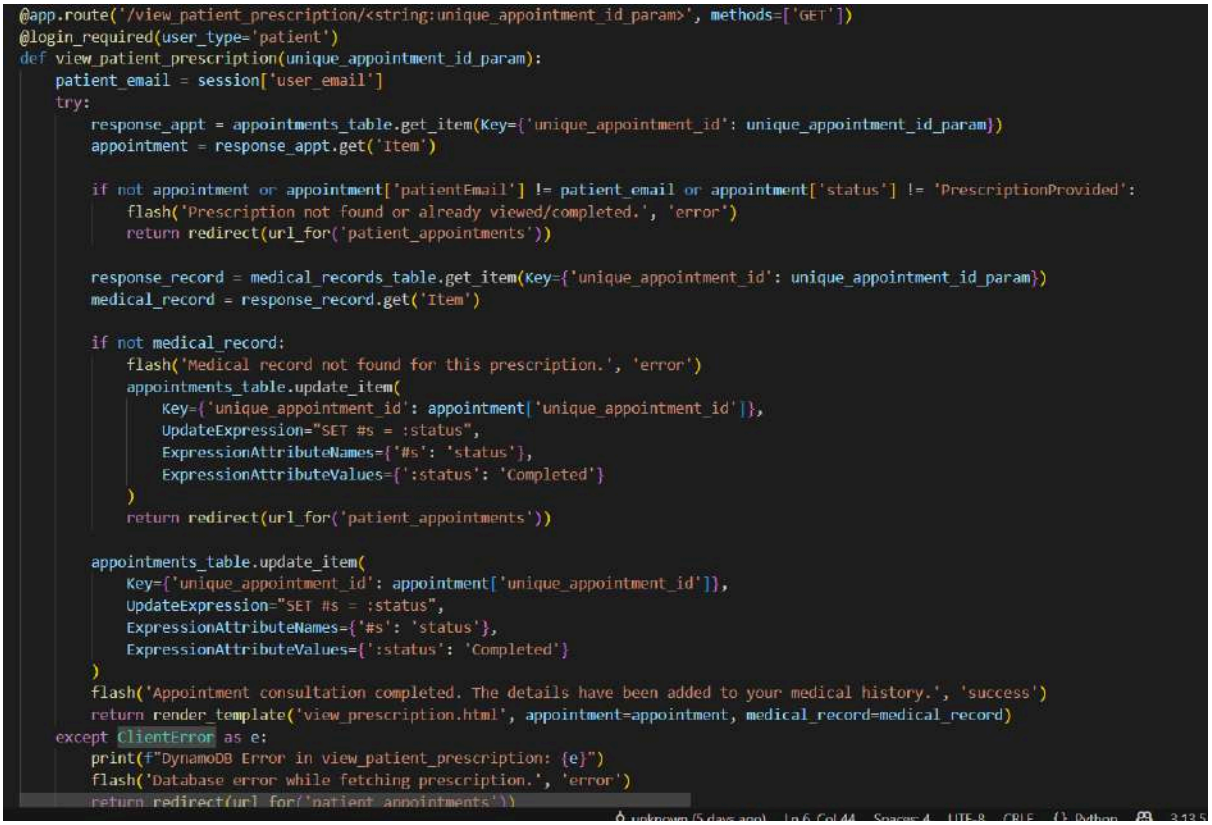
- **Book Appointment Route:** The book appointment route allows users to select a date, time, and doctor for their appointment. Upon submission, the system stores the appointment details in DynamoDB and sends a confirmation notification via SNS.



```python
@app.route('/patient_appointments')
@login_required(user_type='patient')
def patient_appointments():
    user_email = session['user_email']
    pending, upcoming, prescription_ready, past = [], [], [], []
    current_date_time = datetime.now()

    try:
        # DynamoDB Scan with FilterExpression (less efficient for large tables, consider GSI for patientEmail)
        response = appointments_table.scan(
            FilterExpression=boto3.dynamodb.conditions.Attr('patientEmail').eq(user_email)
        )
        appointments = response.get('Items', [])

        for appt in appointments:
            try:
                appt_dt = datetime.strptime(f"{appt['date']} {appt['time']}", '%Y-%m-%d %H:%M')
                if appt['status'] == 'Pending': pending.append(appt)
                elif appt['status'] == 'Accepted':
                    if appt_dt <= current_date_time:
                        appointments_table.update_item(
                            Key={'unique_appointment_id': appt['unique_appointment_id']},
                            UpdateExpression="SET #s = :status",
                            ExpressionAttributeNames={'#s': 'status'},
                            ExpressionAttributeValues={':status': 'Completed'}
                        )
                        appt['status'] = 'Completed'
                    (past if appt['status'] == 'Completed' else upcoming).append(appt)
                elif appt['status'] == 'PrescriptionProvided': prescription_ready.append(appt)
                elif appt['status'] in ['Completed', 'Rejected']: past.append(appt)
            except ValueError:
                print(f"Error parsing date/time for appointment {appt.get('unique_appointment_id')}. Placing in general active list.")
                (upcoming if appt.get('status') in ['Pending', 'Accepted', 'PrescriptionProvided'] else past).append(appt)
```

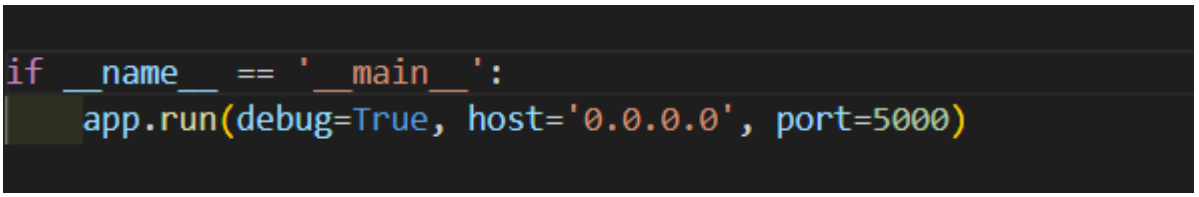
```python
        # Sort lists after categorization
        pending.sort(key=lambda x: (x['date'], x['time']))
        upcoming.sort(key=lambda x: (x['date'], x['time']))
        prescription_ready.sort(key=lambda x: (x['date'], x['time']))
        past.sort(key=lambda x: (x['date'], x['time']), reverse=True) # Past usually sorted descending

    except ClientError as e:
        print(f"DynamoDB Error in patient_appointments: {e}")
        flash('Could not load appointments. Database error.', 'error')
    except Exception as e:
        print(f"Error in patient_appointments: {e}")
        flash('An unexpected error occurred.', 'error')

    _update_session_user_data(user_email)
    return render_template('patient_appointments.html',
                           pending_appointments=pending, upcoming_appointments=upcoming,
                           prescription_ready_appointments=prescription_ready, past_appointments=past)
```

- **Issue Prescription Route:** This route enables doctors to issue new prescriptions to patients. It stores the prescription details in DynamoDB and sends an SNS notification to inform the patient.



```python
@app.route('/view_patient_prescription/<string:unique_appointment_id_param>', methods=['GET'])
@login_required(user_type='patient')
def view_patient_prescription(unique_appointment_id_param):
    patient_email = session['user_email']
    try:
        response_appt = appointments_table.get_item(Key={'unique_appointment_id': unique_appointment_id_param})
        appointment = response_appt.get('Item')

        if not appointment or appointment['patientEmail'] != patient_email or appointment['status'] != 'PrescriptionProvided':
            flash('Prescription not found or already viewed/completed.', 'error')
            return redirect(url_for('patient_appointments'))

        response_record = medical_records_table.get_item(Key={'unique_appointment_id': unique_appointment_id_param})
        medical_record = response_record.get('Item')

        if not medical_record:
            flash('Medical record not found for this prescription.', 'error')
            appointments_table.update_item(
                Key={'unique_appointment_id': appointment['unique_appointment_id']},
                UpdateExpression="SET #s = :status",
                ExpressionAttributeNames={'#s': 'status'},
                ExpressionAttributeValues={':status': 'Completed'}
            )
            return redirect(url_for('patient_appointments'))

        appointments_table.update_item(
            Key={'unique_appointment_id': appointment['unique_appointment_id']},
            UpdateExpression="SET #s = :status",
            ExpressionAttributeNames={'#s': 'status'},
            ExpressionAttributeValues={':status': 'Completed'}
        )
        flash('Appointment consultation completed. The details have been added to your medical history.', 'success')
        return render_template('view_prescription.html', appointment=appointment, medical_record=medical_record)
    except ClientError as e:
        print(f"DynamoDB Error in view_patient_prescription: {e}")
        flash('Database error while fetching prescription.', 'error')
        return redirect(url_for('patient_appointments'))
```

- **Deployment Code:**

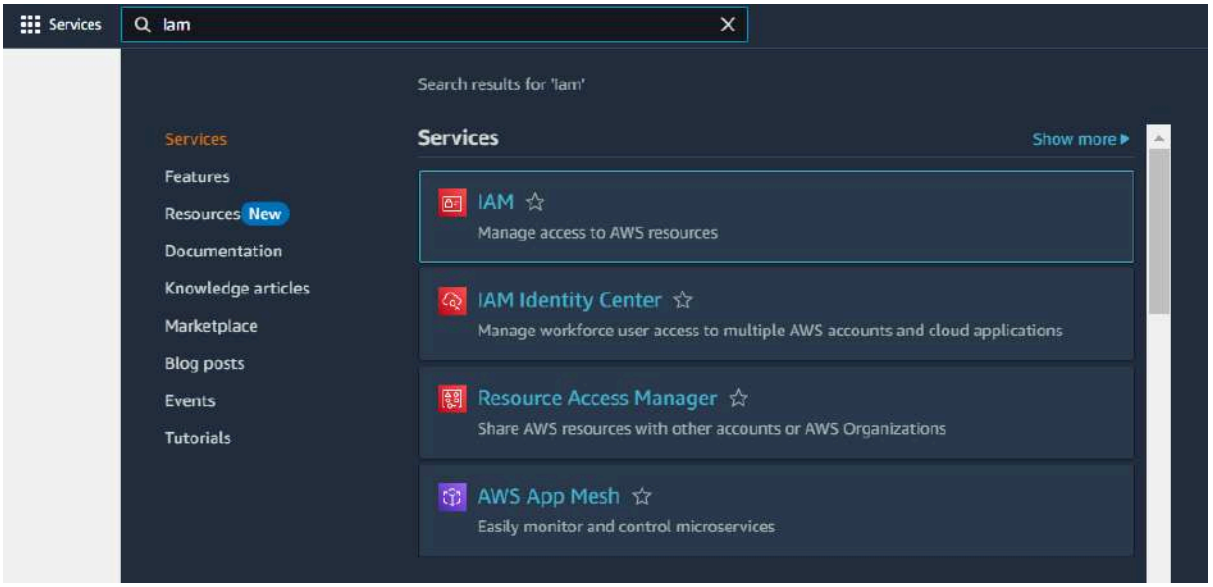


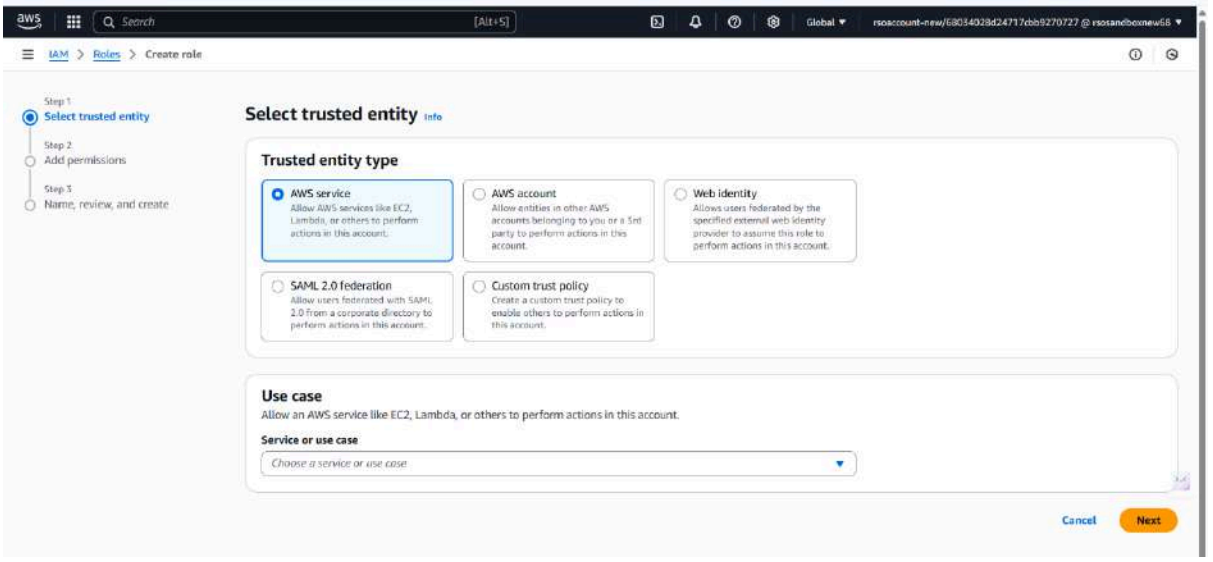
```python
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

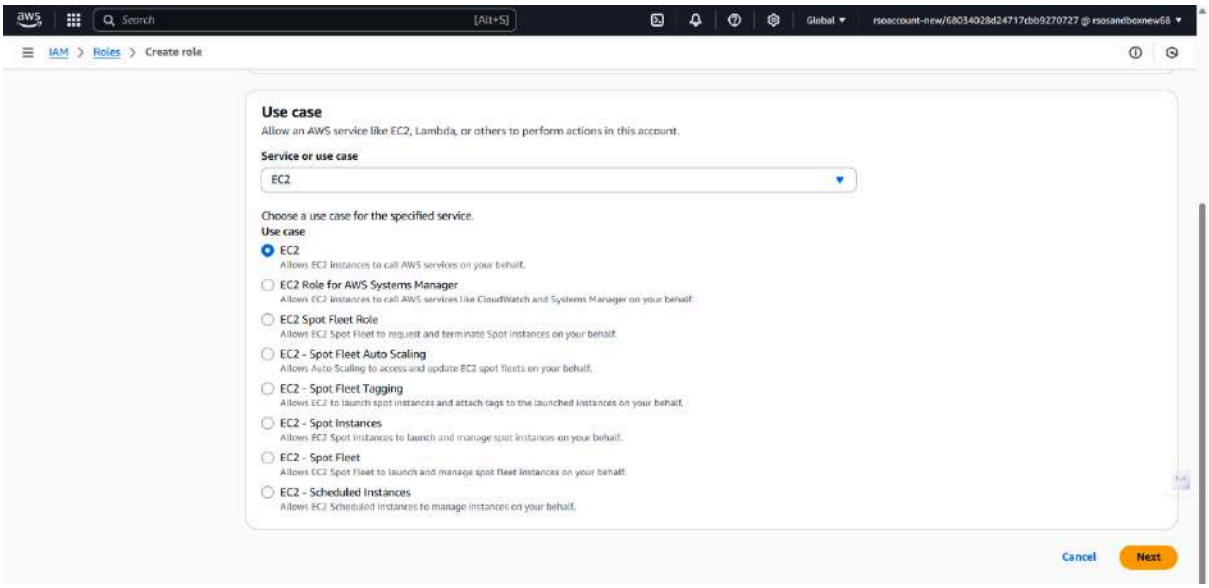
Milestone 5. IAM Role Setup

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.


Services
Q Iam
Search results for 'iam'
Services
Features
Resources New
Documentation
Knowledge articles
Marketplace
Blog posts
Events
Tutorials
Services
Show more
IAM
Manage access to AWS resources
IAM Identity Center
Manage workforce user access to multiple AWS accounts and cloud applications
Resource Access Manager
Share AWS resources with other accounts or AWS Organizations
AWS App Mesh
Easily monitor and control microservices

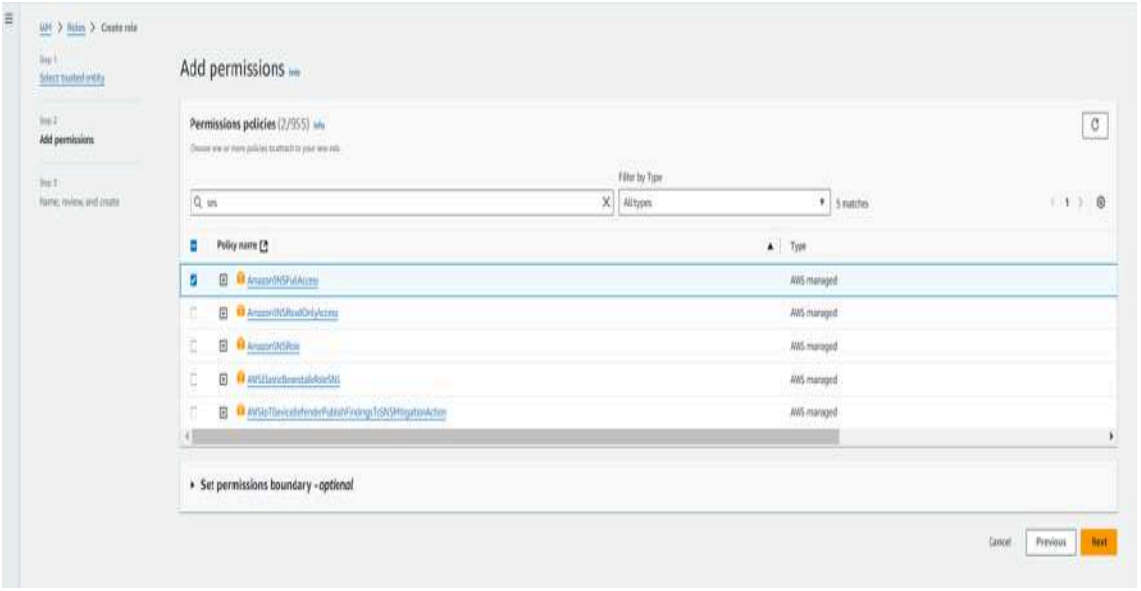

IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Select trusted entity Info
Trusted entity type
AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.
AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.
Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.
Service or use case
Choose a service or use case
Cancel
Next

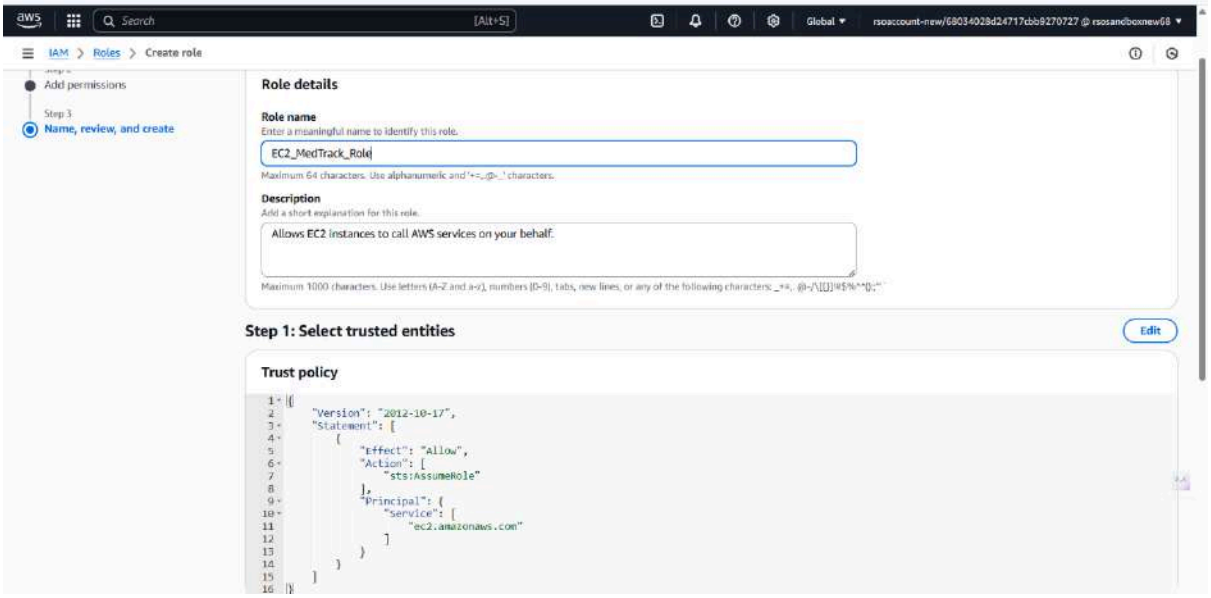

IAM > Roles > Create role
Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.
Service or use case
EC2
Choose a use case for the specified service.
Use case
EC2
Allows EC2 instances to call AWS services on your behalf.
EC2 Role for AWS Systems Manager
Allows EC2 instances to call AWS services like CloudWatch and Systems Manager on your behalf.
EC2 Spot Fleet Role
Allows EC2 Spot Fleet to request and terminate Spot Instances on your behalf.
EC2 - Spot Fleet Auto Scaling
Allows Auto Scaling to access and update EC2 spot fleets on your behalf.
EC2 - Spot Fleet Tagging
Allows EC2 to launch spot instances and attach tags to the launched instances on your behalf.
EC2 - Spot Instances
Allows EC2 Spot Instances to launch and manage spot instances on your behalf.
EC2 - Spot Fleet
Allows EC2 Spot Fleet to launch and manage spot fleet instances on your behalf.
EC2 - Scheduled Instances
Allows EC2 Scheduled Instances to manage instances on your behalf.
Cancel
Next

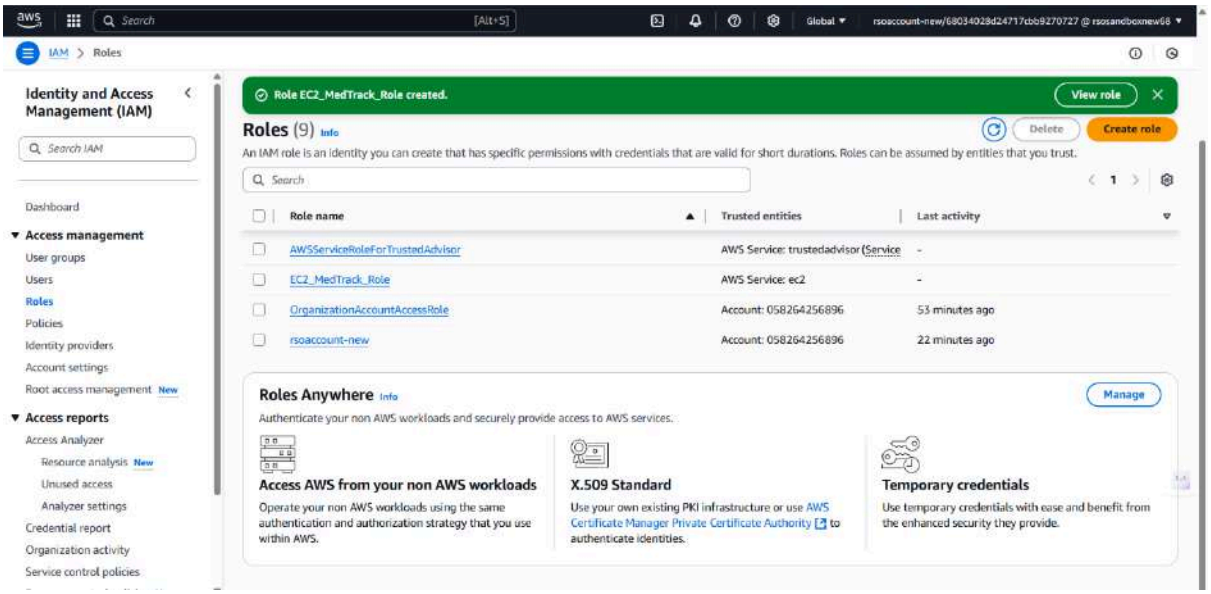
- **Activity 5.2: Attach Policies.**

  Attach the following policies to the role:

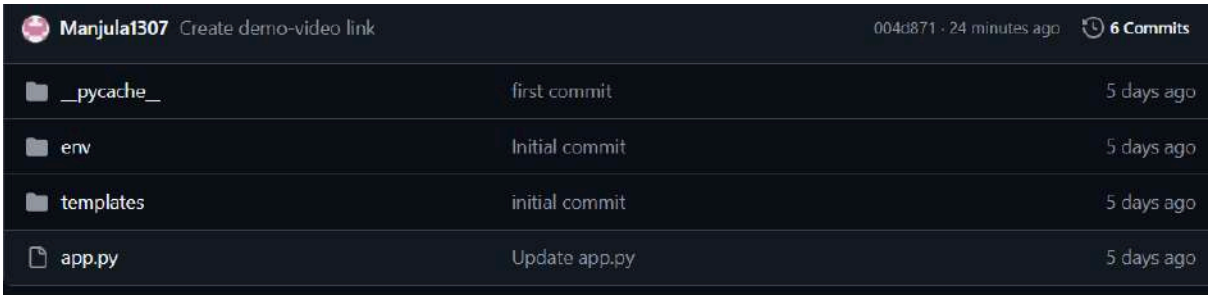
  - AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.
  - AmazonSNSFullAccess: Grants EC2 the ability to send notifications via SNS.

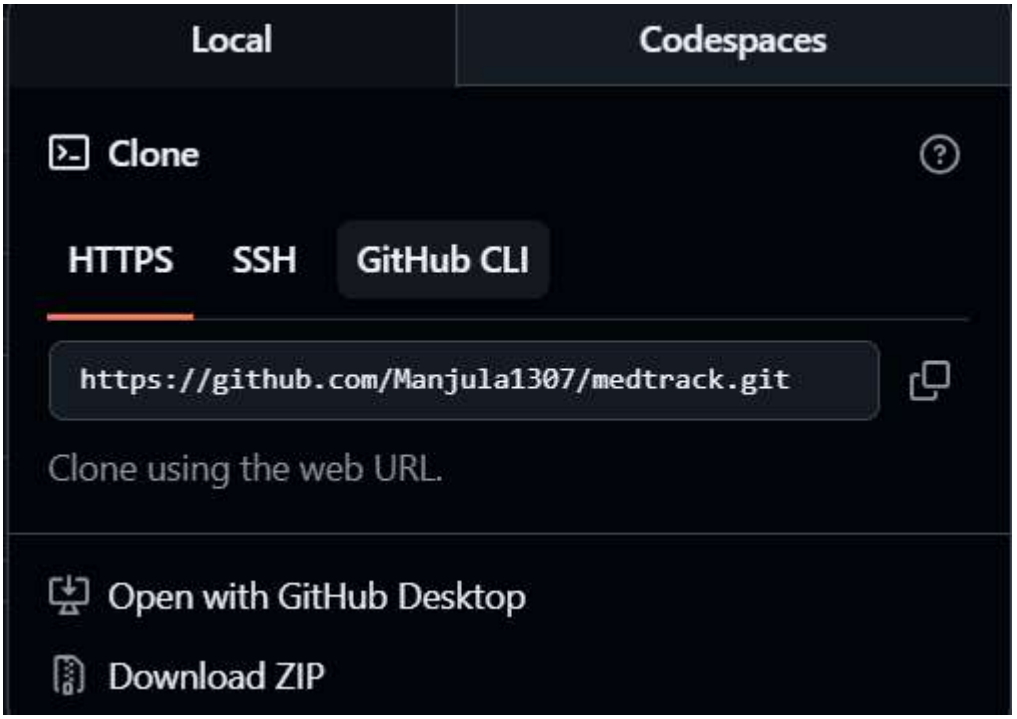






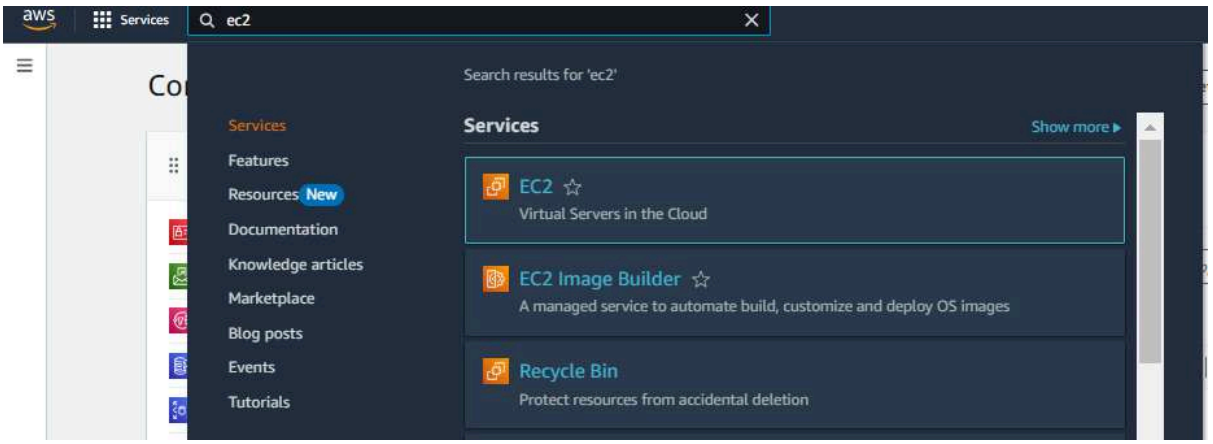
- **Milestone 6. EC2 Instance Setup**
  - **Note:** Load your Flask app and Html files into GitHub repository.



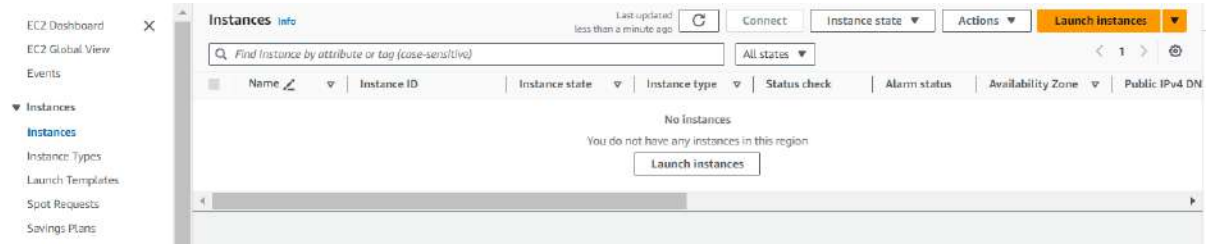


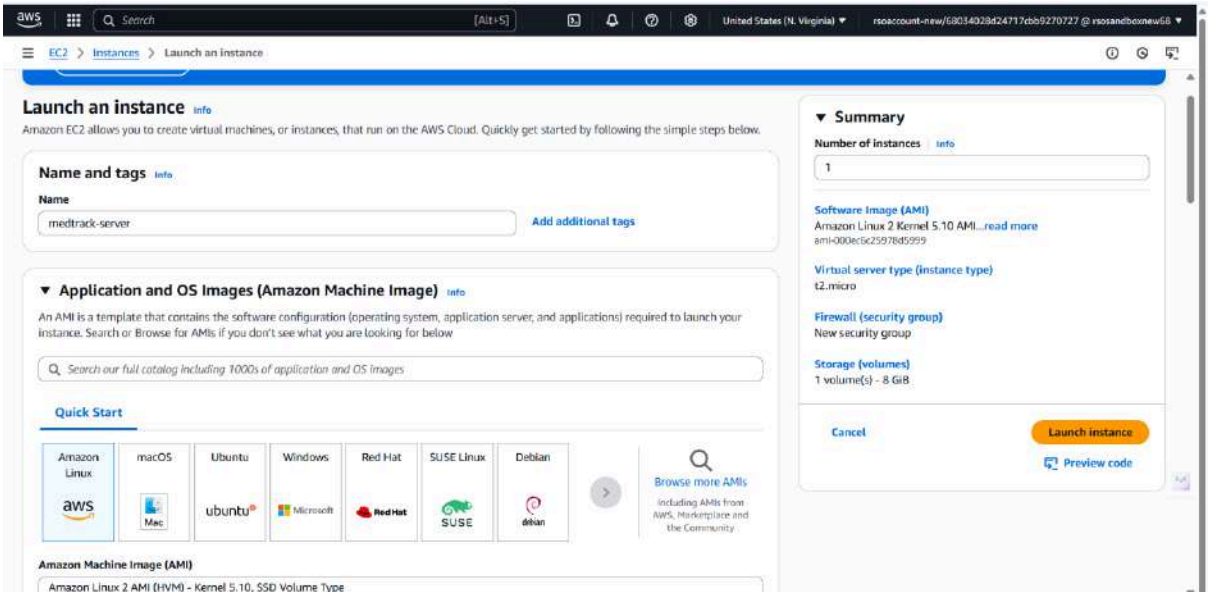
Activity 6.1: **Launch an EC2 instance to host the Flask application.**

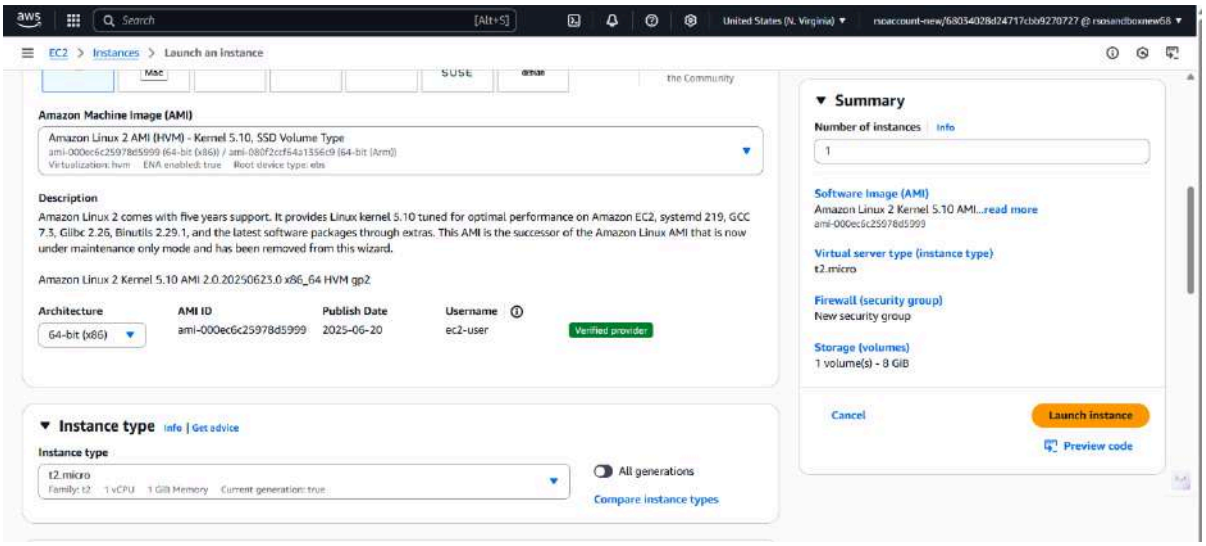
- **Launch EC2 Instance**
  - In the AWS Console, navigate to EC2 and launch a new instance.



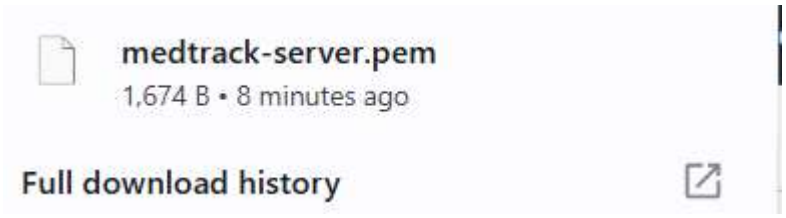
- Click on Launch instance to launch EC2 instance







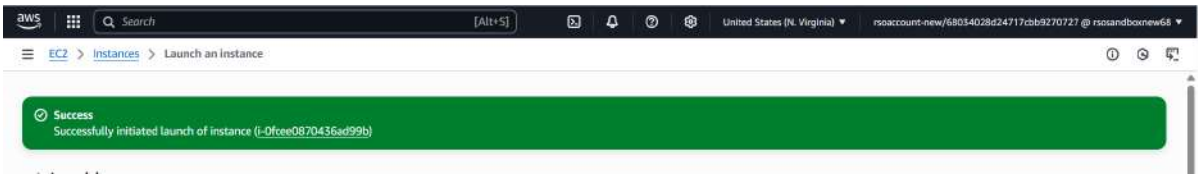
- Create and download the key pair for server access:



- **Activity 6.2: Configure security groups for HTTP, and SSH access.**

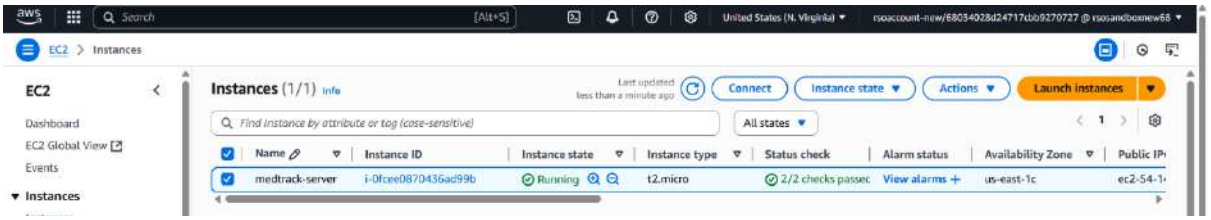
For network settings during EC2 instance launch:

1. In the **Network Settings** section, select the **VPC** and **Subnet** you wish to use (if unsure, the default VPC and subnet should work).
2. Ensure **Auto-assign Public IP** is enabled so your instance can be accessed from the internet.
3. In **Security Group**, either select an existing one or create a new one that allows SSH (port 22) access to your EC2 instance for remote login.



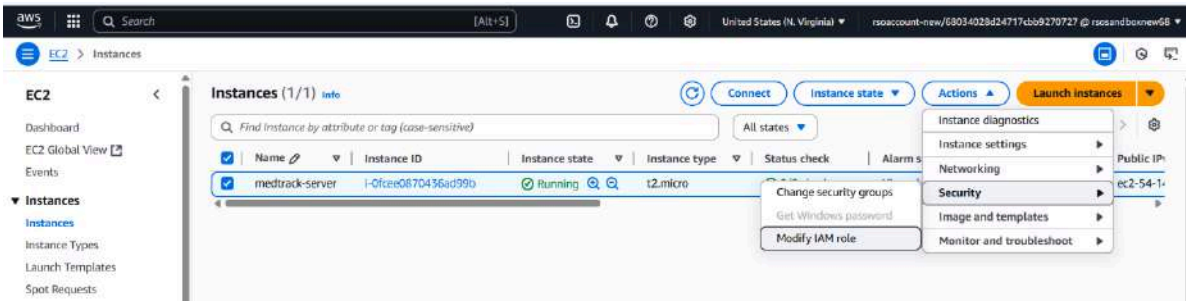
- To connect to EC2 using **EC2 Instance Connect,** start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to Security and selecting **Modify IAM Role** to attach the appropriate role. After the IAM 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

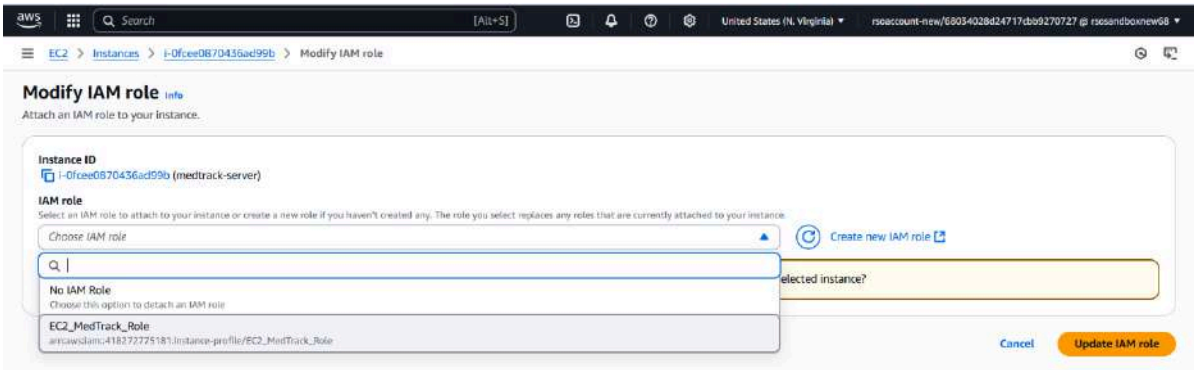
presented, choose **EC2 Instance Connect**. Finally, click Connect again, and a new browser-based terminal will open, allowing you to access your EC2 instance directly from your browser.



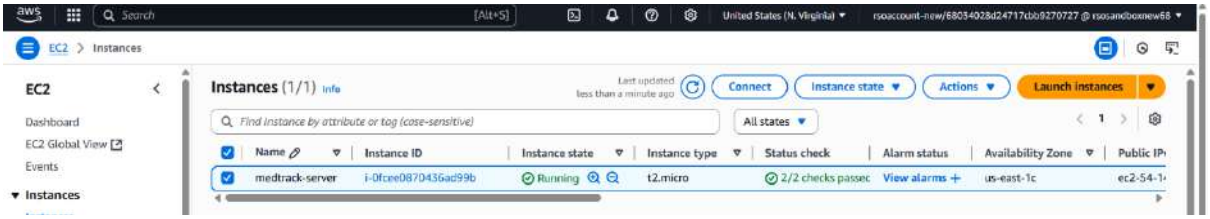
To modify the **IAM role** for your EC2 instance:

1. Go to the **AWS IAM Console**, select **Roles**.
2. Click **Attach Policies**, then choose the required policies (e.g., **DynamoDBFullAccess**, **SNSFullAccess**) and click **Attach Policy**.
3. If needed, update the instance to use this modified role by selecting the EC2 instance, clicking **Actions**, then **Security**, and **Modify IAM role** to select the updated role.

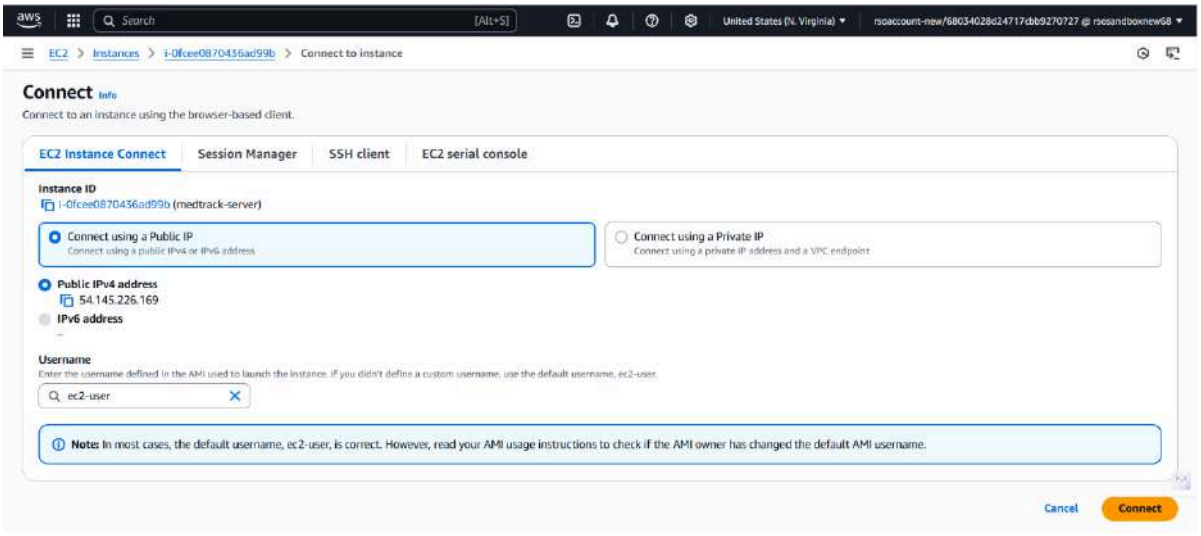


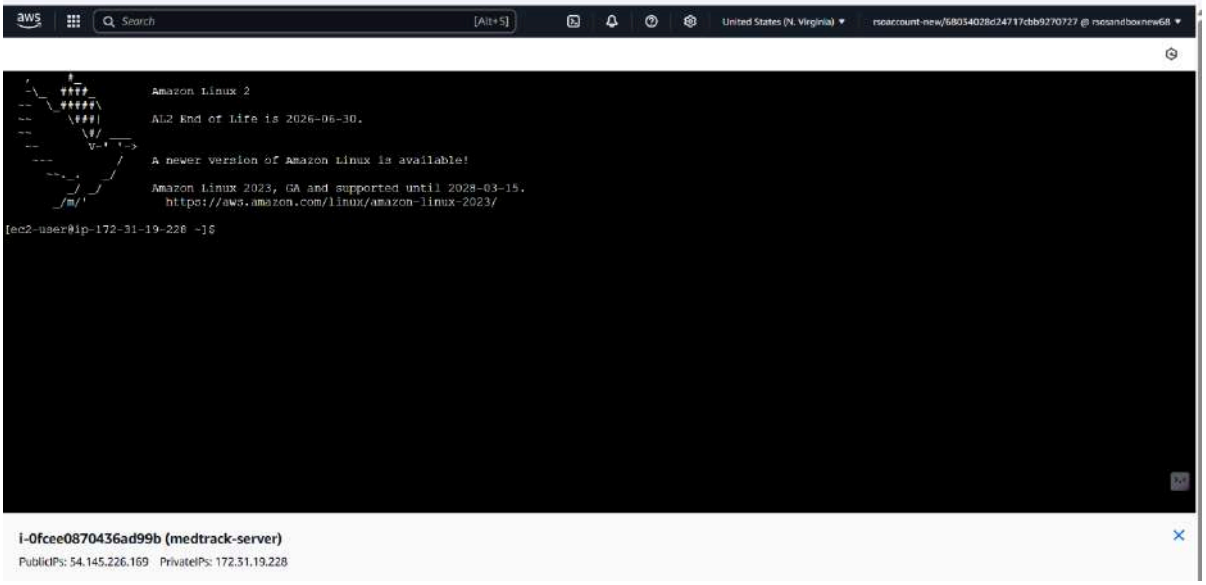


- Connect to your EC2 instance:
  1. Go to the **EC2 Dashboard**, select your running instance, and click **Connect**.



- Now connect the EC2 with the files





Milestone 7: Deployment on EC2

Deployment on an EC2 instance involves launching a server, configuring security groups for public access, and uploading your application files. After setting up necessary dependencies and environment variables, start your application and ensure it's running on the correct port. Finally, bind your domain or use the public IP to make the application accessible online.

Activity 7.1: 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 7.2:Clone Your Flask Project from GitHub

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

Run: 'git clone https://github.com/your-github-username/your-repository-name.git'

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

here: 'git clone https://github.com/Ravi-teja-777/medtrack_app.git

- This will download your project to the EC2 instance.

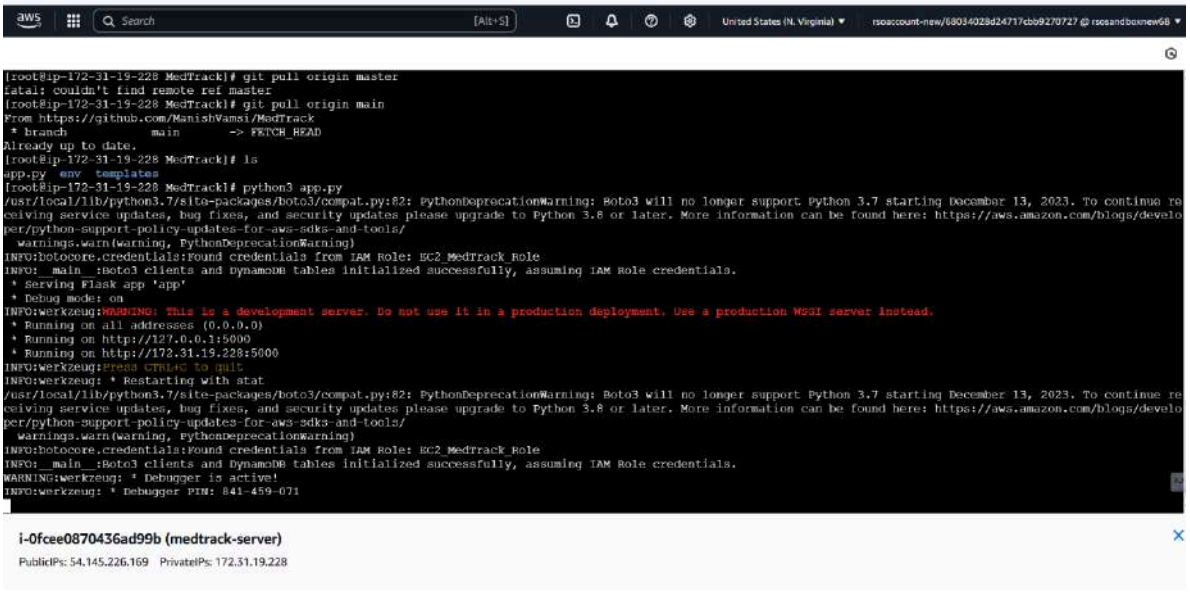
To navigate to the project directory, run the following command:

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

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:

Public-IPs: https://54.145.226.169:5000/

Milestone 8: Testing and Deployment

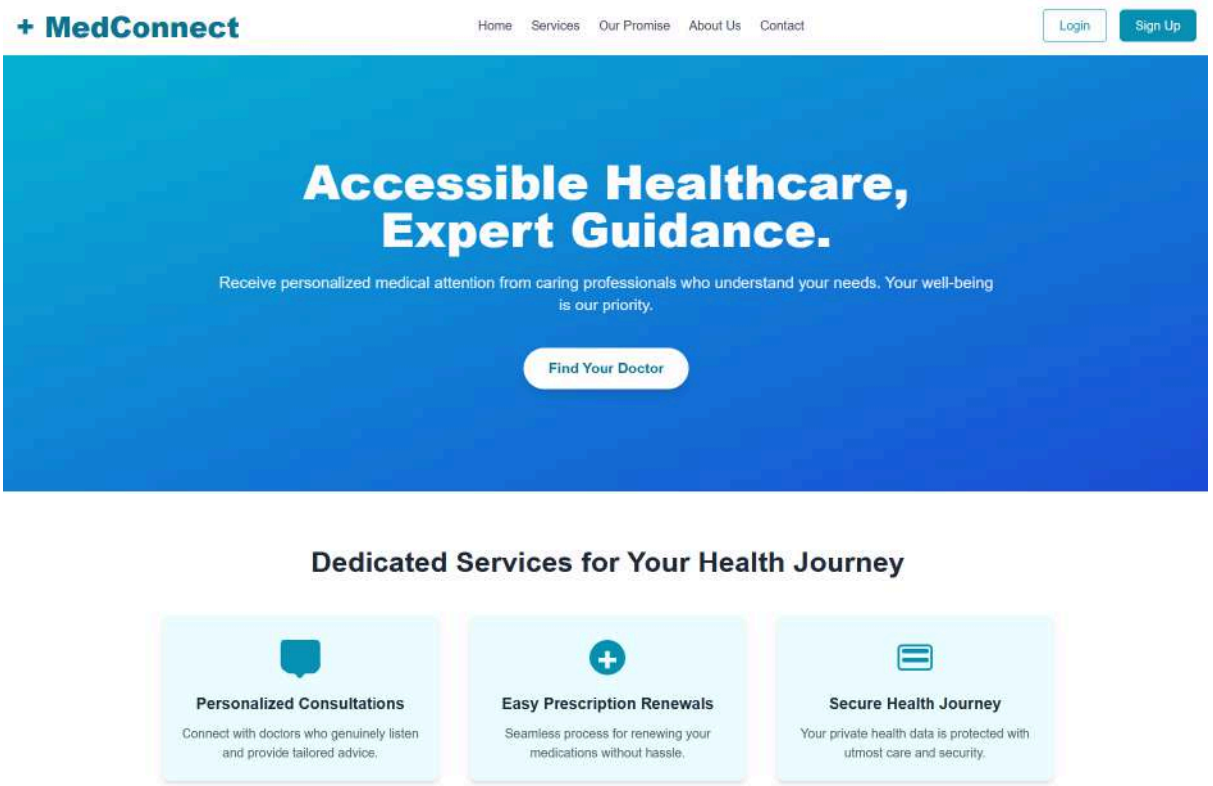
- **Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.**

Functional testing to verify the project

Home Page:
The Home Page of your project is the main entry point for users, where they can interact with the system. It typically includes:

1. Input Fields: For users to enter basic information like appointment requests, diagnosis submissions, or service bookings.
2. Navigation: Links to other sections such as the login page, dashboard, or service options.
3. Responsive Design: Ensures the page is accessible across devices with a clean, user-friendly interface.

The Home Page serves as the initial interface that directs users to the key functionalities of your web application.



DOCTOR AND PATIENT REGISTRATION PAGE:

The Doctor Registration Page allows doctors to register and create an account on the platform. It typically includes:

1. Input Fields: For doctor details such as name, specialty, qualifications, and contact information.
2. Login Credentials: Fields for setting a username and password for secure access.
3. Submit Button: A button to submit the registration details, which will then be stored in the database after validation.

PATIENT AND DOCTOR LOGIN PAGES:

The Patient and Doctor Login Pages allow users to securely access their accounts on the platform. Each login page typically includes:

1. Username and Password Fields: Users enter their credentials (username and password) to authenticate their account.
2. Login Button: A button to submit login details and validate user access.

Once logged in, patients and doctors are redirected to their respective dashboards to manage appointments, medical records, and other relevant tasks.

Full Name

Email Address

Password

Confirm Password

I am a:

(•) Patient ( ) Doctor

Patient Details

Date of Birth

dd-mm-yyyy

Gender

Select Gender

Sign Up

Full Name

Email Address

Password

Confirm Password

I am a:

( ) Patient (•) Doctor

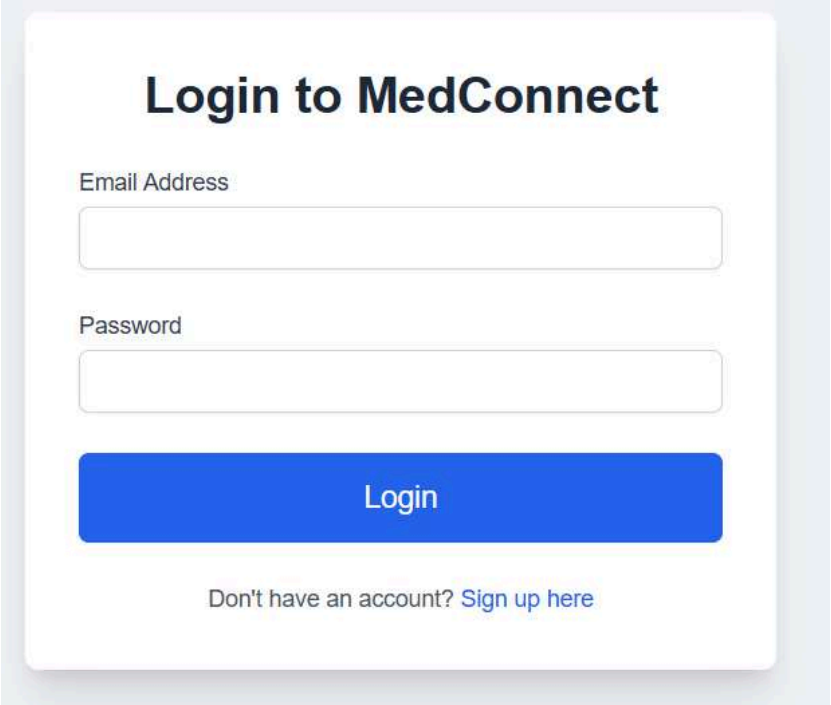
Doctor Details

Specialization

e.g., Cardiology, Pediatrics

Medical License Number

Sign Up

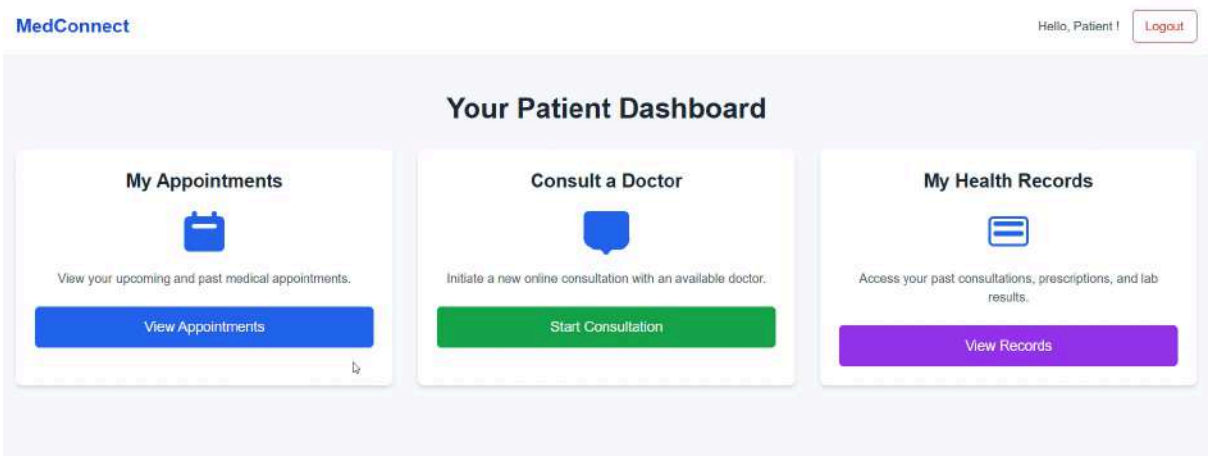


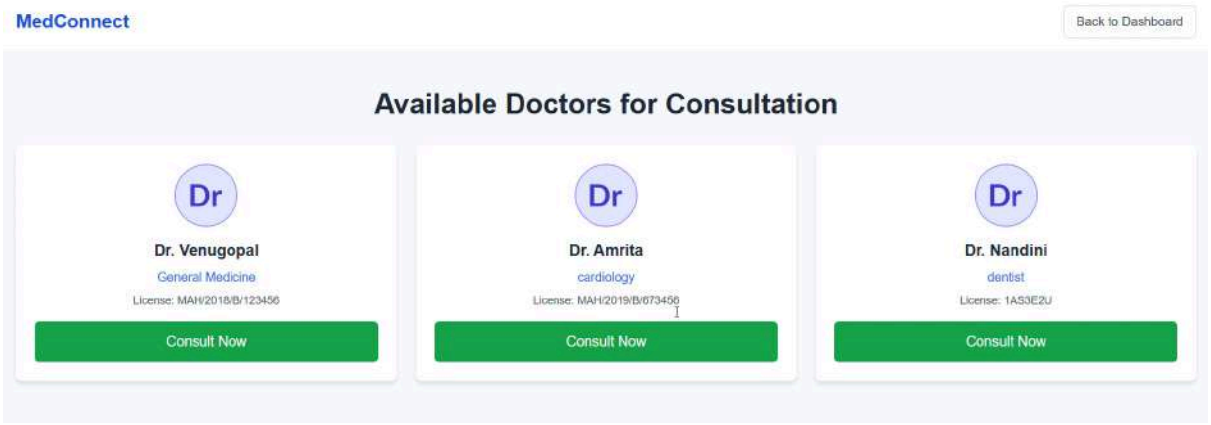
User Dashboard:

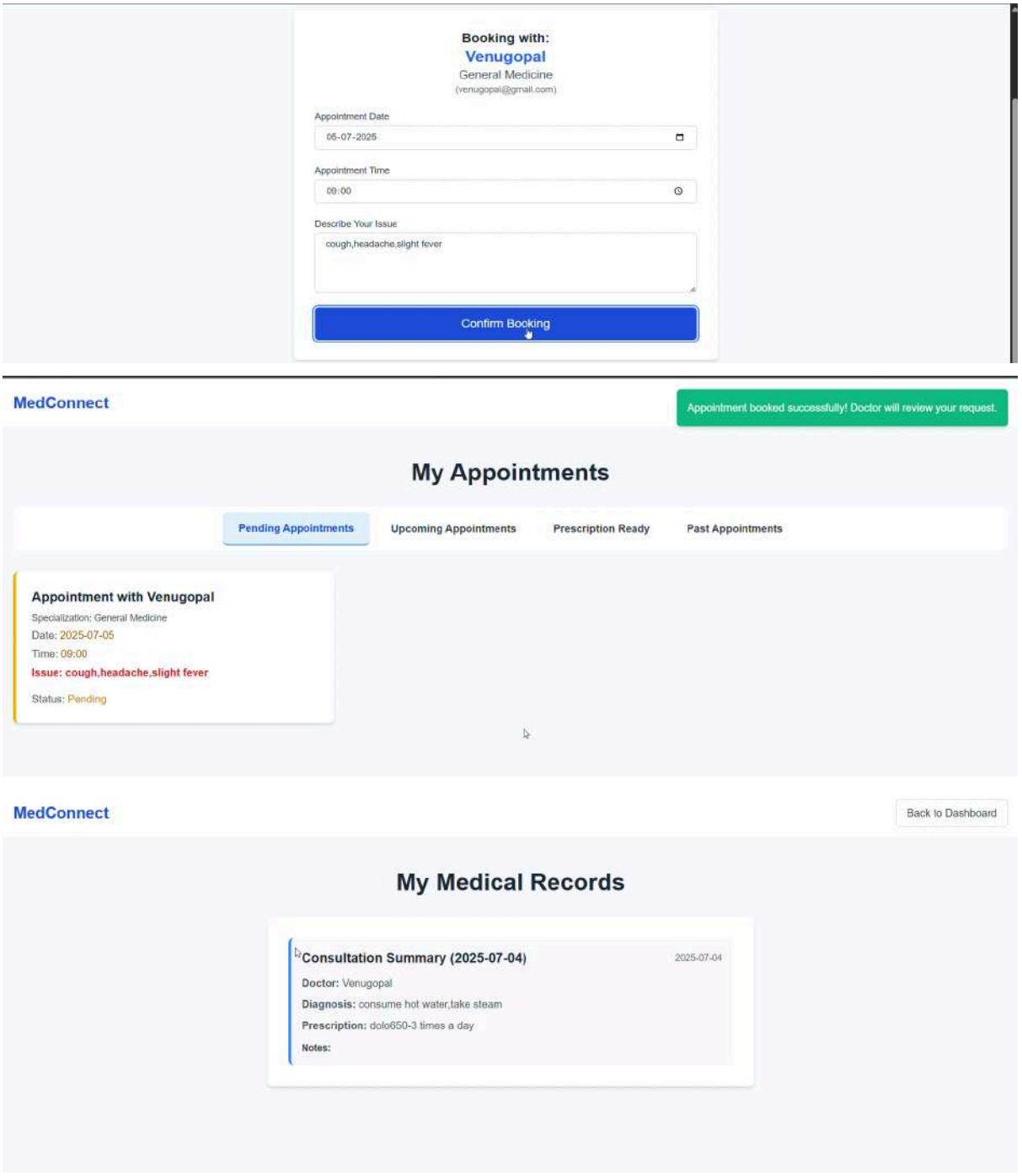
The User Dashboard (for patients) provides an easy interface to manage appointments and track their status. It typically includes:

1. Book Appointment Section: A form for selecting a doctor, choosing an appointment time, and submitting the request.
2. Appointment Status: A section showing the current status of appointments (e.g., confirmed, pending, or completed) with options to view details or cancel.
3. Medication Remainders: Patients can easily set up personalized reminders for their medications, including dosage, frequency, and specific times.

This dashboard helps patients book new appointments and keep track of their healthcare schedules.


MedConnect
Hello, Patient !
Logout
Your Patient Dashboard
My Appointments
View your upcoming and past medical appointments.
View Appointments
Consult a Doctor
Initiate a new online consultation with an available doctor.
Start Consultation
My Health Records
Access your past consultations, prescriptions, and lab results.
View Records


MedConnect
Back to Dashboard
Available Doctors for Consultation
Dr
Dr. Venugopal
General Medicine
License: MAH/2018/B/123456
Consult Now
Dr
Dr. Amrita
cardiology
License: MAH/2019/B/673456
Consult Now
Dr
Dr. Nandini
dentist
License: 1AS3E2U
Consult Now



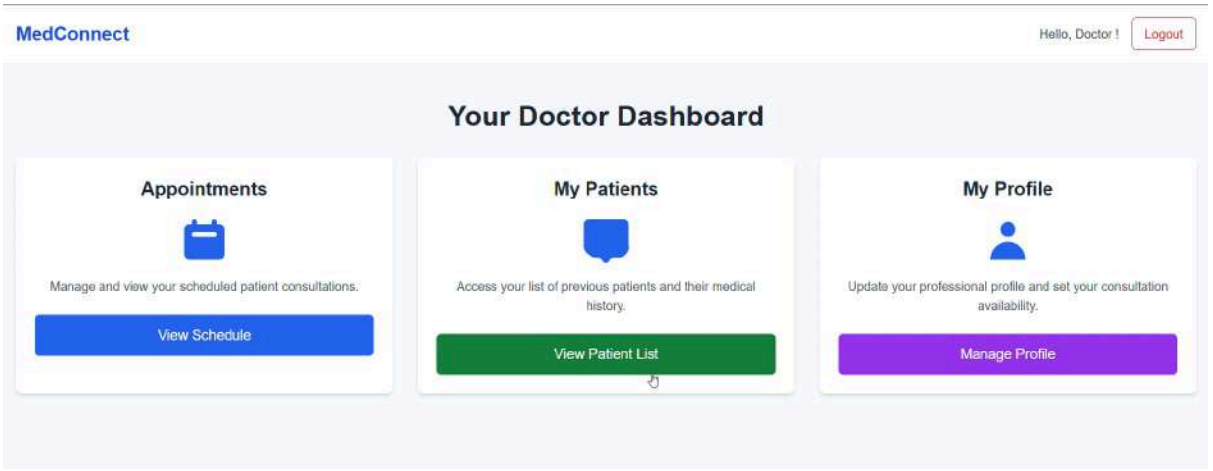
Doctor Dashboard:

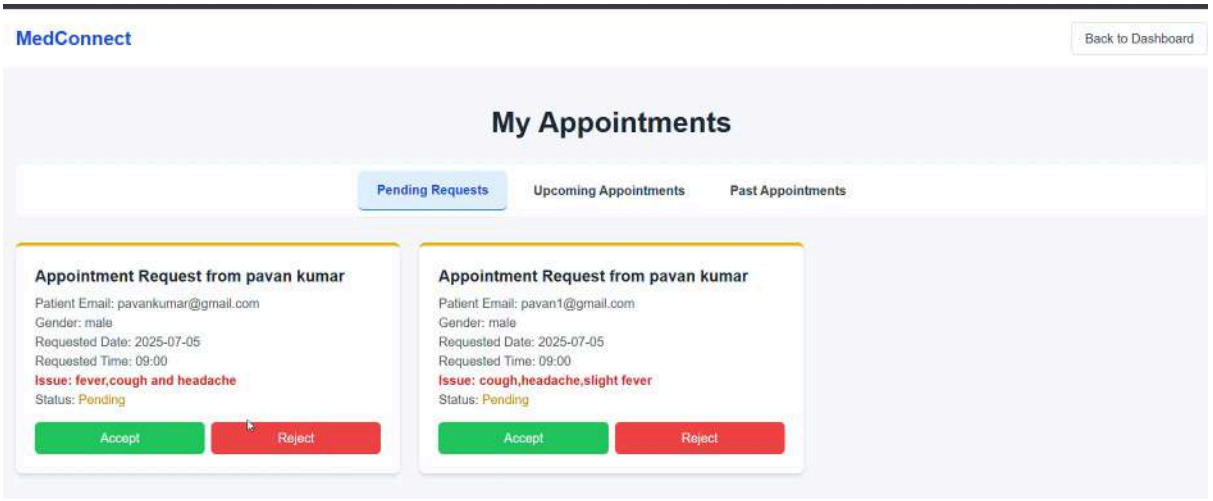
The **Doctor Dashboard** provides doctors with a comprehensive view of their upcoming appointments and patient details. It typically includes:

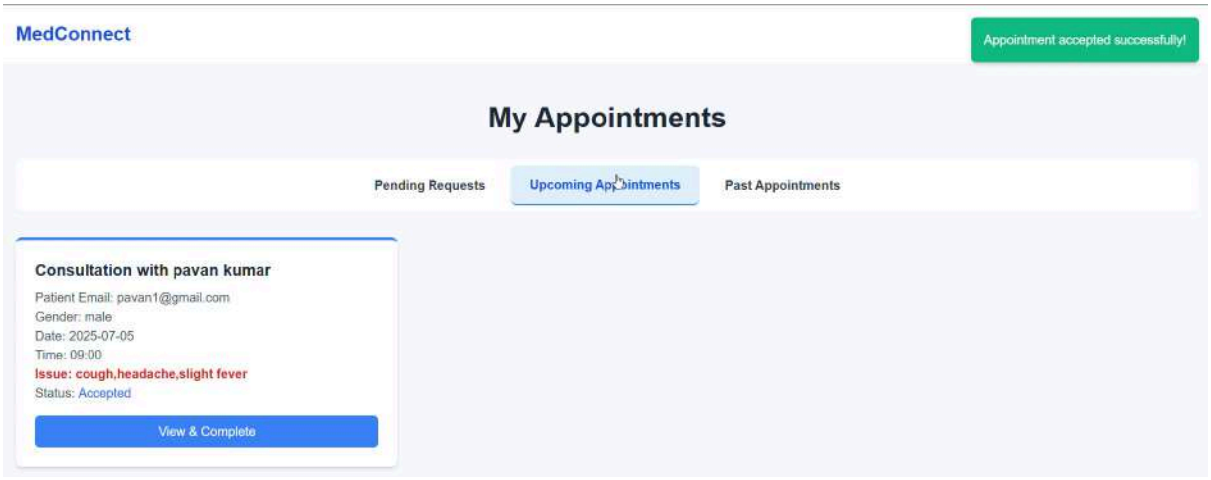
1. **Upcoming Appointments List**: A table or list showing patient names, appointment times, and appointment statuses (e.g., confirmed, pending).
2. **Patient Details**: Quick access to each patient's medical history, contact information, and previous visit records.

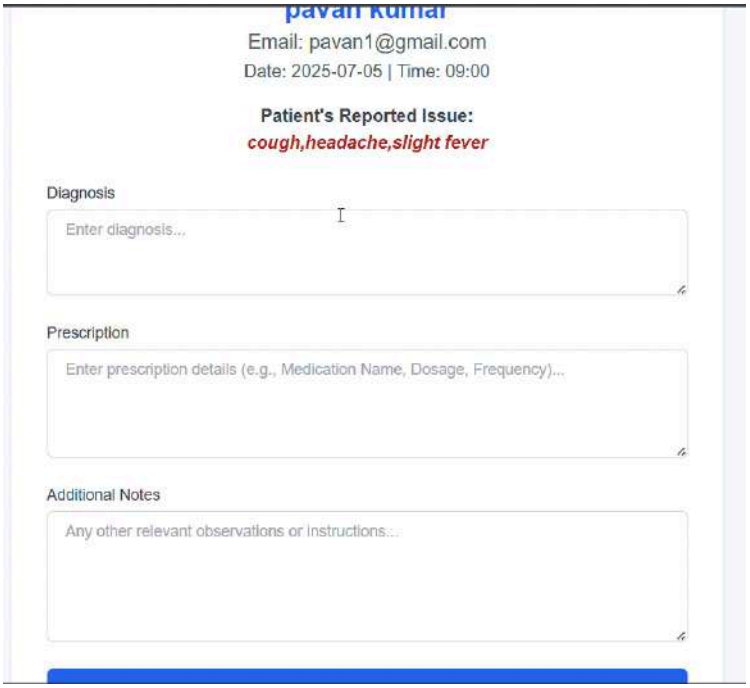
3. **Appointment Actions**: Options to view, confirm , or cancel appointments, ensuring efficient management.

The dashboard serves as the main interface for doctors to manage their schedules, track patient interactions, and provide timely care.









DynamoDB Database:

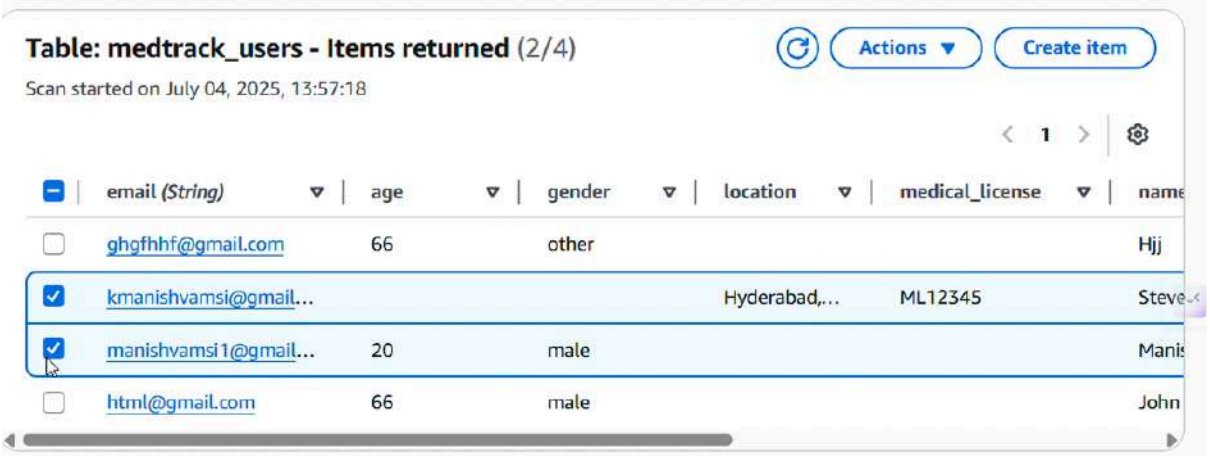
1. **medtrack_users:**
   This table stores all user profiles for the Medtrack application, including patient and doctor accounts. It contains essential login credentials, personal details, and user-specific attributes like specialization or age.

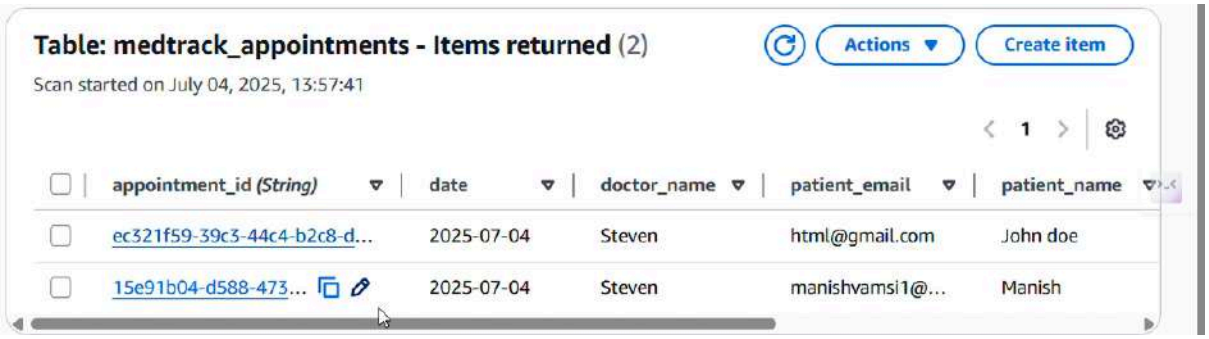
2. **medtrack_appointments:**
   This table is dedicated to managing all scheduled medical appointments within the system. It records details such as the patient, doctor, date, time, reason for visit, and the current status of each appointment.

3. **medtrack_prescriptions:**

This table holds all digital prescriptions issued by doctors through the application. It details the prescribed medication, dosage, instructions, the prescribing doctor, and the patient it was issued for.

4. **medtrack_medication_reminders:**
   This table stores personalized medication reminders set by patients to help them adhere to their treatment plans. It includes the medication name, dosage, frequency, times, and tracks whether a dose has been taken





Conclusion:

The **MedTrack application** has been successfully developed and deployed using a robust cloud-based architecture tailored for modern healthcare environments. Leveraging AWS services such as EC2 for hosting, DynamoDB for secure and scalable patient data management, and SNS for real-time alerts, the platform ensures reliable and efficient access to essential medical tracking services. This system addresses critical challenges in healthcare such as managing patient records, monitoring medication schedules, and ensuring timely communication between healthcare providers and patients.

The cloud-native approach enables seamless scalability, allowing MedTrack to support increasing numbers of users and data without compromising performance or reliability. The integration of Flask with AWS ensures smooth backend operations, including patient registration, medication reminders, and health updates. Thorough testing has validated that all features—from user onboarding to alert notifications—function reliably and securely.

In conclusion, the MedTrack application delivers a smart, efficient solution for modernizing healthcare management, improving patient care, and streamlining communication between medical staff and patients. This project highlights the transformative power of cloud-based technologies in solving real-world challenges in the healthcare sector.