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.

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 page, 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.

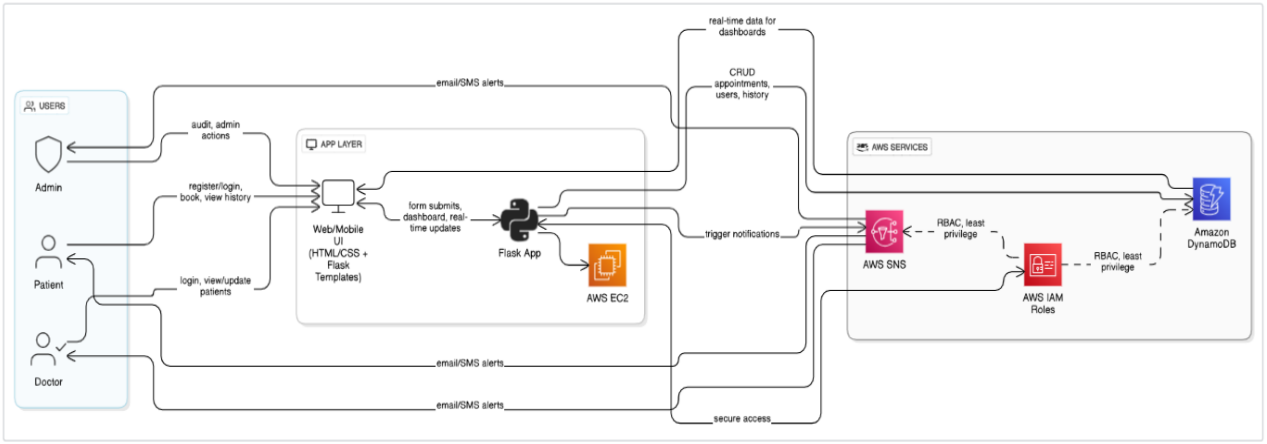
Scenario 2: Secure User Management with IAM

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.

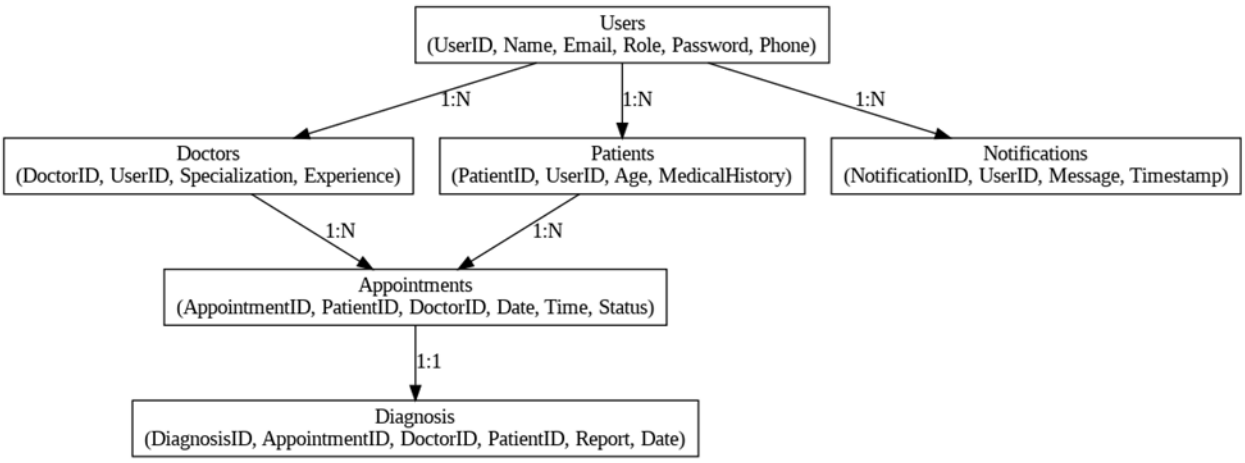
Scenario 3: Easy Access to Medical History and Resources

The MedTrack system provides doctors and patients with easy access to medical histories and relevant resources. For example, a doctor logs in to view a patient's medical history and upcoming appointments. They can quickly access, and update records as needed. 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.

AWS 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. **SNS Overview**: SNS Documentation
6. **Git Version Control**: Git Documentation

Project WorkFlow:

Milestone 1. Web Application Development and Setup

- Develop the Backend Using Flask.
- Integrate AWS Services Using boto3.

Milestone 2. AWS Account Setup and Login

- Set up an AWS account if not already done.
- Login to AWS Management Console.

Milestone 3. DynamoDB Database Creation and Setup

- Create a DynamoDB Table.
- Configure Attributes for User Data and Book Requests.

Milestone 4. SNS Notification Setup

- Create SNS topics for book request notifications.
- Subscribe users and library staff to SNS email notifications.

Milestone 5. IAM Role Setup

- Create IAM Role
- Attach Policies

Milestone 6. EC2 Instance Setup

- Launch an EC2 instance to host the Flask application.
- Configure security groups for HTTP, and SSH access.

Milestone 7. Deployment using EC2

- Upload Flask Files
- Run the Flask App

Milestone 8. Testing and Deployment

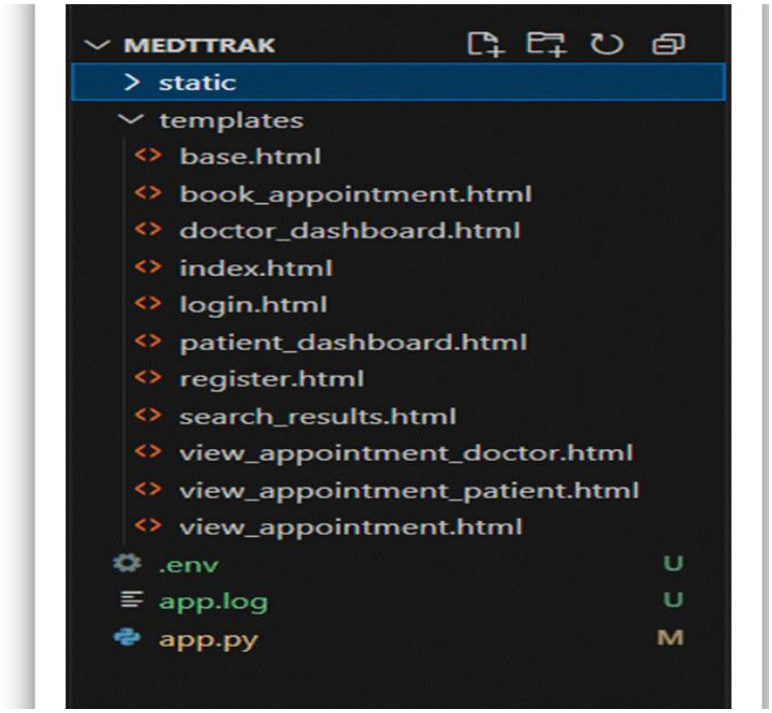
- Conduct functional testing to verify user registration, login, book requests, and notifications

Milestone 1: Web Application Development and Setup

Backend Development and Application Setup focuses on establishing the core structure of the application. This includes configuring the backend framework, setting up routing, and integrating database connectivity. It lays the groundwork for handling user interactions, data management, and secure access

FLASK DEPLOYMENT

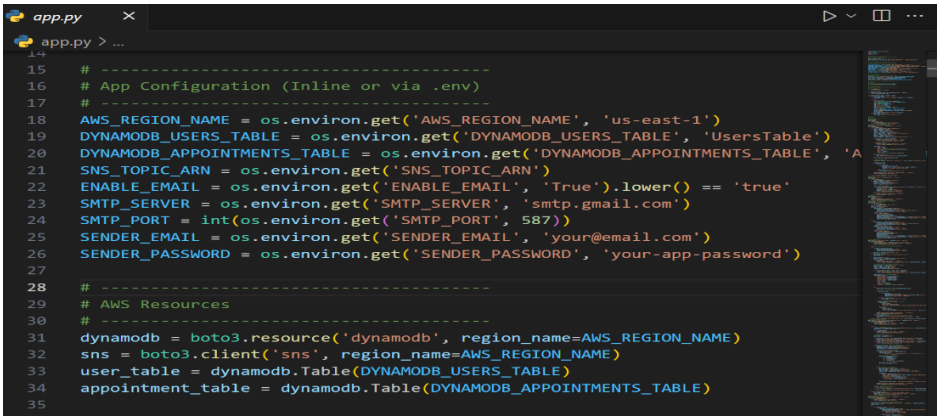
File Explorer Structure



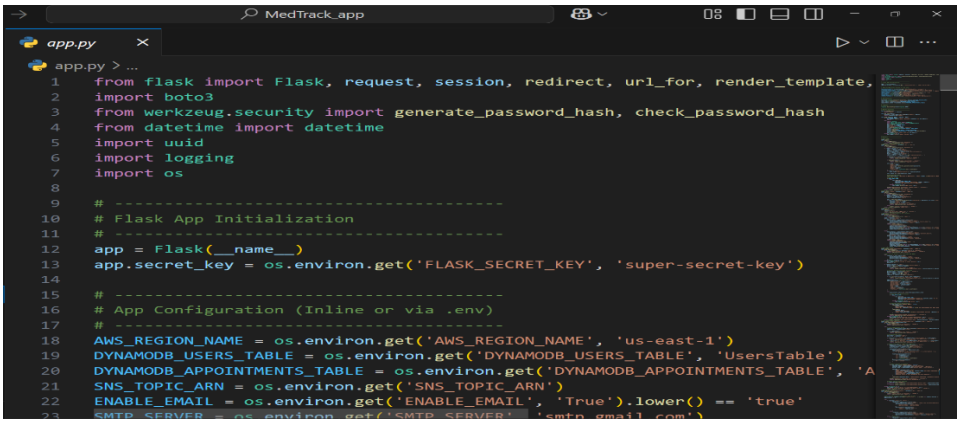
Flask App Initialization:

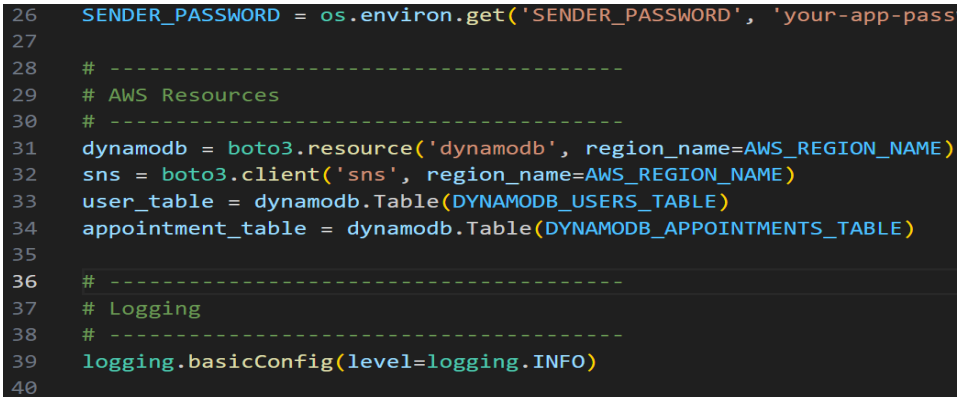
In the MedTrack project, the Flask app is initialized to establish the backend infrastructure, enabling it to handle multiple user interactions such as patient registration, appointment booking, and submission of medical reports. The Flask framework processes incoming requests, communicates with the DynamoDB database for storing user data, and integrates seamlessly with AWS services. Additionally, the routes and APIs are defined to manage different functionalities like secure login, appointment scheduling, and medical history retrieval. This initialization sets up the foundation for smooth, real-time communication between patientsand doctors while ensuring the app is scalable and secure.

Use boto3 to connect to DynamoDB for handling user registration, book requests database operations and also mention region_name where Dynamodb tables are created.



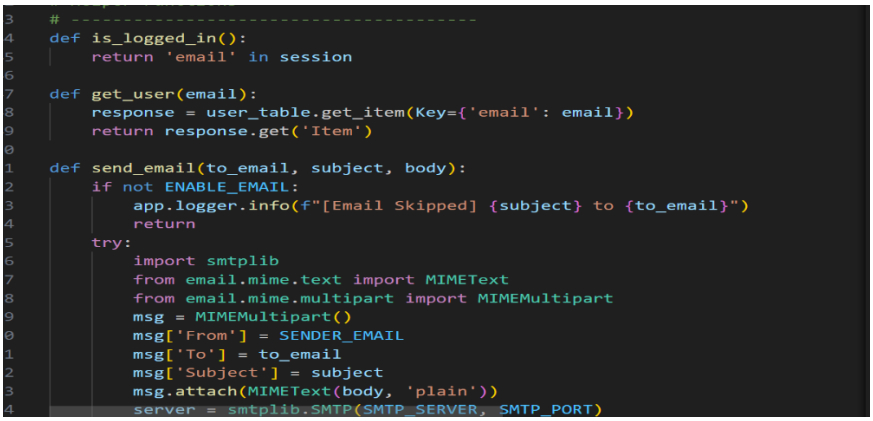
In the MedTrack project, AWS SNS sends real-time notifications to patients and doctors about appointments and updates. DynamoDB stores user data, medical records, and appointments securely, offering fast, scalable access. Both services are integrated with Flask to ensure smooth communication and efficient data management



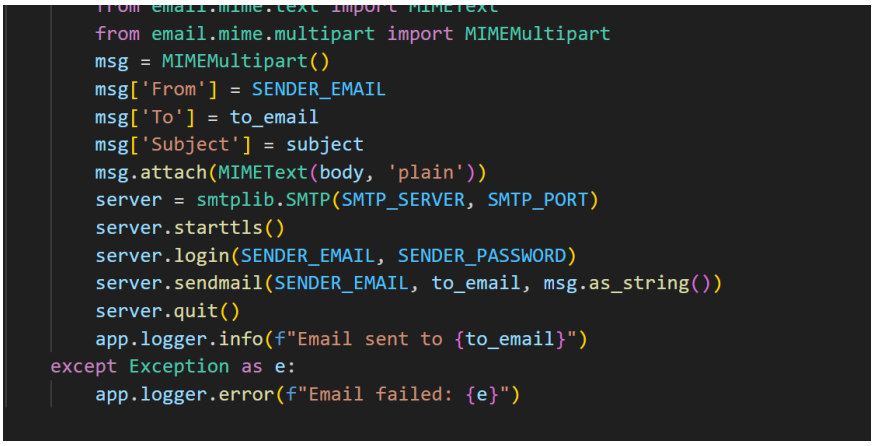


SNS Connection

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. Also, specify the chosen email service in SMTP_SERVER (e.g., Gmail, Yahoo, etc.) and enter the subscribed email in the SENDER_EMAIL section. Create an 'App password' for the email ID and store it in.



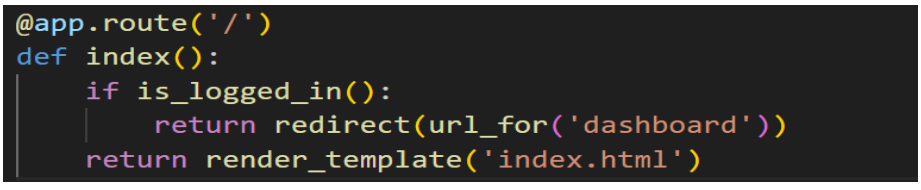
```
# ----------------------------------------
def is_logged_in():
    return 'email' in session

def get_user(email):
    response = user_table.get_item(Key={'email': email})
    return response.get('Item')

def send_email(to_email, subject, body):
    if not ENABLE_EMAIL:
        app.logger.info(f"[Email Skipped] {subject} to {to_email}")
        return
    try:
        import smtplib
        from email.mime.text import MIMEText
        from email.mime.multipart import MIMEMultipart
        msg = MIMEMultipart()
        msg['From'] = SENDER_EMAIL
        msg['To'] = to_email
        msg['Subject'] = subject
        msg.attach(MIMEText(body, 'plain'))
        server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
```



```
        from email.mime.multipart import MIMEMultipart
        msg = MIMEMultipart()
        msg['From'] = SENDER_EMAIL
        msg['To'] = to_email
        msg['Subject'] = subject
        msg.attach(MIMEText(body, 'plain'))
        server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
        server.starttls()
        server.login(SENDER_EMAIL, SENDER_PASSWORD)
        server.sendmail(SENDER_EMAIL, to_email, msg.as_string())
        server.quit()
        app.logger.info(f"Email sent to {to_email}")
    except Exception as e:
        app.logger.error(f"Email failed: {e}")
```

Routes for Web Pages:

Home Page

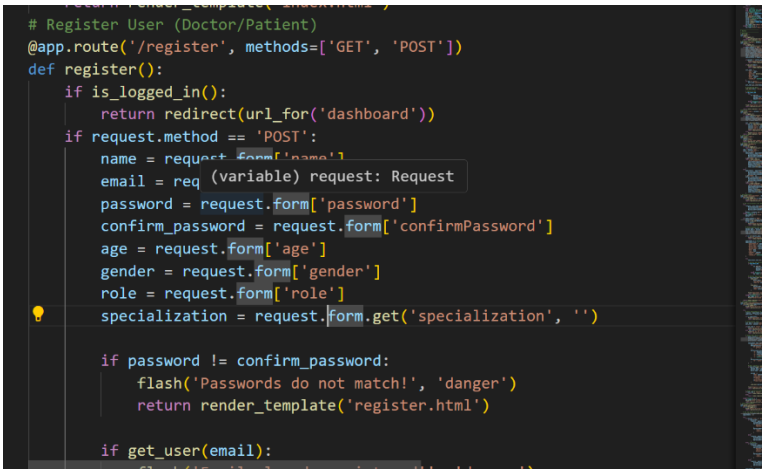


```
@app.route('/')
def index():
    if is_logged_in():
        return redirect(url_for('dashboard'))
    return render_template('index.html')
```

Register User (Doctor/Patient)

The login route handles user input for authentication, verifying credentials against stored data in DynamoDB. On successful login, it increments the login count and redirects the user to the appropriate dashboard. This process ensures secure and efficient access to the platform

Register Route (GET/POST): Verifies user credentials, increments login count, and redirects to the dashboard on success.




```python
# Register User (Doctor/Patient)
@app.route('/register', methods=['GET', 'POST'])
def register():
    if is_logged_in():
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        name = request.form['name']
        email = req
        password = request.form['password']
        confirm_password = request.form['confirmPassword']
        age = request.form['age']
        gender = request.form['gender']
        role = request.form['role']
        specialization = request.form.get('specialization', '')

        if password != confirm_password:
            flash('Passwords do not match!', 'danger')
            return render_template('register.html')

        if get_user(email):
```

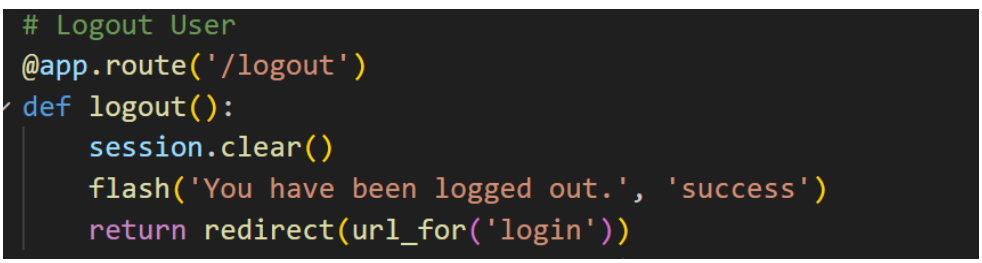
Login router

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. This ensures secure access to the platform while maintaining user activity logs.

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

        user = get_user(email)
        if user and check_password_hash(user['password'], password) and user['
            session['email'] = email
            session['role'] = role
            session['name'] = user['name']
            flash('Login successful.', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Invalid credentials.', 'danger')
    return render_template('login.html')
# Logout User
```

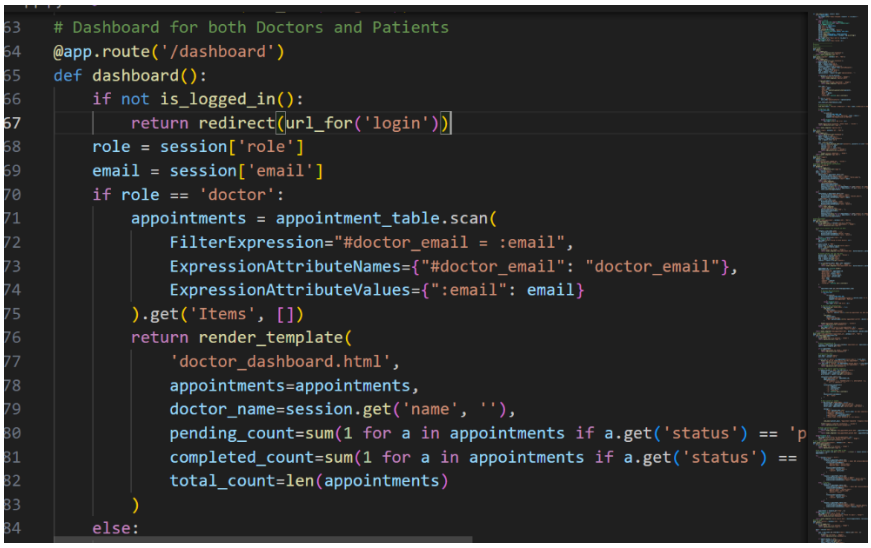
Logout Route:

The logout functionality allows users to securely end their session, clearing any session data and redirecting them to the login page. The dashboard provides users with an overview of their activities, such as upcoming appointments for patients or patient records for doctors, with relevant actions based on user roles.



```python
# Logout User
@app.route('/logout')
def logout():
    session.clear()
    flash('You have been logged out.', 'success')
    return redirect(url_for('login'))
```

Dashboard for both Doctors and Patients Router



```
# Dashboard for both Doctors and Patients
@app.route('/dashboard')
def dashboard():
    if not is_logged_in():
        return redirect(url_for('login'))
    role = session['role']
    email = session['email']
    if role == 'doctor':
        appointments = appointment_table.scan(
            FilterExpression="#doctor_email = :email",
            ExpressionAttributeNames={"#doctor_email": "doctor_email"},
            ExpressionAttributeValues={":email": email}
        ).get('Items', [])
        return render_template(
            'doctor_dashboard.html',
            appointments=appointments,
            doctor_name=session.get('name', ''),
            pending_count=sum(1 for a in appointments if a.get('status') == 'p
            completed_count=sum(1 for a in appointments if a.get('status') ==
            total_count=len(appointments)
        )
    else:
```

Book Appointment Router

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. This ensures smooth scheduling and timely updates for both patients and doctors.

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

    # Get list of doctors (for both GET and POST)
    try:
        response = user_table.scan(
            FilterExpression="#role = :role",
            ExpressionAttributeNames={"#role": "role"},
            ExpressionAttributeValues={":role": 'doctor'}
        )
        doctors = response.get('Items', [])
    except Exception as e:
        app.logger.error(f"Failed to fetch doctors: {e}")
        doctors = []

    if request.method == 'POST':
        doctor_email = request.form.get('doctor_email')
```

View Appointment Route

The view appointment route allows users to access a list of their upcoming appointments. It retrieves appointment details from DynamoDB and displays them on the user's dashboard, providing information like date, time, and doctor. This ensures users can easily track their scheduled appointments.

```python
# view appointment details
@app.route('/view_appointment/<appointment_id>', methods=['GET', 'POST'])
def view_appointment(appointment_id):
    if not is_logged_in():
        flash('Please log in to continue.', 'danger')
        return redirect(url_for('login'))

    try:
        # Fetch appointment by ID
        response = appointment_table.get_item(Key={'appointment_id': appointme
        appointment = response.get('Item')

        if not appointment:
            flash('Appointment not found.', 'danger')
            return redirect(url_for('dashboard'))

        # Authorization check
        user_email = session['email']
        user_role = session['role']

        if user_role == 'doctor' and appointment['doctor_email'] != user_email
            flash('You are not authorized to view this appointment.', 'danger'
```

Search Appointment Router

The search appointment route enables users to search for specific appointments by date, doctor, or status. It queries DynamoDB to fetch relevant appointment details and displays the results in real-time, allowing users to quickly find specific appointments from their history or upcoming schedule.

```python
# search appointments
@app.route('/search_appointments', methods=['GET', 'POST'])
def search_appointments():
    if not is_logged_in():
        flash('Please log in to continue.', 'danger')
        return redirect(url_for('login'))

    # Get the search term from either POST or GET
    search_term = request.form.get('search_term', '').strip() if request.metho
    appointments = []

    try:
        if session['role'] == 'doctor':
            if search_term:
                response = appointment_table.scan(
                    FilterExpression="#doctor_email = :email AND contains(#pat
                    ExpressionAttributeNames={
                        "#doctor_email": "doctor_email",
                        "#patient_name": "patient_name"
                    },
                    ExpressionAttributeValues={
                        ":email": session['email'],
```

Profile Router

The profile route allows users to view and update their personal information, such as name, contact details, and medical history. It retrieves user data from DynamoDB and displays it on the dashboard. Users can also make changes to their profile, which are then saved securely in the database.

```python
#profile page
@app.route('/profile', methods=['GET', 'POST'])
def profile():
    if not is_logged_in():
        flash('Please log in to continue.', 'danger')
        return redirect(url_for('login'))

    email = session['email']
    try:
        user = user_table.get_item(Key={'email': email}).get('Item', {})
        if not user:
            flash('User not found.', 'danger')
            return redirect(url_for('dashboard'))

        if request.method == 'POST':
            name = request.form.get('name')
            age = request.form.get('age')
            gender = request.form.get('gender')

            update_expression = "SET #name = :name, age = :age, gender = :gend
            expression_values = {
                ':name': name,
```

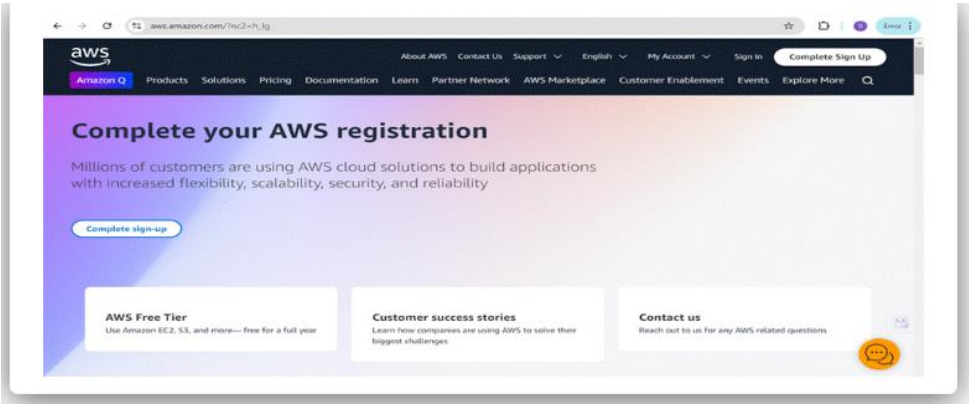
Deployment Code

The health routing feature in the MedTrack project checks the system's status by sending a request to a specific endpoint, ensuring the backend services are functioning properly. The __name__ == '__main__' block is used in the Flask app to ensure that the application runs only if the script is executed directly, not when imported as a module, enabling local development or deployment on a server. This setup ensures that the app runs smoothly and is self-contained during execution.

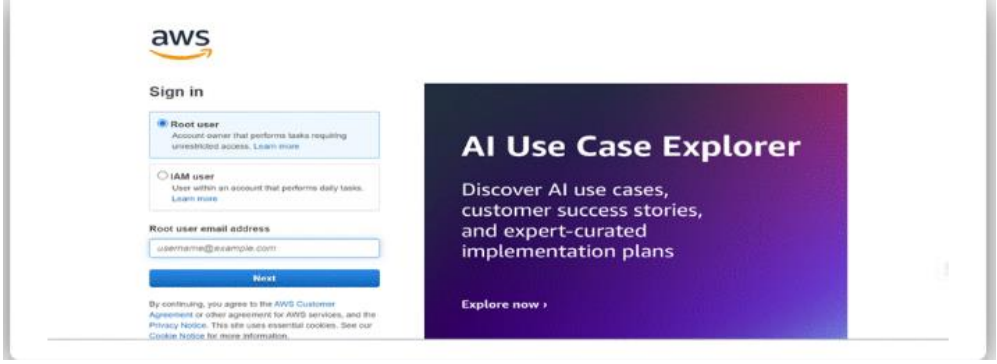
```
        flash('Diagnosis submitted successfully.', 'success')
        return redirect(url_for('dashboard'))

    except Exception as e:
        app.logger.error(f"Submit diagnosis error: {e}")
        flash('An error occurred while submitting the diagnosis. Please try ag
        return redirect(url_for('view_appointment', appointment_id=appointment
# ------------------------------------
# Entrypoint for WSGI (production servers)
# ------------------------------------
application = app  # For gunicorn, uWSGI, etc.

# For local development
if __name__ == '__main__':
    port = int(os.environ.get('PORT', 5000))
    debug_mode = os.environ.get('FLASK_DEBUG', 'False').lower() == 'true'
    app.run(host='0.0.0.0', port=port, debug=debug_mode)
```

Milestone 2. AWS Account Setup and Login

- Go to the AWS website (https://aws.amazon.com/).
- Click on the "Create an AWS Account" button.
- Follow the prompts to enter your email address and choose a password.
- Provide the required account information, including your name, address, and phone number.
- Enter your payment information. (Note: While AWS offers a free tier, a credit card or debit card is required for verification.)
- Complete the identity verification process.
- Choose a support plan (the basic plan is free and sufficient for starting).
- Once verified, you can sign in to your new AWS accounts.



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

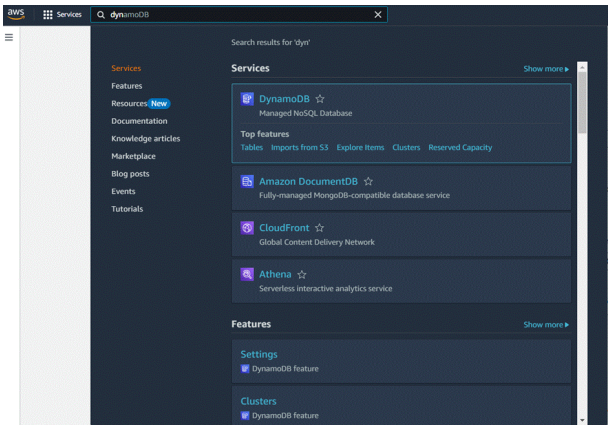


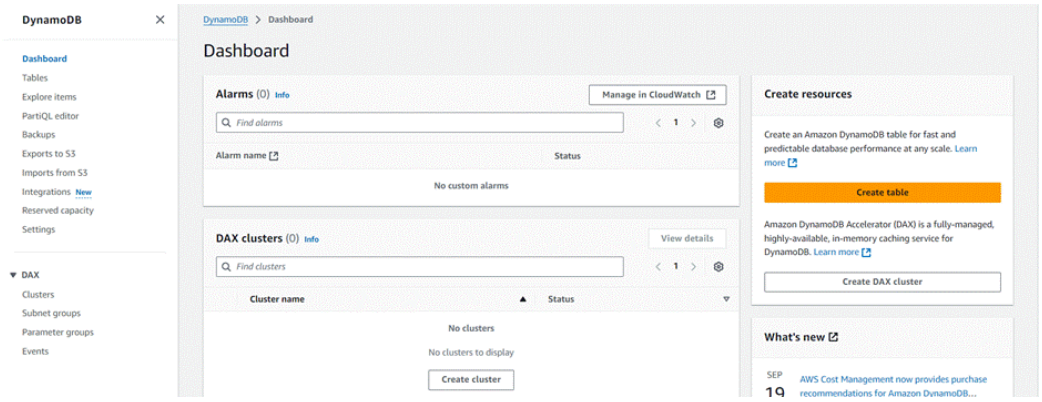
Milestone 3: DynamoDB Database Creation and Setup

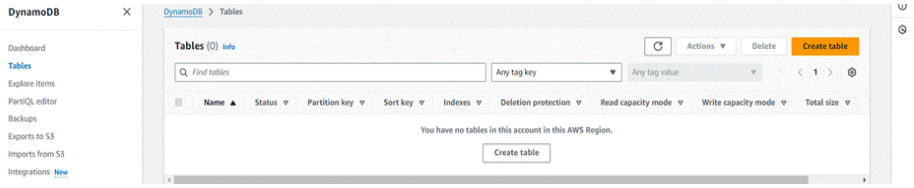
Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

Navigate to the DynamoDB

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

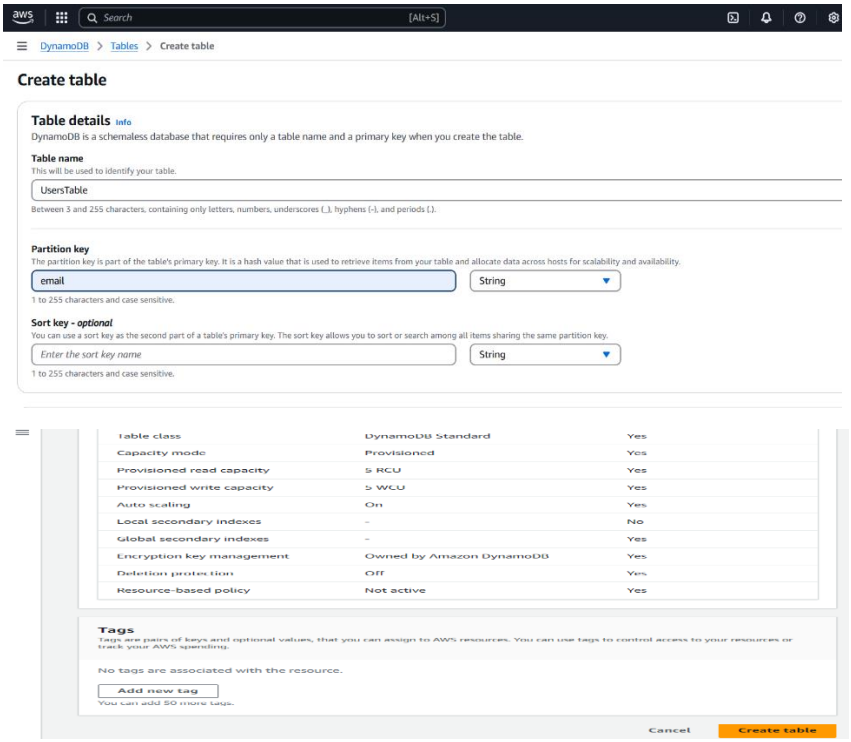




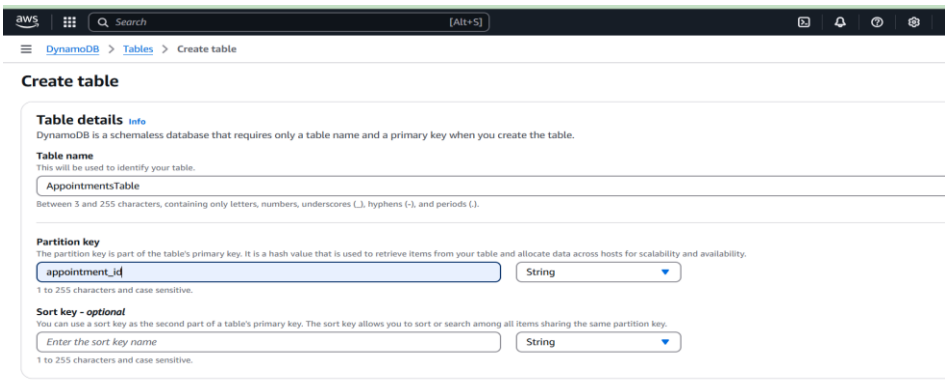


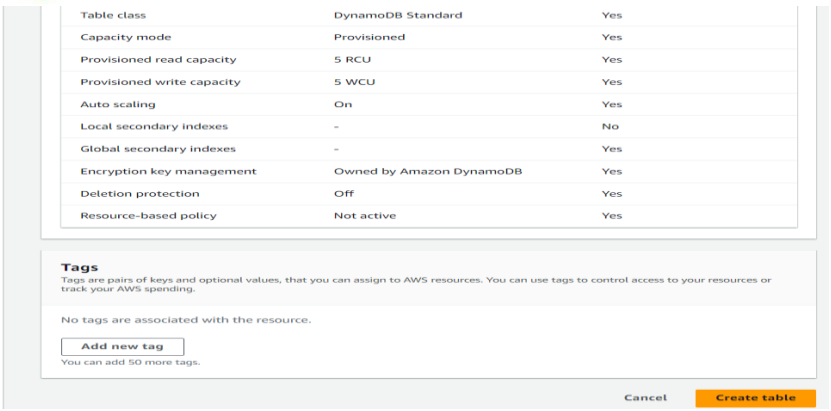
Create a DynamoDB table for storing data

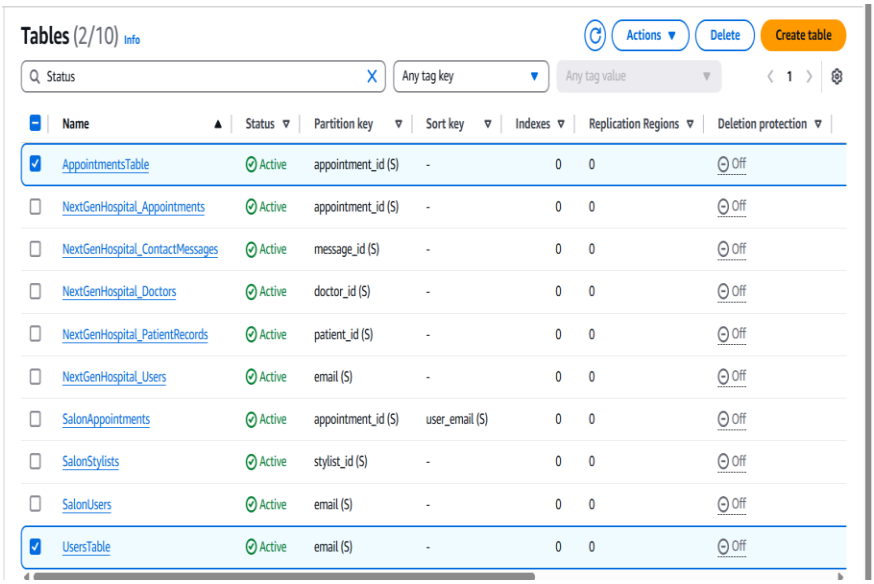
- Create Users table with partition key “Email” with type String and click on create tables.



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





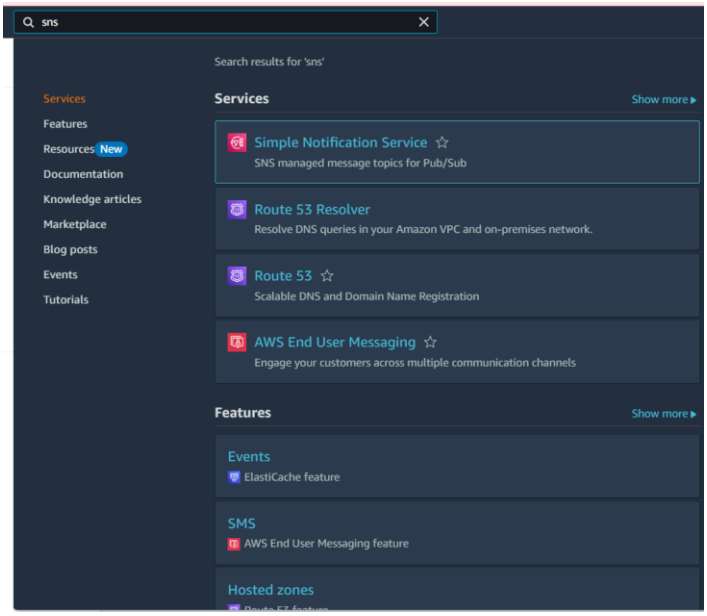


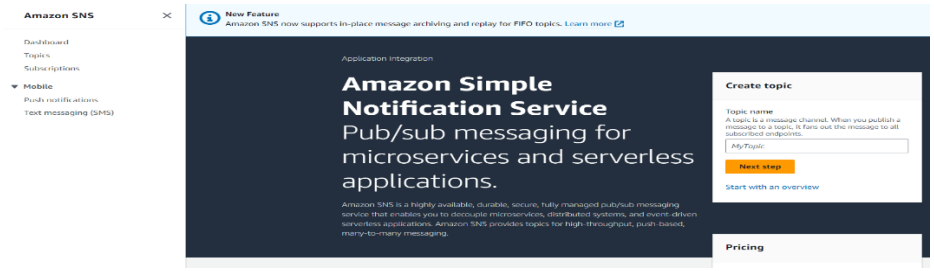
Milestone 4 : SNS Notification Setup

Amazon SNS is a fully managed messaging service that enables real-time notifications through channels like SMS, email, or app endpoints. You create topics, configure subscriptions, and integrate SNS into your app to send notifications based on specific events.

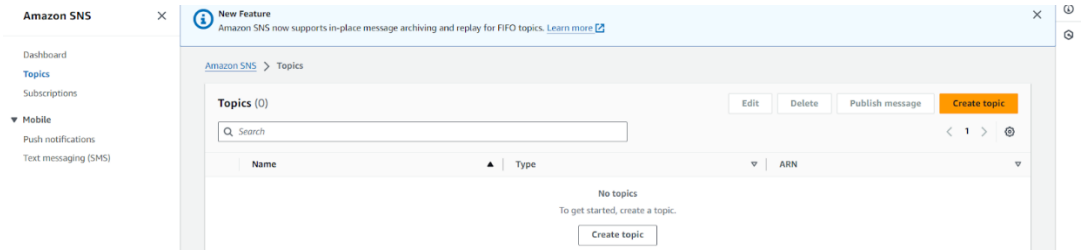
SNS topics for email notifications

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

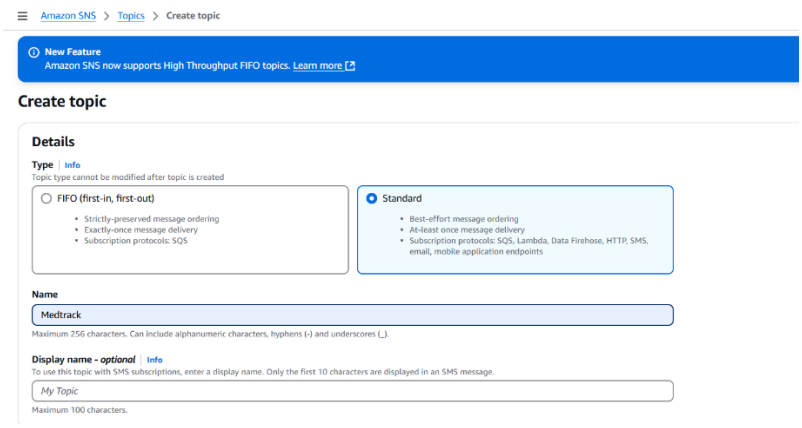


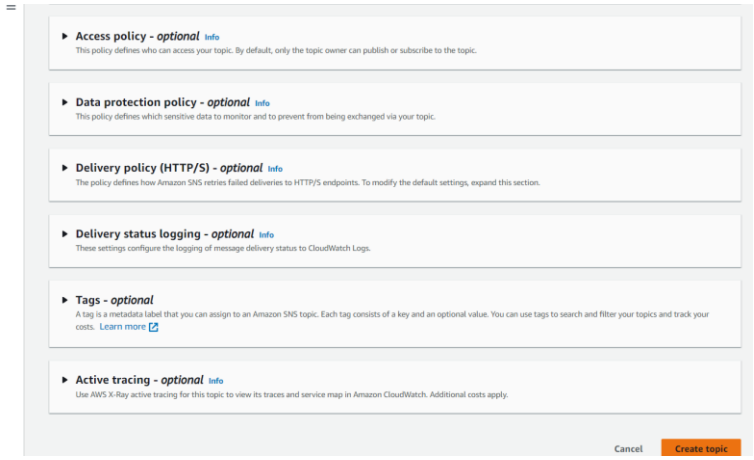


- Click on **Create Topic** and choose a name for the topic.

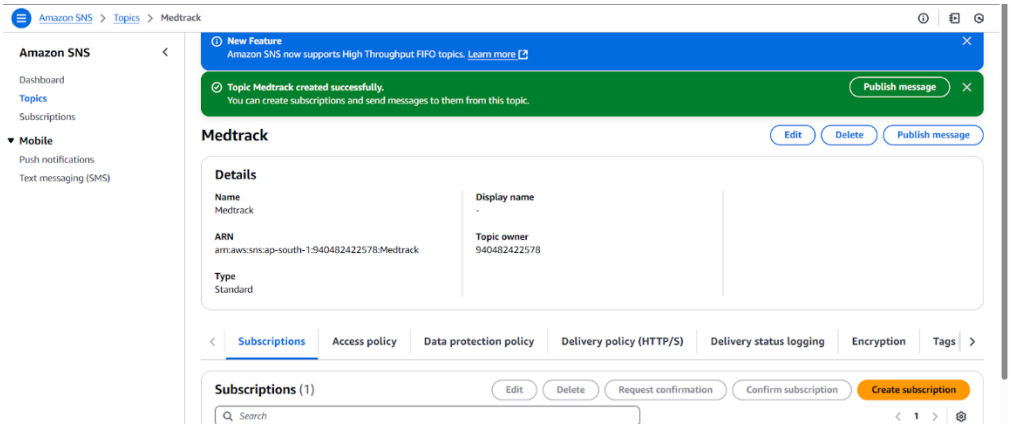


- Choose Standard type for general notification use cases and Click on Create Topic.



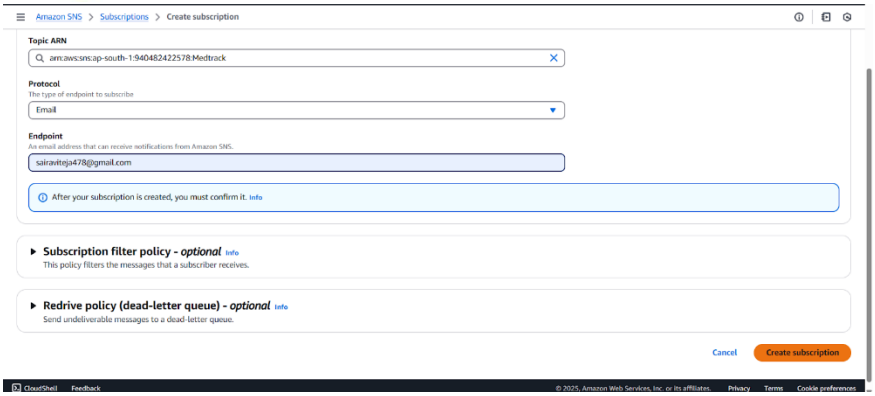


- Configure the SNS topic and note down the **Topic ARN**.

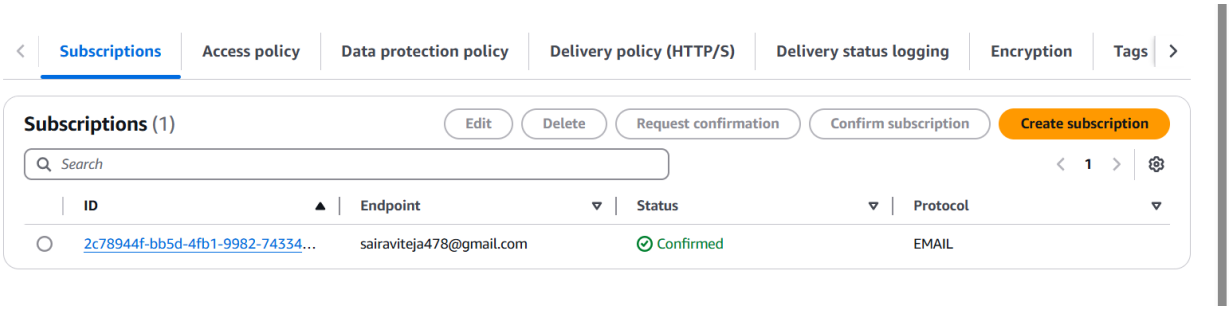


Subscribe users and Admin

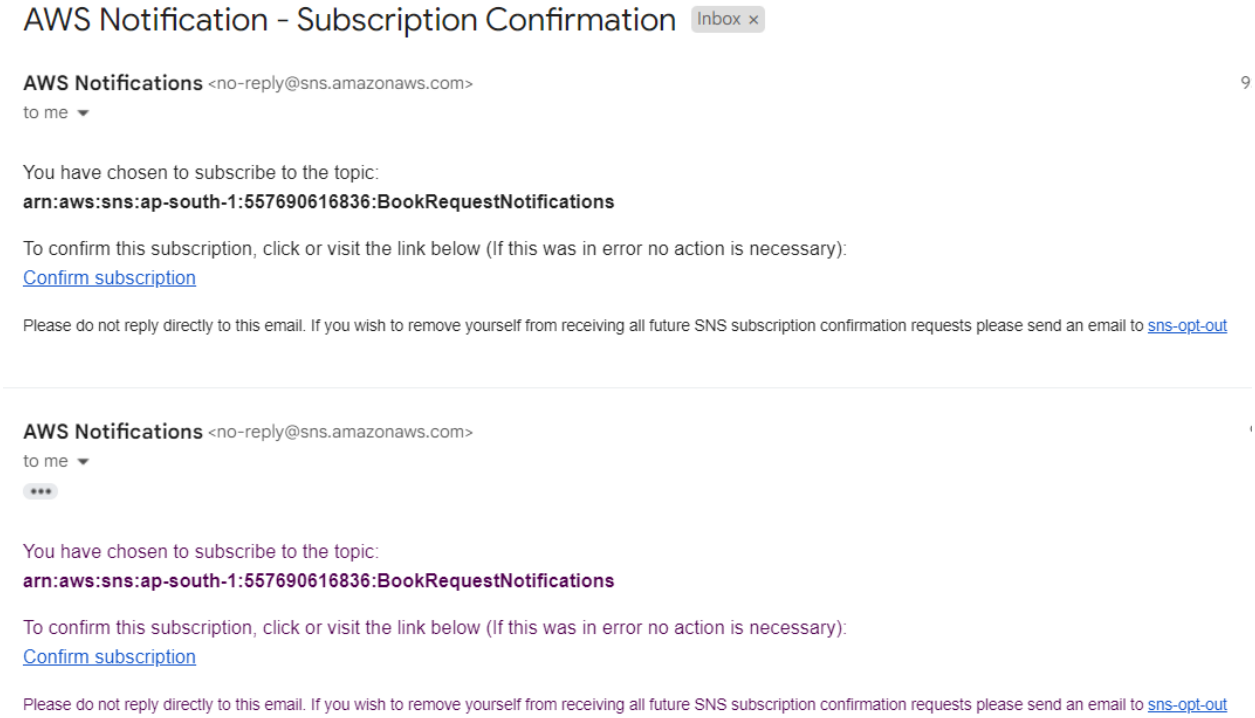
- Subscribe users (or admin staff) to this topic via email. When a book request is made, notifications will be sent to the subscribed emails.



- 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.



Simple Notification Service

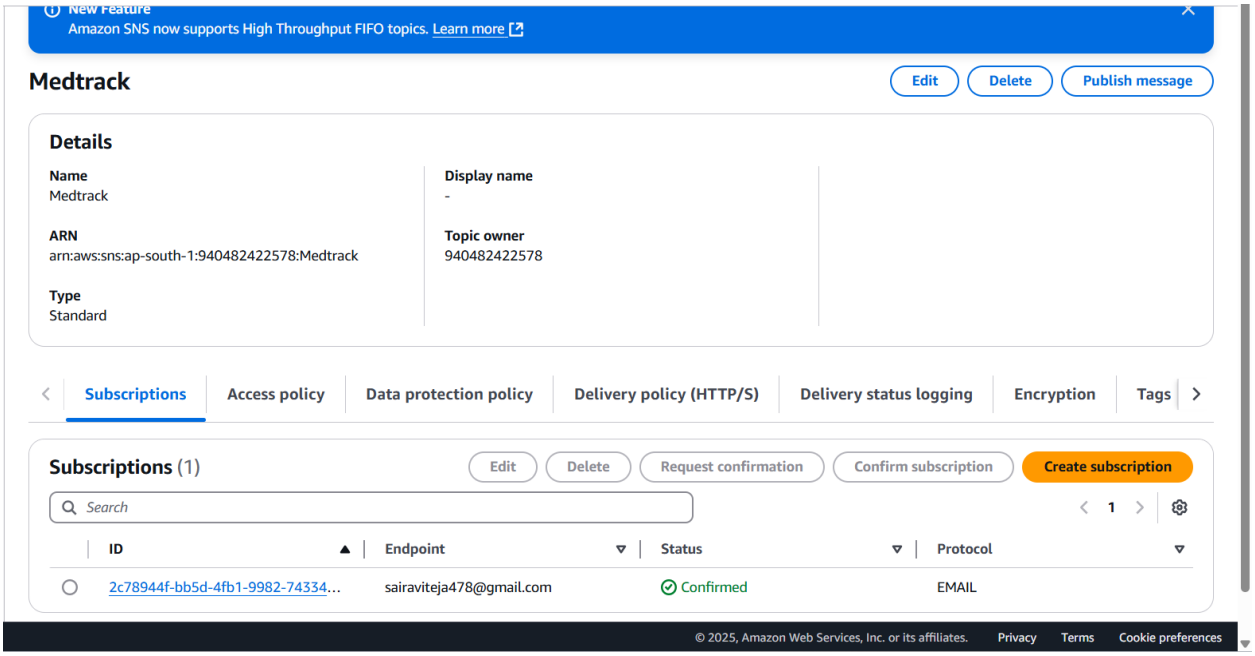
Subscription confirmed!

You have successfully subscribed.

Your subscription's id is:
arn:aws:sns:ap-south-1:557690616836:BookRequestNotifications:d78e0371-9235-404d-952c-85c2743607c4

If it was not your intention to subscribe, click here to unsubscribe.

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

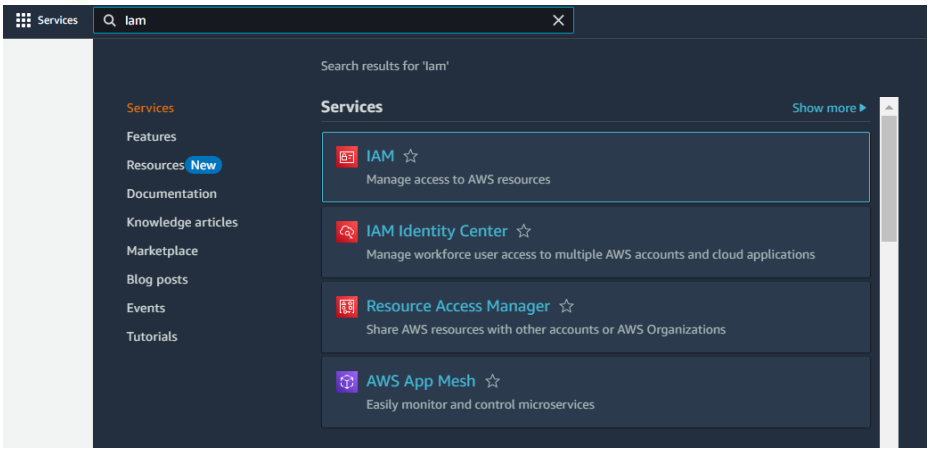


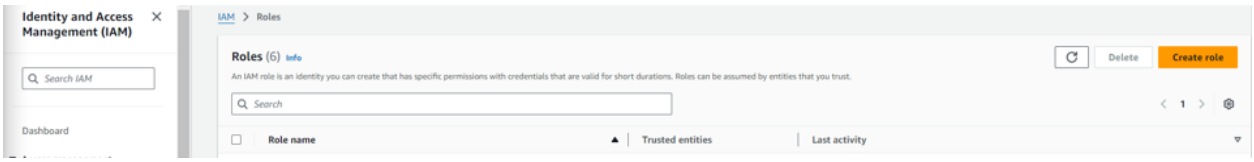
Milestone 5: IAM Role Setup

IAM (Identity and Access Management) role setup involves creating roles that define specific permissions for AWS services. To set it up, you create a role with the required policies, assign it to users or services, and ensure the role has appropriate access to resources like EC2, S3, or RDS. This allows controlled access and ensures security best practices in managing AWS resources.

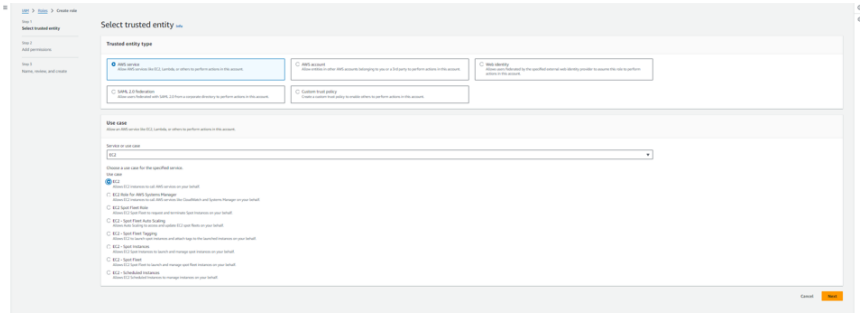
Create IAM Role.

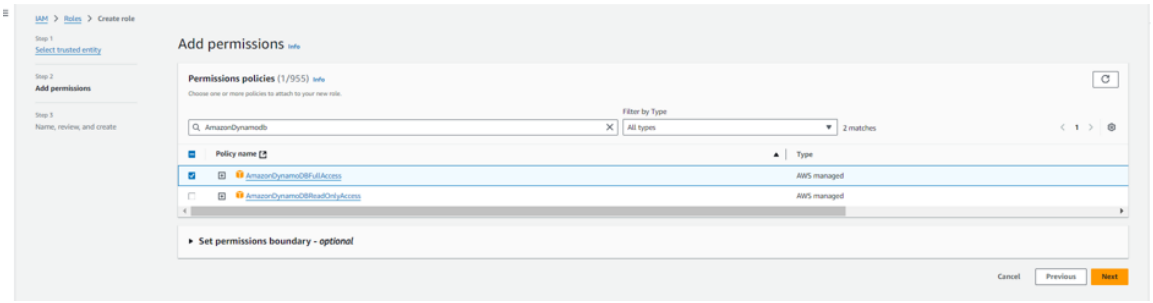
- In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.





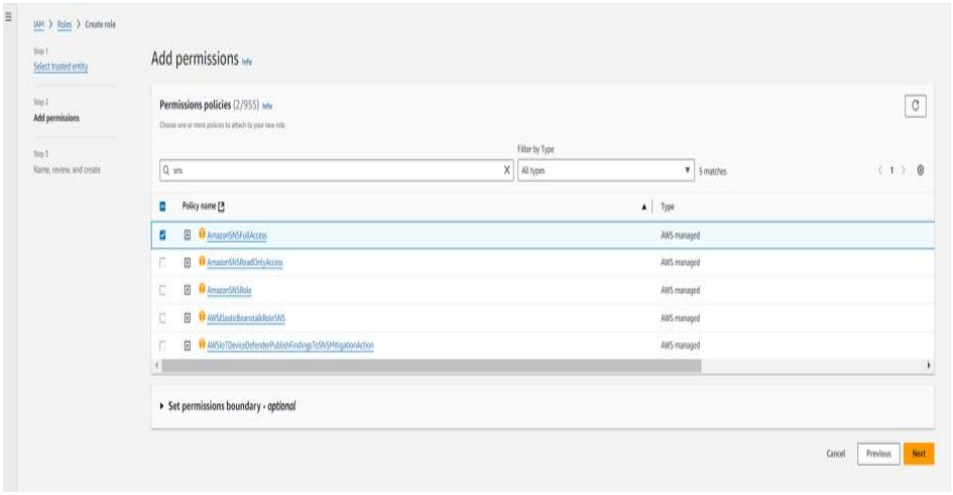
To create and select DynamoDBFullAccess and SNSFullAccess, go to the AWS IAM console, create a new role, and assign the respective policies. DynamoDBFullAccess allows full access to DynamoDB resources, while SNSFullAccess enables sending notifications via SNS. Attach the role to the relevant services to ensure proper integration with the project.



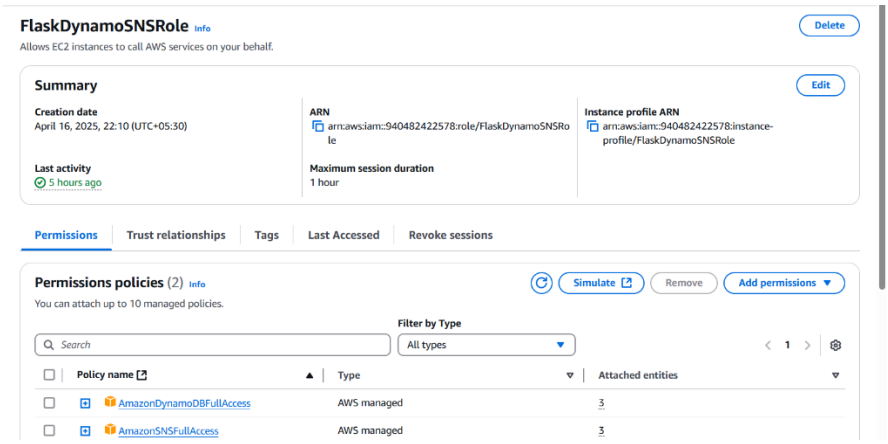


Attach Policies.
Attach the following policies to the role:

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



To create a role named flaskdynamodbsns, go to the AWS IAM console, create a new role, and assign DynamoDBFullAccess and SNSFullAccess policies. Name the role flaskdynamodbsns and attach it to the necessary AWS services. This role will allow your Flask app to interact with both DynamoDB and SNS seamlessly.

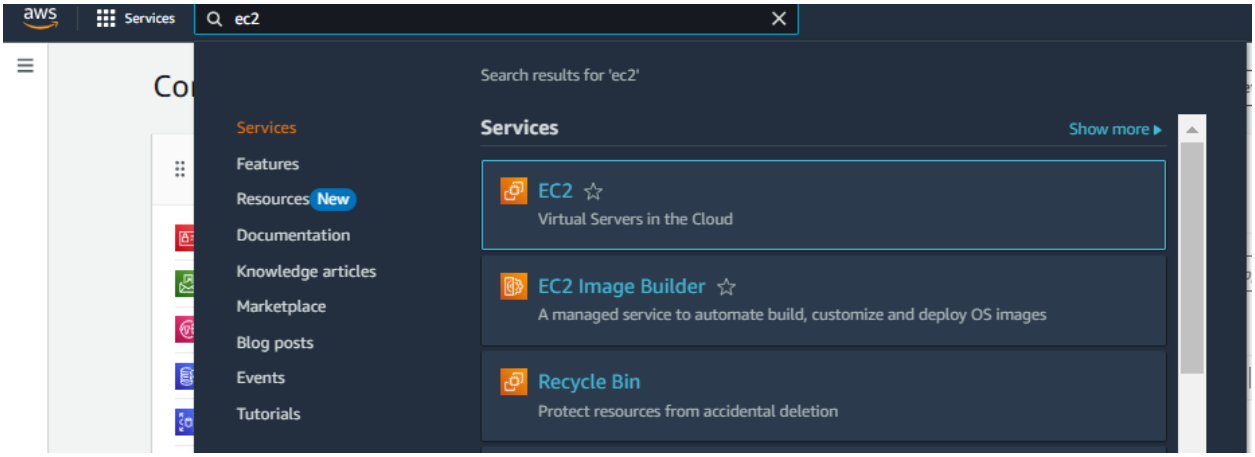


Milestone 6: EC2 Instance setup

To set up a public EC2 instance, choose an appropriate Amazon Machine Image (AMI) and instance type. Ensure the security group allows inbound traffic on necessary ports (e.g., HTTP/HTTPS for web applications). After launching the instance, associate it with an Elastic IP for consistent public access, and configure your application or services to be publicly accessible.

Launch an EC2 instance to host the Flask application.

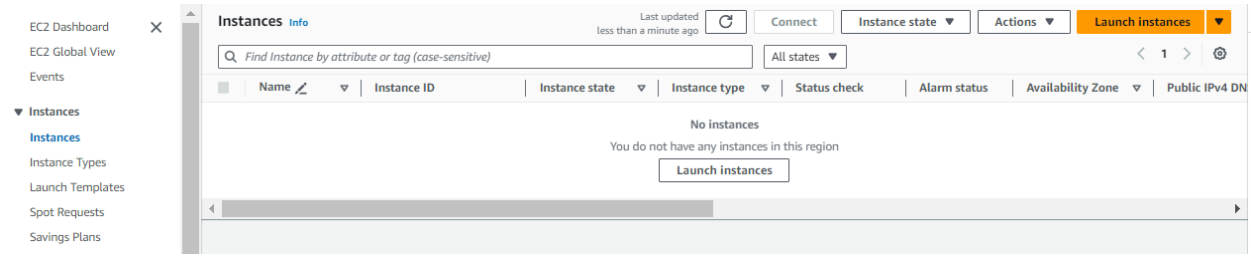
- **Launch EC2 Instance**
  - In the AWS Console, navigate to EC2 and launch a new instance.

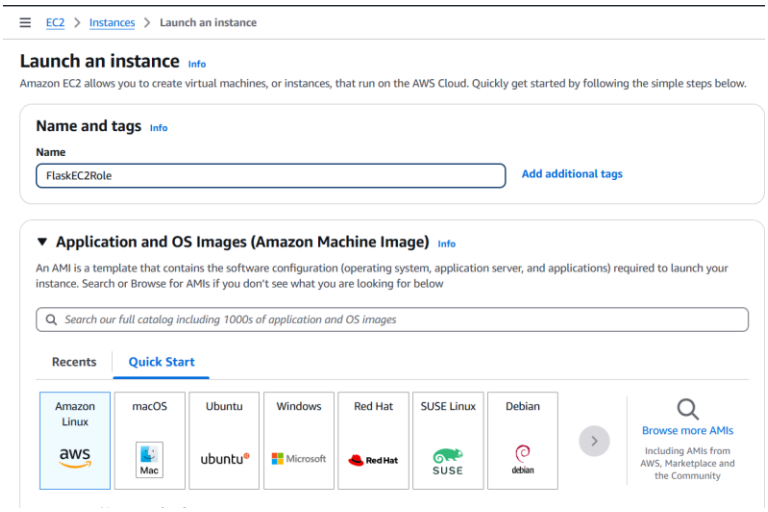


To launch an EC2 instance with the name **flaskec2role**, follow these steps:

1. Go to the **AWS EC2 Dashboard** and click on **Launch Instance**.

2. Select your desired AMI, instance type, configure instance details, and under **IAM role**, choose the role **flaskec2role**. Finally, launch the instance.

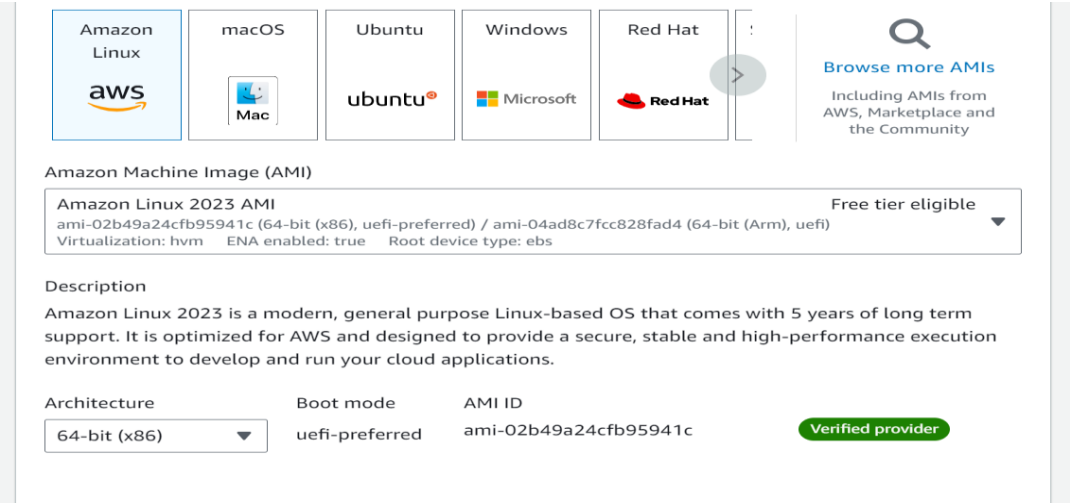




To launch an EC2 instance with **Amazon Linux 2** or **Ubuntu** as the AMI and **t2.micro** as the instance type (free-tier eligible):

1. In the **Launch Instance** wizard, choose **Amazon Linux 2** or **Ubuntu** from the available AMIs.

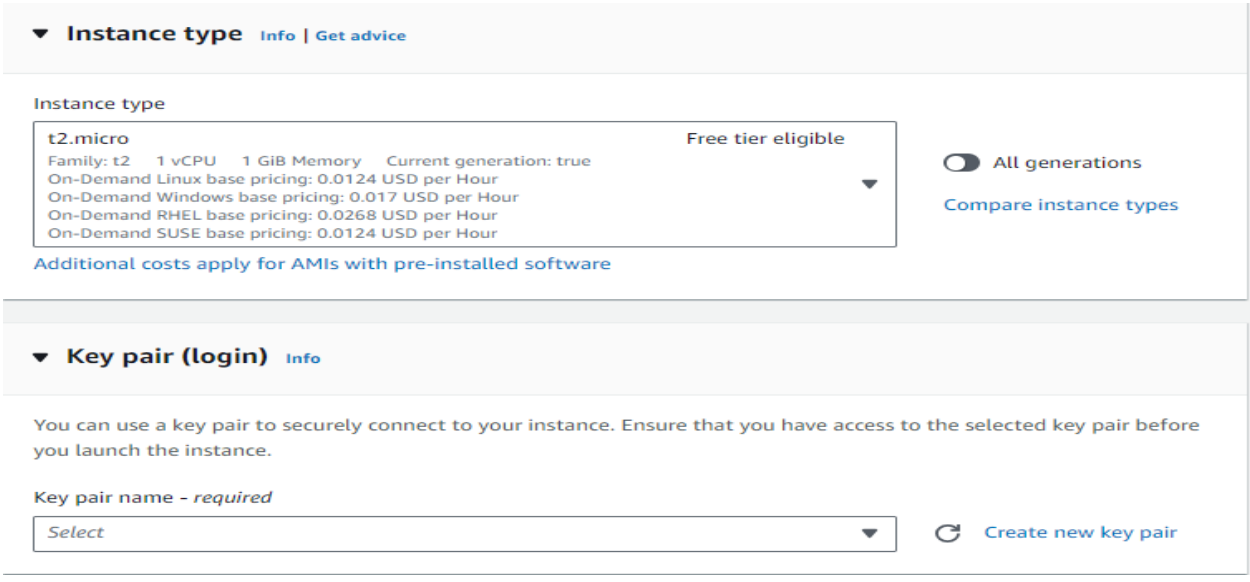
2. Select **t2.micro** as the instance type, which is free-tier eligible, and continue with the configuration and launch steps.

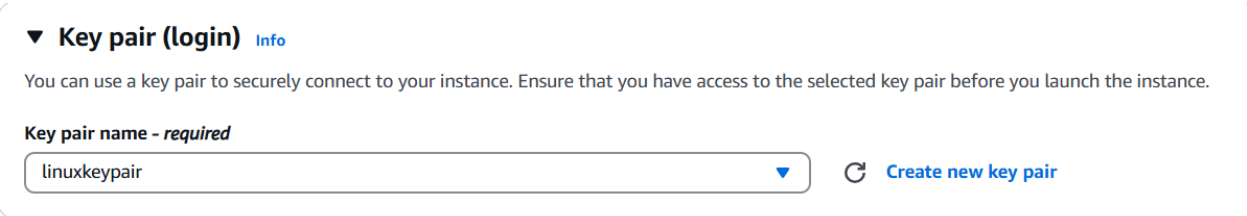


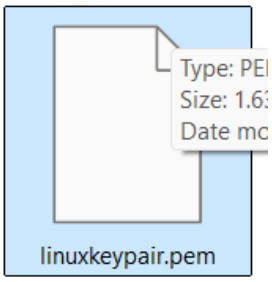
To create and download the key pair for server access:

1. In the **Launch Instance** wizard, under the **Key Pair** section, click **Create a new key pair**.

2. Name your key pair (e.g., **flaskkeypair**) and click **Download Key Pair**. This will download the .pem file to your system, which you will use to access the EC2 instance securely via SSH.







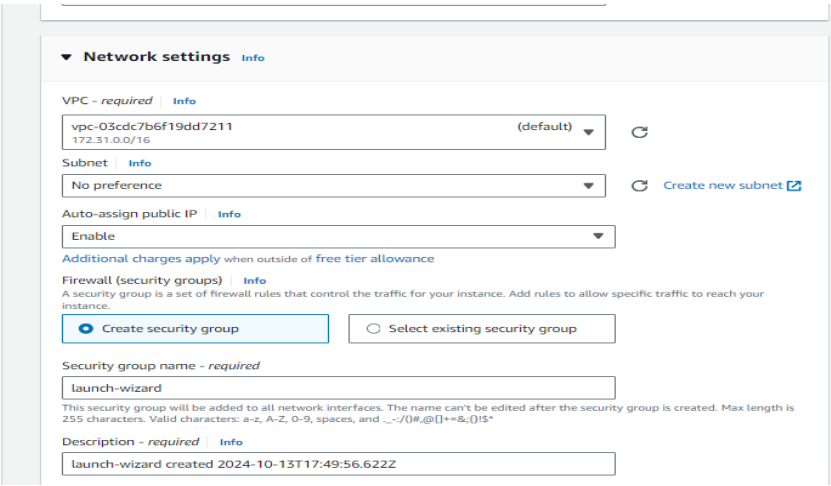
Configure security groups for HTTP, and SSH access

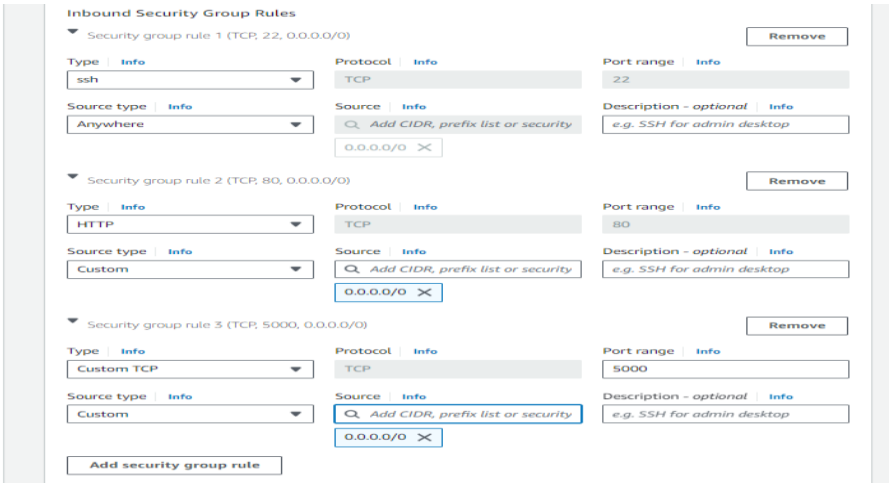
For network settings during EC2 instance launch:

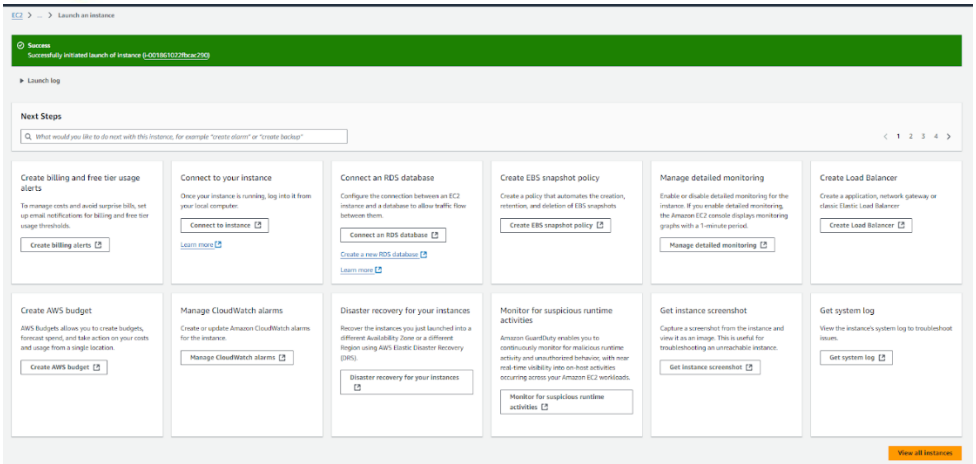
1. In the **Network Settings** section, select the **VPC** and **Subnet** you wish to use (if unsure, the default VPC and subnet should work).

2. Ensure **Auto-assign Public IP** is enabled so your instance can be accessed from the internet.

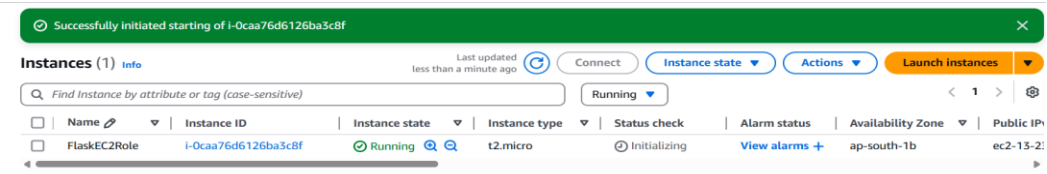
3. In **Security Group**, either select an existing one or create a new one that allows SSH (port 22) access to your EC2 instance for remote login.



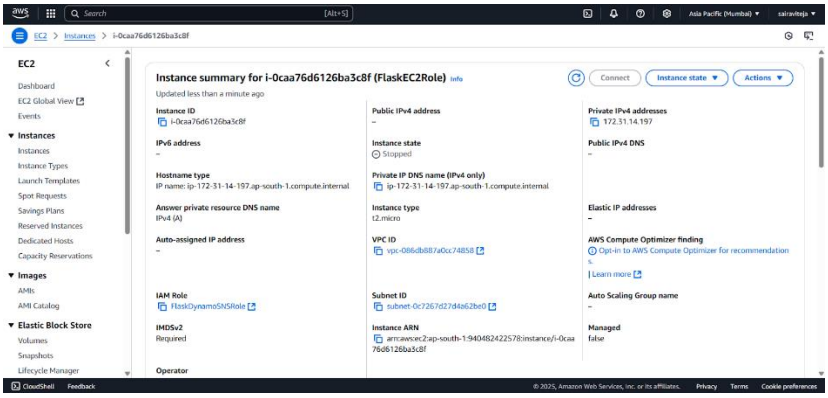




To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected,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



The EC2 instance you are launching is configured with Amazon Linux 2 or Ubuntu as the AMI, t2.micro as the instance type (free-tier eligible), and flaskec2role IAM role for appropriate permissions. The flaskkeypair key pair is created for secure server access via SSH, and the instance is set to auto-assign a public IP for internet accessibility. The security group is configured to allow SSH (port 22) access for remote login.

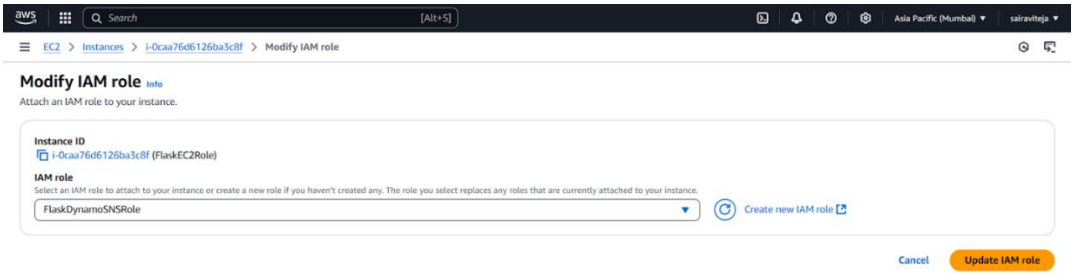


To modify the **IAM role** for your EC2 instance:

1. Go to the **AWS IAM Console**, select **Roles**, and find the **flaskec2role**.

2. Click **Attach Policies**, then choose the required policies (e.g., **DynamoDBFullAccess**, **SNSFullAccess**) and click **Attach Policy**.

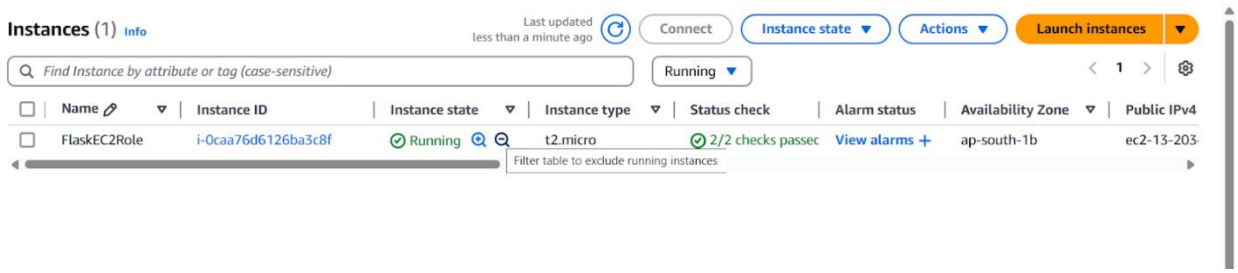
3. If needed, update the instance to use this modified role by selecting the EC2 instance, clicking **Actions**, then **Security**, and **Modify IAM role** to select the updated role.



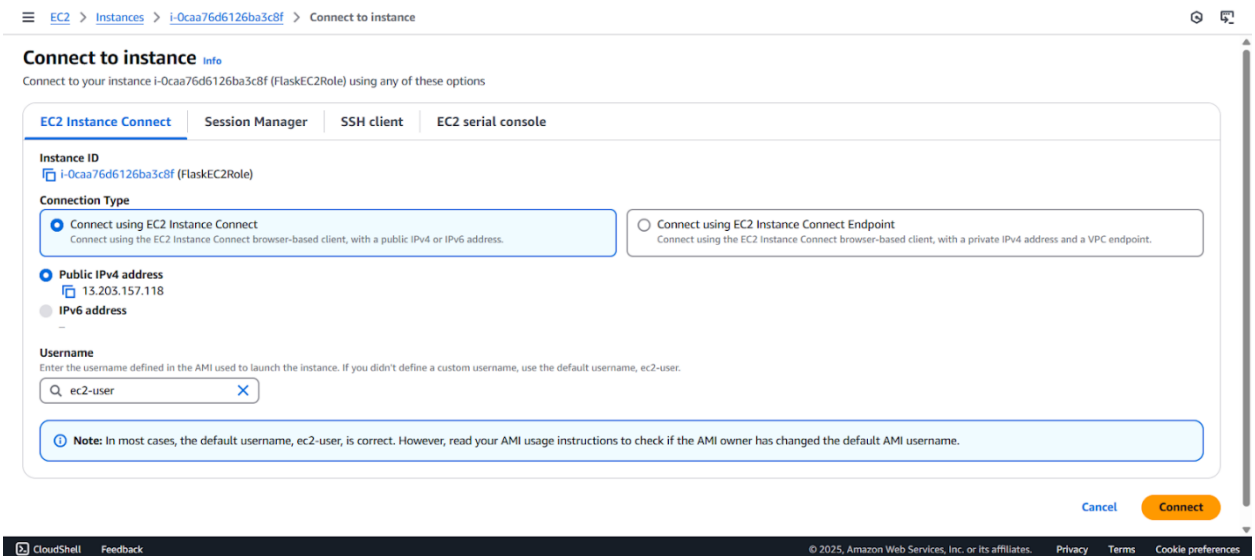
To connect to your EC2 instance:

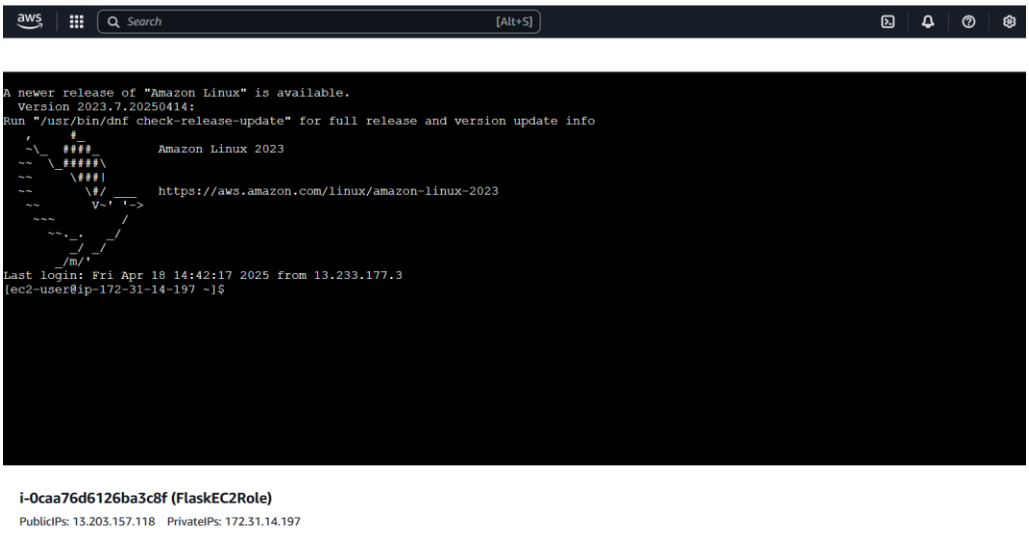
1. Go to the **EC2 Dashboard**, select your running instance, and click **Connect**.

2. Follow the instructions provided in the **Connect To Your Instance** dialog, which will show the SSH command (e.g., ssh -i flaskkeypair.pem ec2-user@<public-ip>) to access your instance using the downloaded .pem key.



- 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.

Install Software on the EC2 Instance

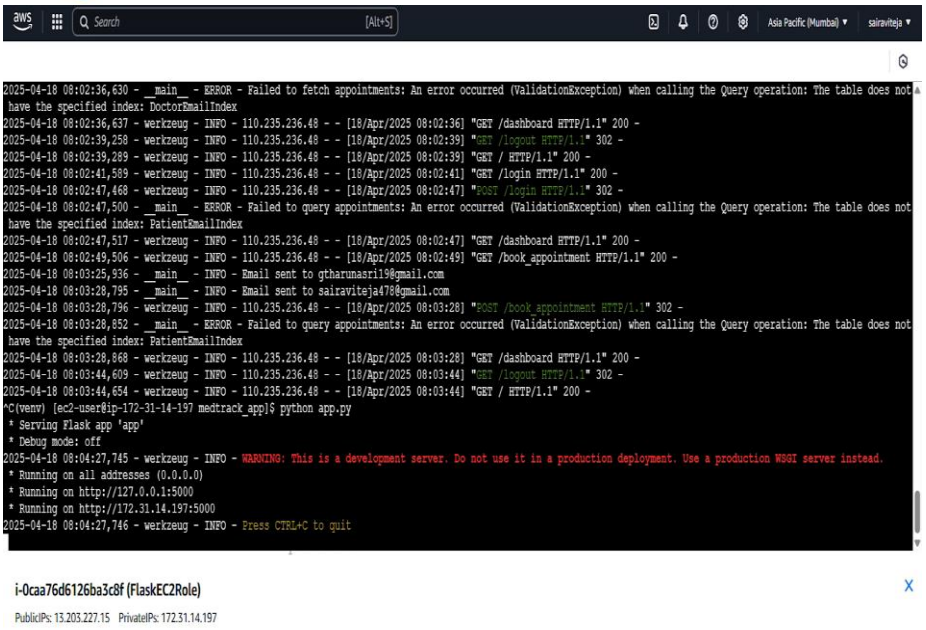
Install Python3, Flask, and Git:
On Amazon Linux 2:
sudo yum update -y
sudo yum install python3 git
sudo pip3 install flask boto3

Verify Installations:
flask --version
git –version

Clone Your Flask Project from GitHub

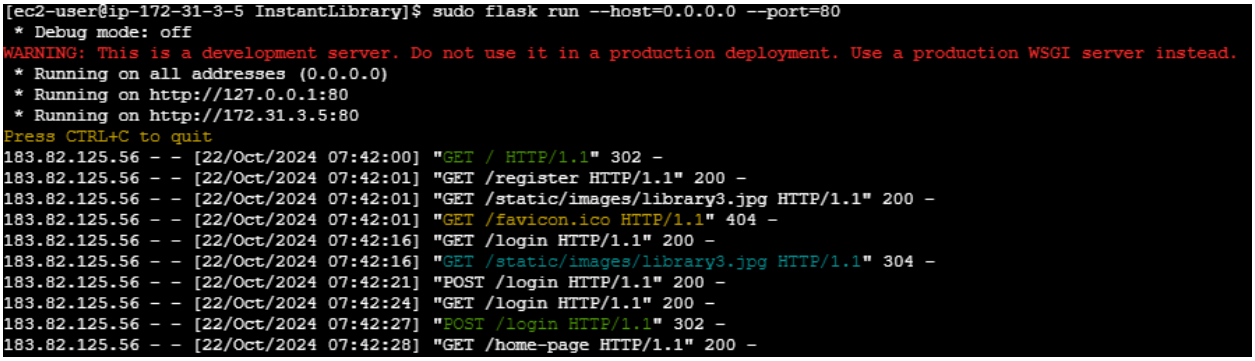
Clone your project repository from GitHub into the EC2 instance using Git.
Run:
'https://github.com/ArchanaPulakhandam/MedTrack_app'

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:
PublicIPs: https://13.201.72.132:5000/

Milestone 8: Testing and Deployment

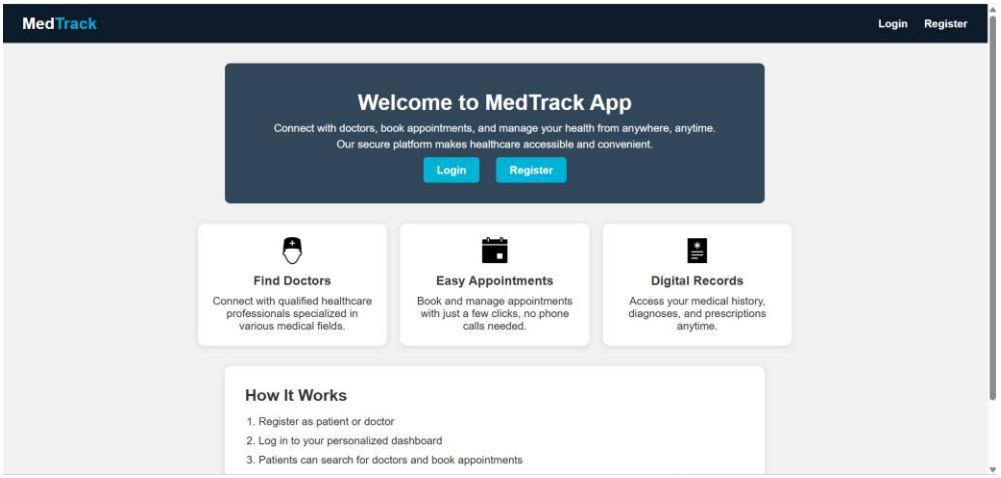
Home Page:
The Home Page of your project is the main entry point for users, where they can interact with the system. It typically includes:

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.



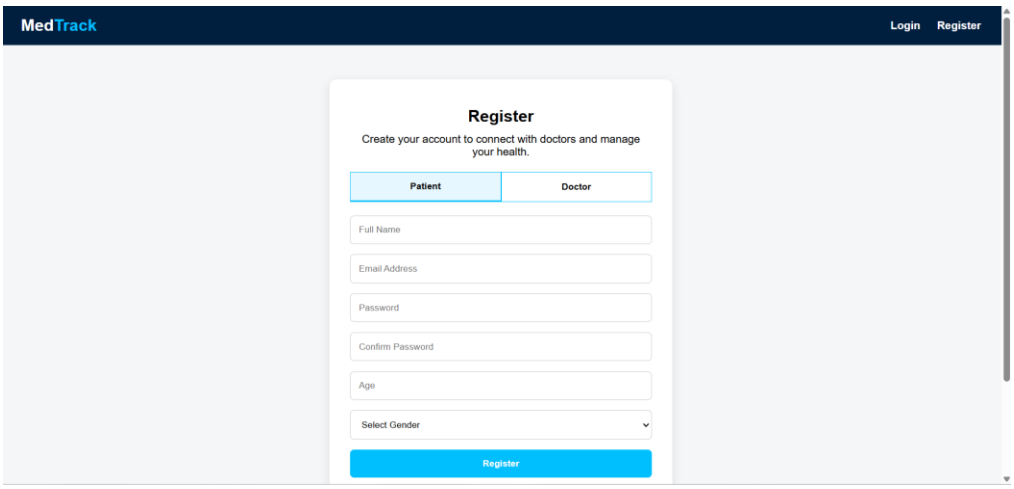
Doctor and Patient register page:

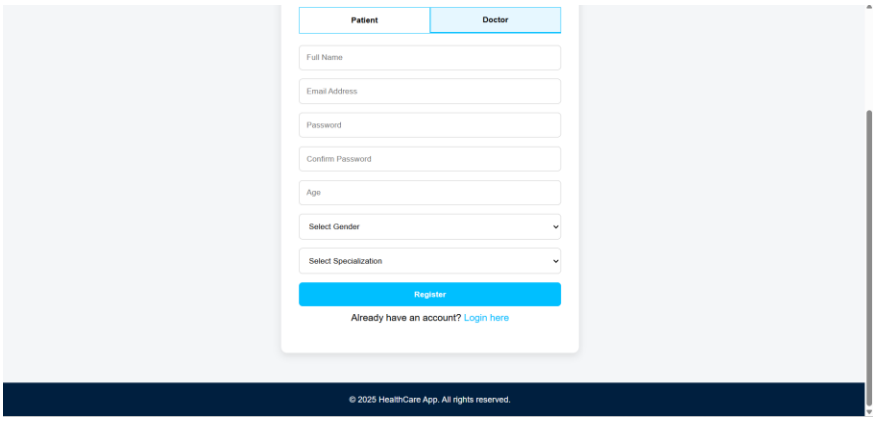
The Doctor Registration Page allows doctors to register and create an account on the platform. It typically includes:

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





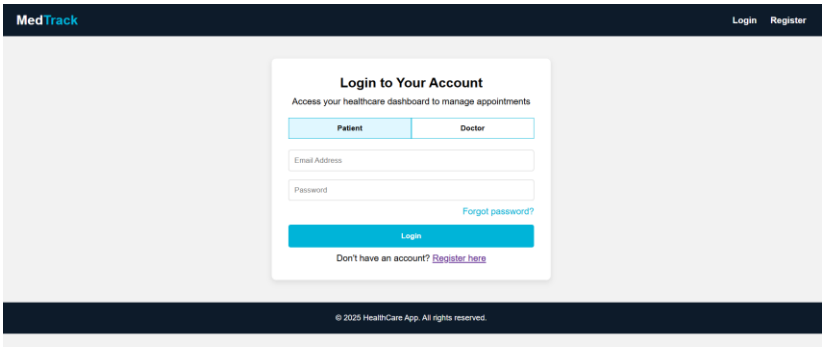
Doctor and Patient login page:

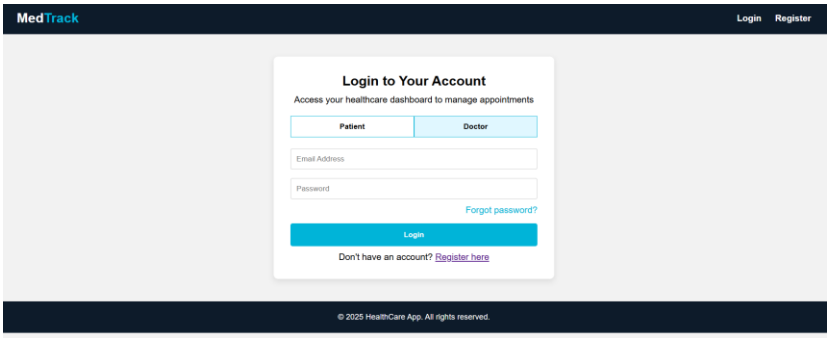
The Patient and Doctor Login Pages allow users to securely access their accounts on the platform. Each login page typically includes:

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.





User Dashboard:

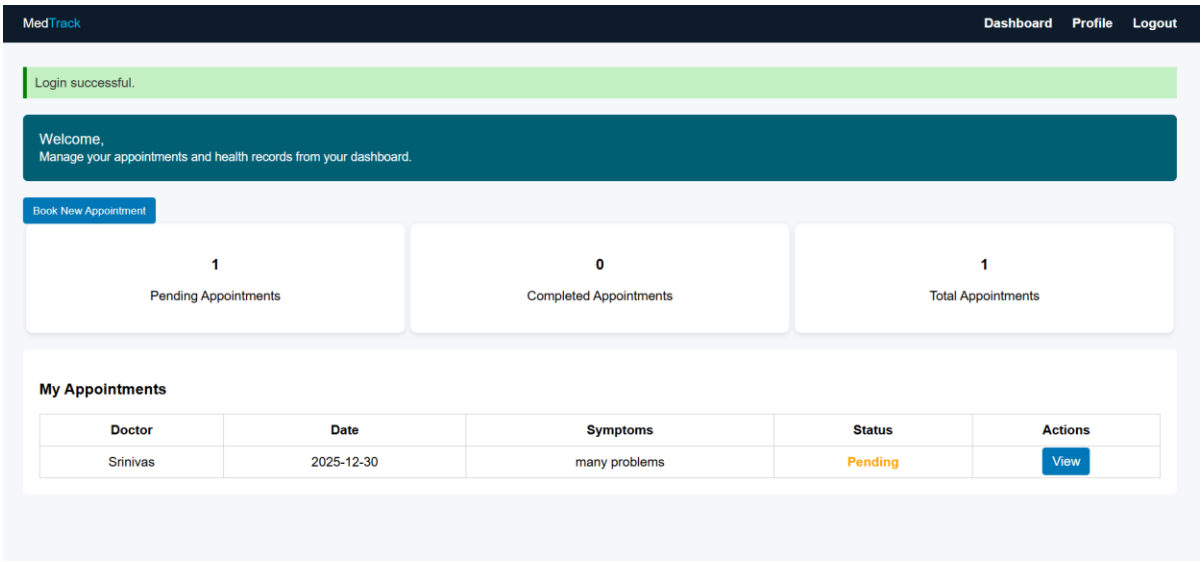
The User Dashboard (for patients) provides an easy interface to manage appointments and track their status. It typically includes:

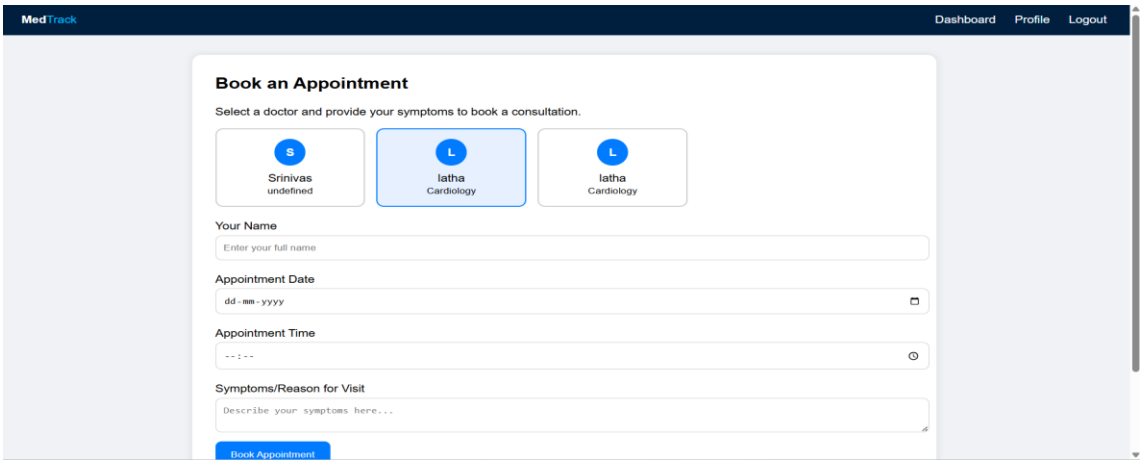
1. Book Appointment Section: A form for selecting a doctor, choosing an appointment time, and submitting the request.

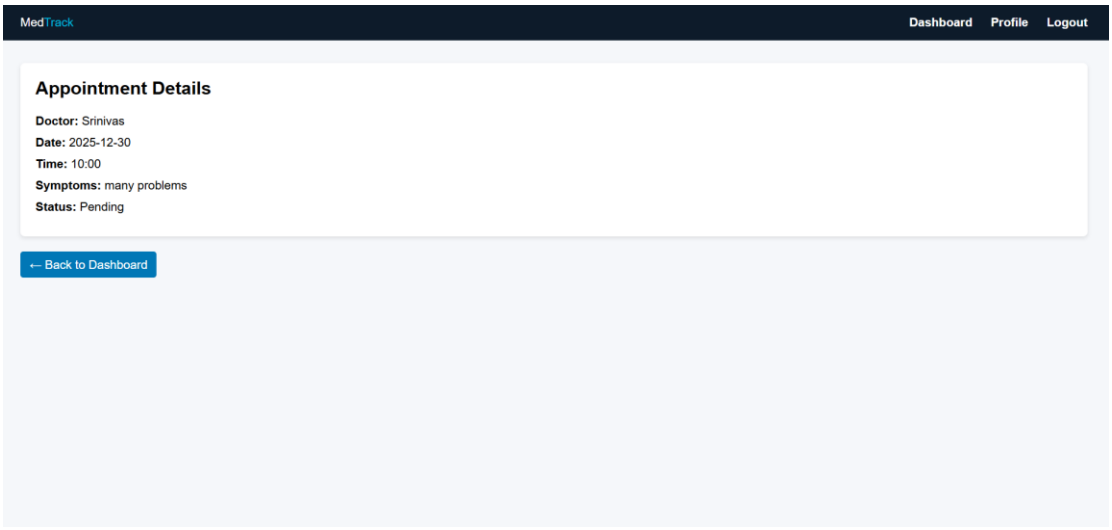
2. Appointment Status: A section showing the current status of appointments (e.g., confirmed, pending, or completed) with options to view details or cancel.

3. Upcoming Appointments: A list of future appointments with relevant details such as doctor name, date, and time.

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







Doctor Dashboard :

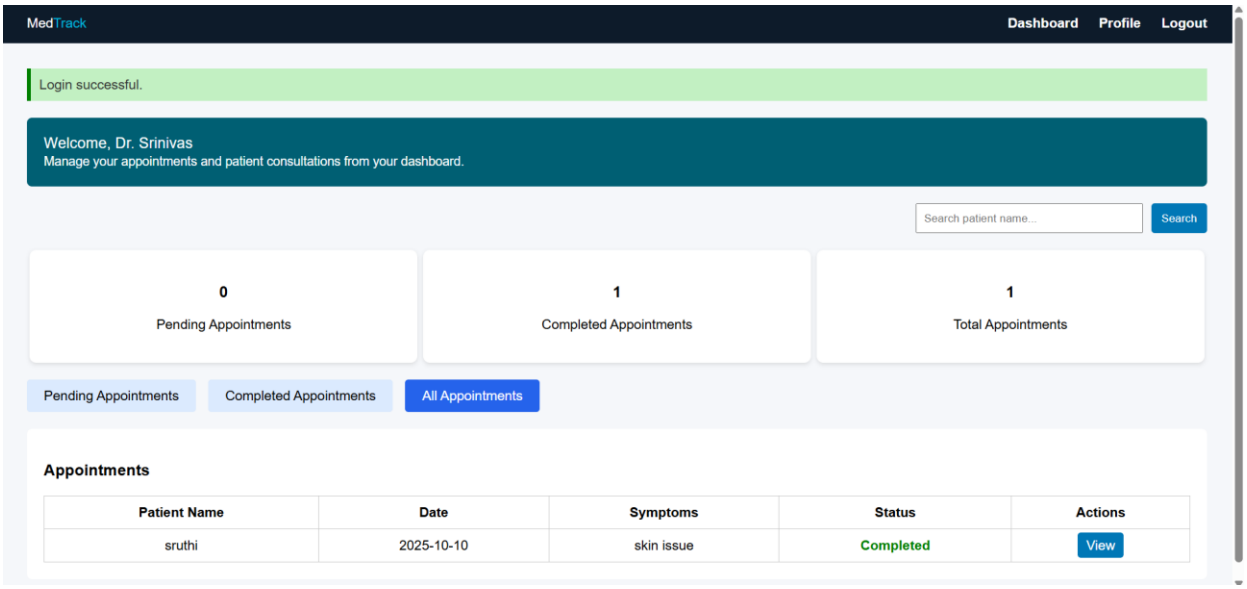
The **Doctor Dashboard** provides doctors with a comprehensive view of their upcoming appointments and patient details. It typically includes:

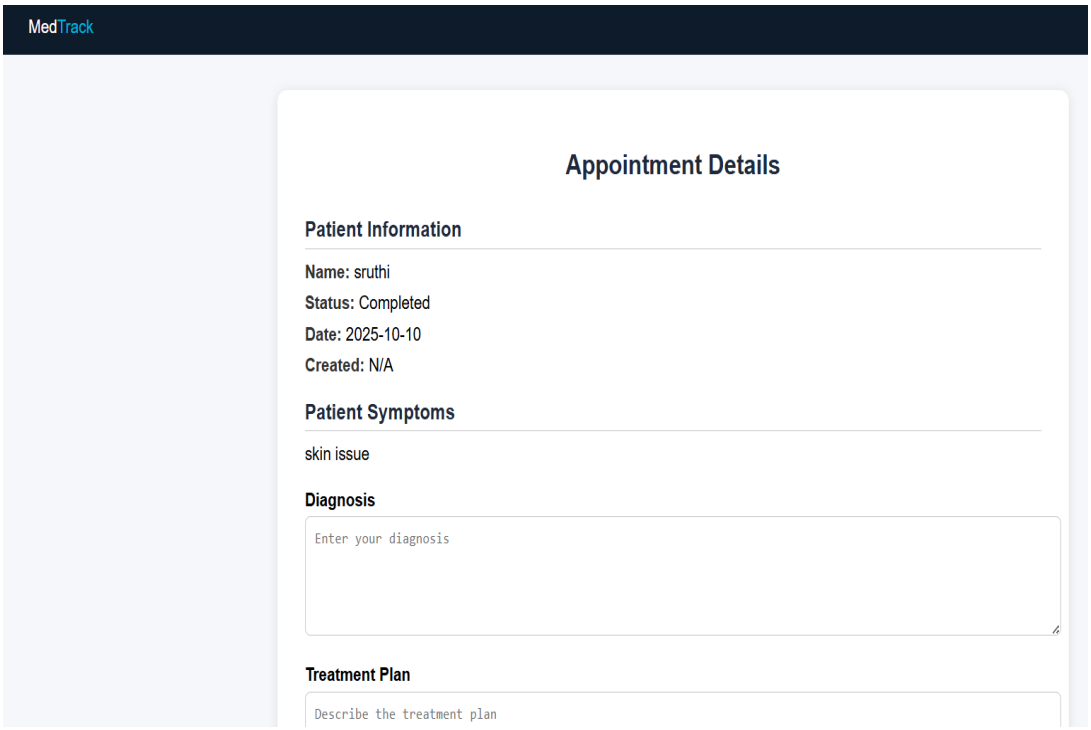
1. **Upcoming Appointments List**: A table or list showing patient names, appointment times, and appointment statuses (e.g., confirmed, pending).

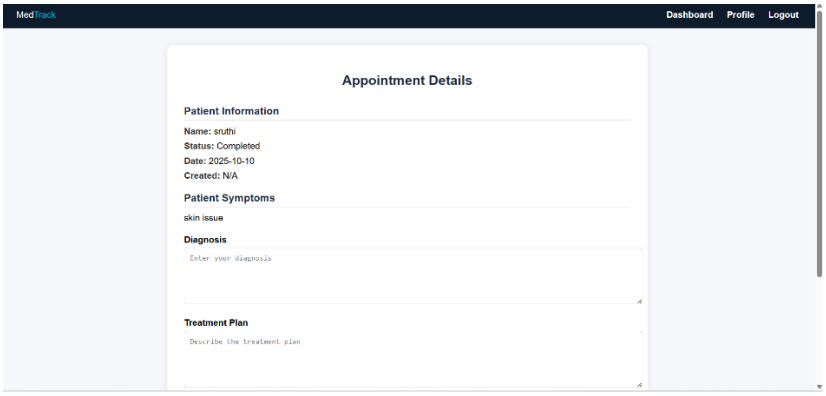
2. **Patient Details**: Quick access to each patient's medical history, contact information, and previous visit records.

3. **Appointment Actions**: Options to view, confirm, reschedule, 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.


MedTrack
Dashboard
Profile
Logout
Login successful.
Welcome, Dr. Srinivas
Manage your appointments and patient consultations from your dashboard.
Search patient name...
Search
0
Pending Appointments
1
Completed Appointments
1
Total Appointments
Pending Appointments
Completed Appointments
All Appointments
Appointments
Patient Name
Date
Symptoms
Status
Actions
sruthi
2025-10-10
skin issue
Completed
View


MedTrack
Appointment Details
Patient Information
Name: sruthi
Status: Completed
Date: 2025-10-10
Created: N/A
Patient Symptoms
skin issue
Diagnosis
Enter your diagnosis
Treatment Plan
Describe the treatment plan



Search Appointment:

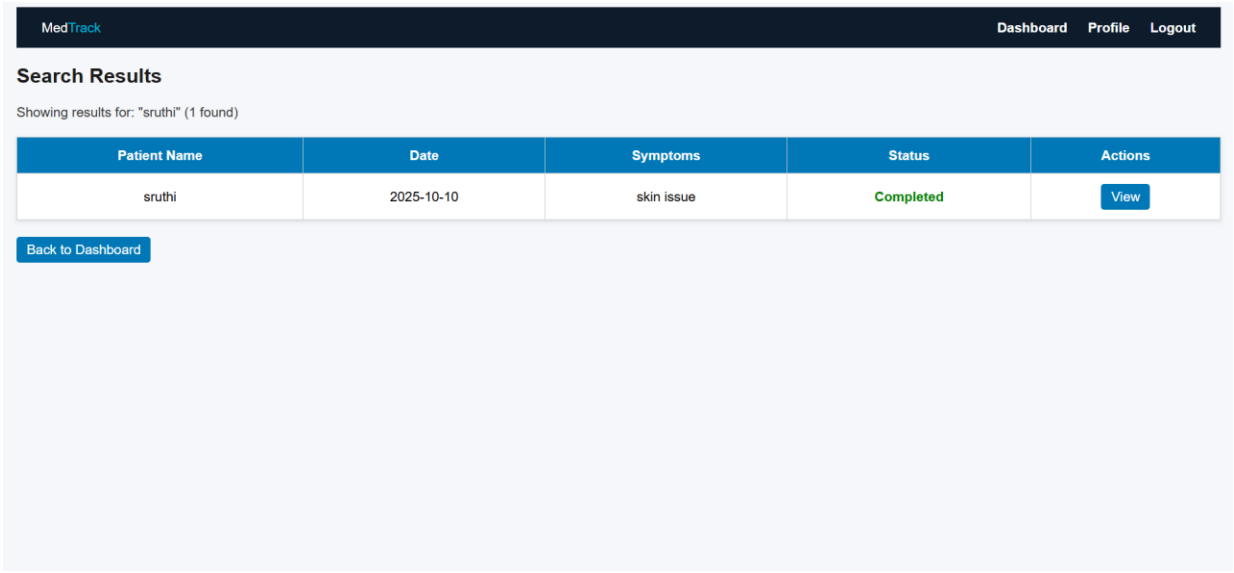
The Search Appointment feature in the Doctor Dashboard allows doctors to quickly find and manage specific patient appointments. It typically includes:

1. Search Bar: A field where doctors can enter a patient's name, appointment date, or other relevant details to filter appointments.

2. Appointment List: Displays the search results, including patient names, appointment times, and statuses, for easy access.

3. Action Options: Options to view, update, or cancel appointments directly from the search results.

This feature streamlines appointment management, making it easier for doctors to find and handle patient requests.



DynamoDB Database updations :

1. **Users table :**

In the Users Table of DynamoDB, the data structure is designed to store user-related information for both patients and doctors. Typical updates include:

- Add New Users: When a new patient or doctor registers, their details such as name, email, role (patient/doctor), contact info, and password hash are added to the table.
- Update User Info: If a user updates their profile (e.g., changing contact details), the corresponding record in the table is modified.
- Status Tracking: Track the status of user accounts (active, inactive) based on their activity or admin updates.

The Users Table serves as the central repository for all user data, enabling quick access and modification of details when necessary

2. **Appointment table :**

In the Appointment Table of DynamoDB, the data structure stores information related to patient appointments. Typical updates include:

- Add New Appointment: When a patient books an appointment, details such as patient ID, doctor ID, appointment date, time, and status (pending, confirmed, canceled) are stored.
- Update Appointment Status: As appointments are confirmed, rescheduled, or canceled, the status field in the table is updated accordingly.
- Appointment History: Historical data about completed appointments can also be stored to track past interactions between patients and doctors.

The Appointment Table allows for efficient management of appointments, ensuring accurate and up-to-date scheduling information for both doctors and patients

3. **Mail to the User:**

- A confirmation email is automatically sent to the user upon successful appointment booking. This mail includes details such as appointment date, time, doctor's name, and status (confirmed or pending).
- In case of rescheduling or cancellation, users are notified instantly via email to ensure they stay updated on any changes.
- This ensures timely communication and improves the overall user experience within the MedTrack system.

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.