MedTrack: A Cloud-Enabled Healthcare Management System

Project Description:

In today's fast-evolving healthcare landscape, efficient communication and coordination between doctors and patients are crucial. Med Track 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. Med Track 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 Med Track 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.

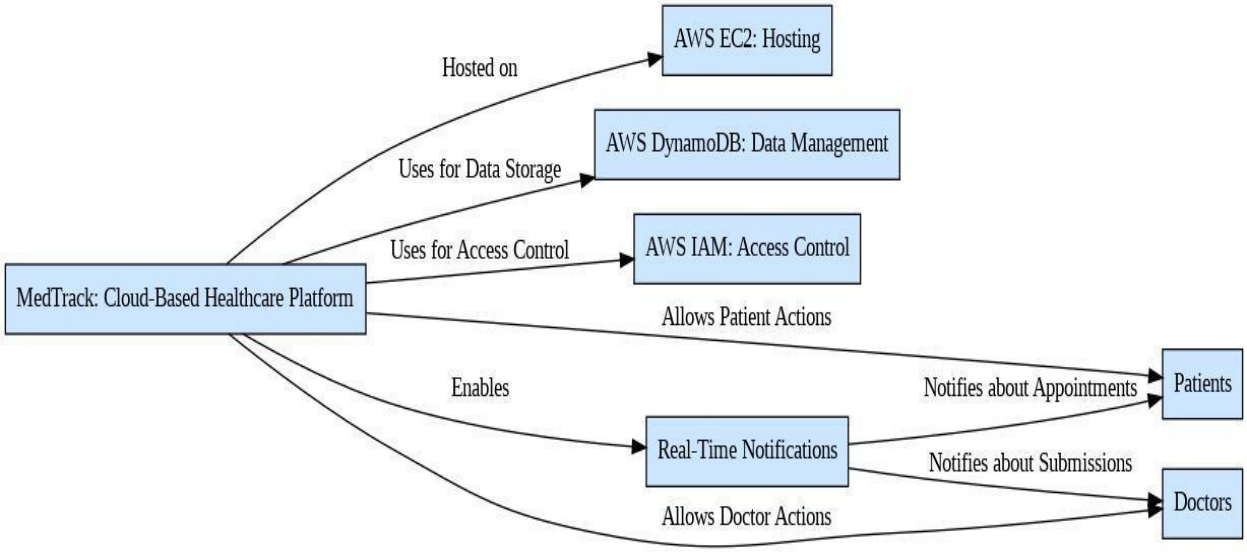
Scenario 2: Secure User Management with IAM

Med Track 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.

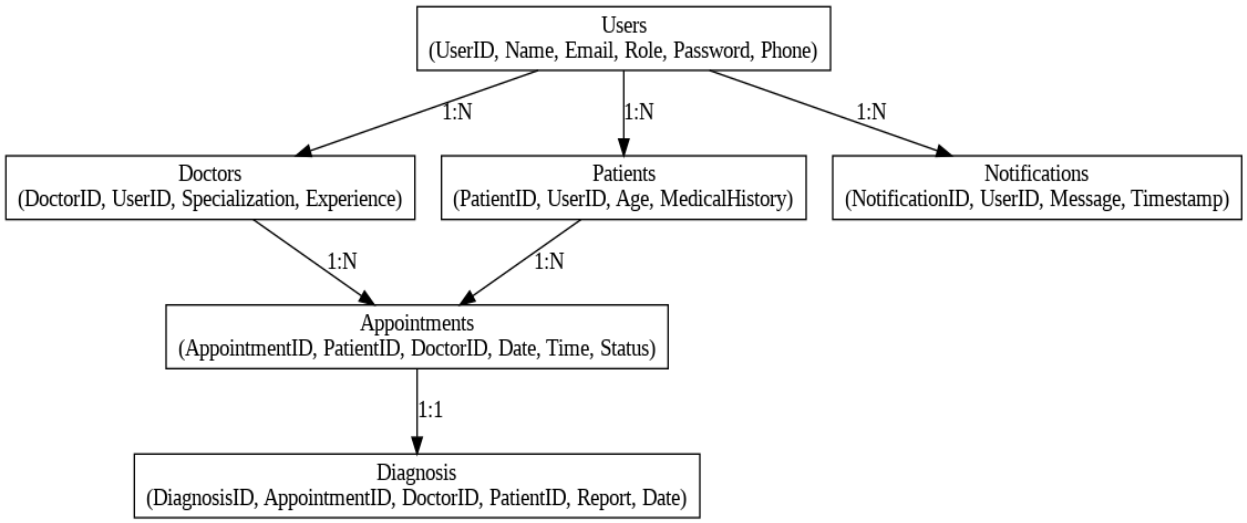
Scenario 3: Easy Access to Medical History and Resources

The Med Track 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. **Git Version Control**: Git Documentation

Project Workflow: Med Track

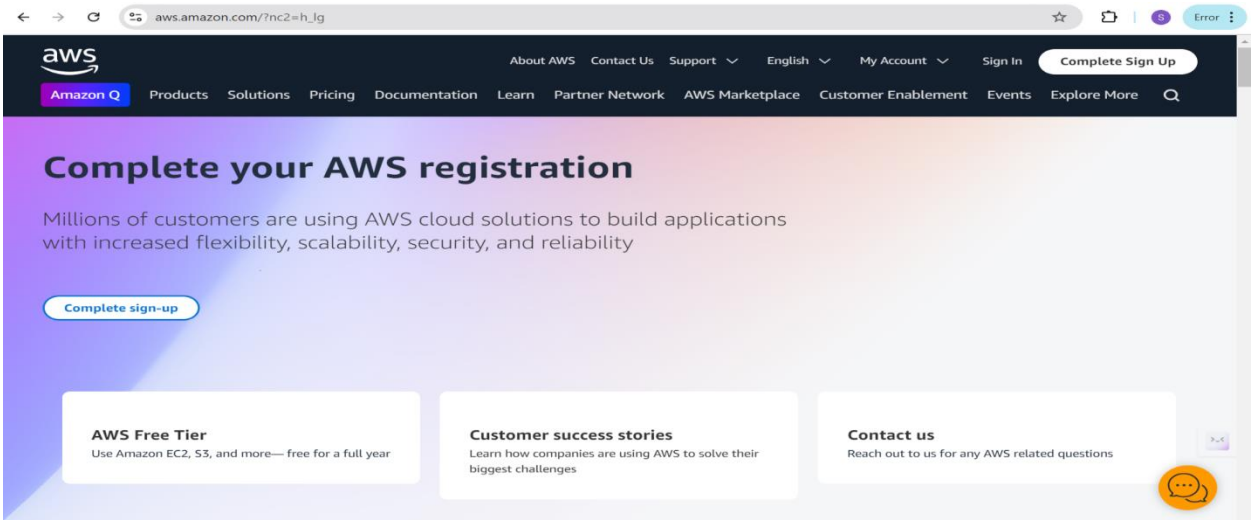
Project Flow

1. **AWS Account Setup and Login**
   - **Activity 1.1:** Set up an AWS account if not already done.
   - **Activity 1.2:** Log in to the AWS Management Console.
2. **DynamoDB Database Creation and Setup**
   - **Activity 2.1:** Create a DynamoDB Table.
   - **Activity 2.2:** Configure Attributes for User Data and Appointment Records.
3. **Backend Development and Application Setup**
   - **Activity 3.1:** Develop the Backend Using Flask.
   - **Activity 3.2:** Integrate AWS Services Using Boto3.
4. **IAM Role Setup**
   - **Activity 4.1:** Create IAM Roles for secure user access.
   - **Activity 4.2:** Attach Policies to ensure appropriate permissions.
5. **EC2 Instance Setup**
   - **Activity 5.1:** Launch an EC2 instance to host the Flask application.
   - **Activity 5.2:** Configure security groups for HTTP and SSH access.
6. **Testing and Deployment on EC2**
   - **Activity 6.1:** Upload Flask Files to the EC2 instance.
   - **Activity 6.2:** Run the Flask App to make it accessible.

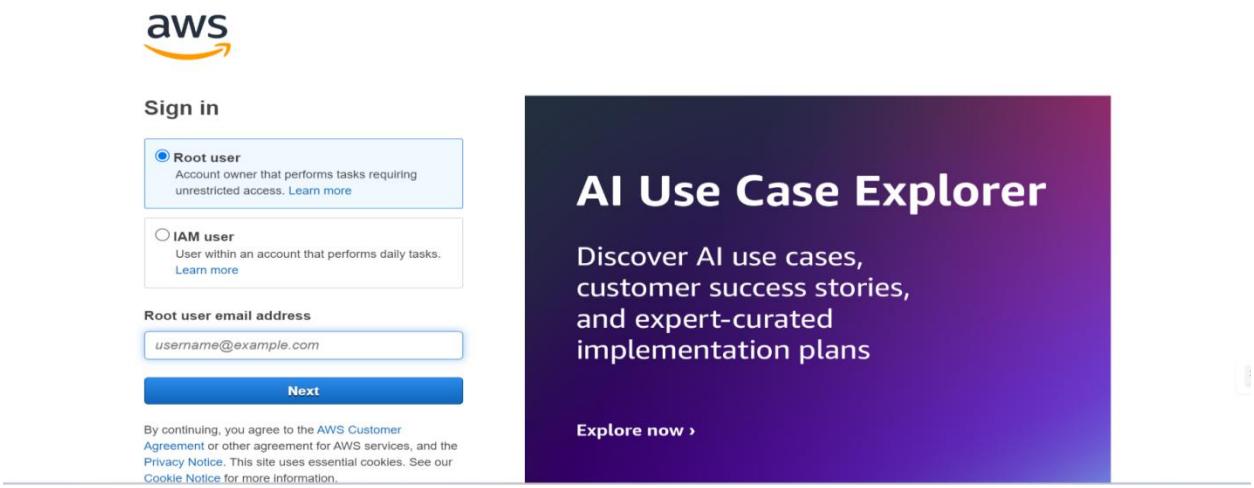
- **Activity 6.3:** Conduct functional testing to verify user registration, login, appointment booking, and data retrieval.

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.

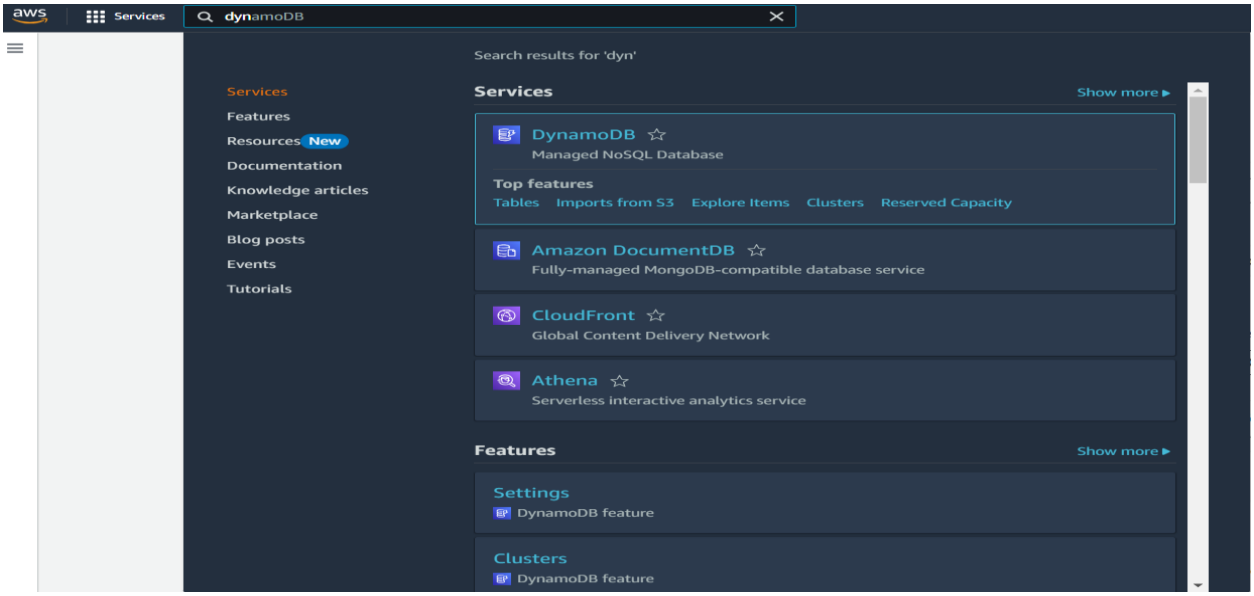


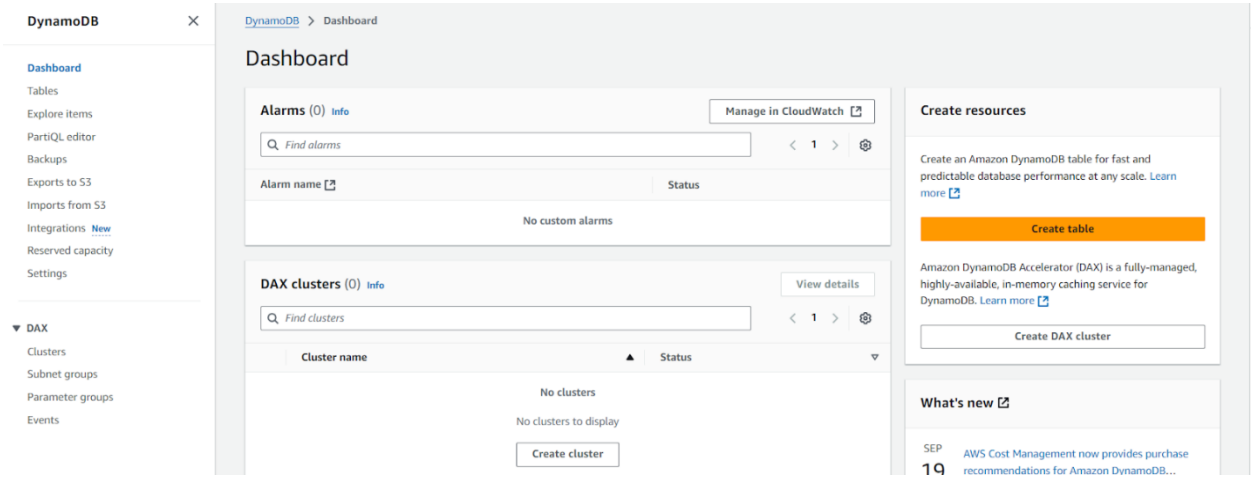
- **Activity 1.2: Log in to the AWS Management Console**
  - After setting up your account, log in to the AWS Management Console.

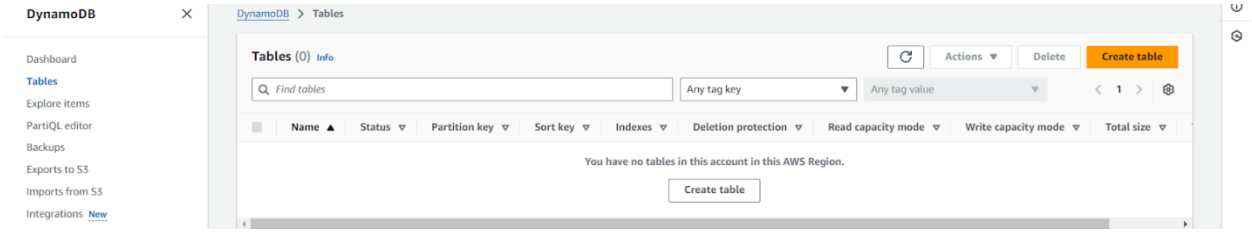


Milestone 2: DynamoDB Database Creation and Setup

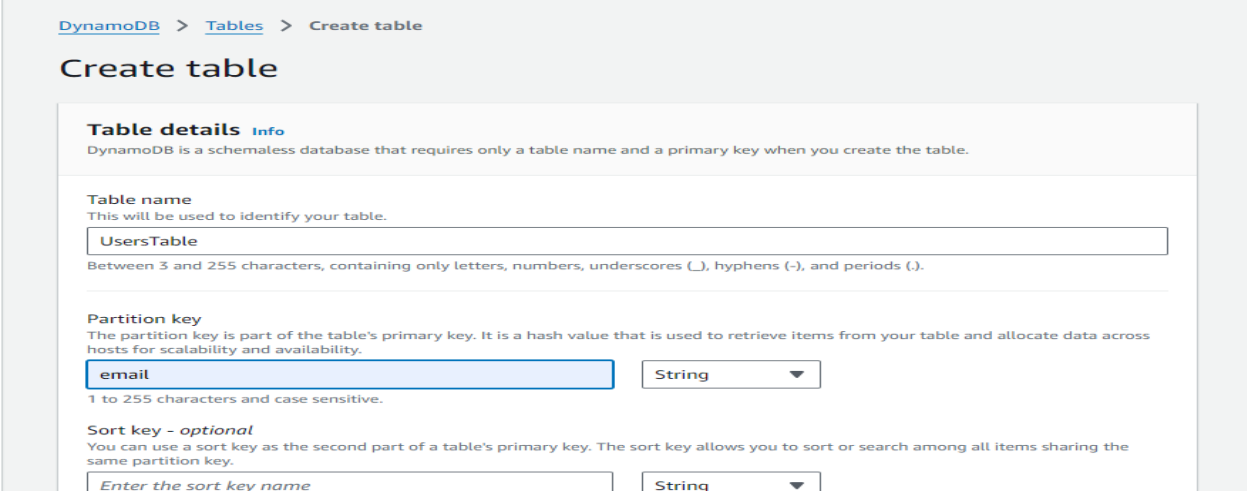
- **Activity 2.1: Navigate to the DynamoDB**
  - In the AWS Console, navigate to DynamoDB and click on create tables.

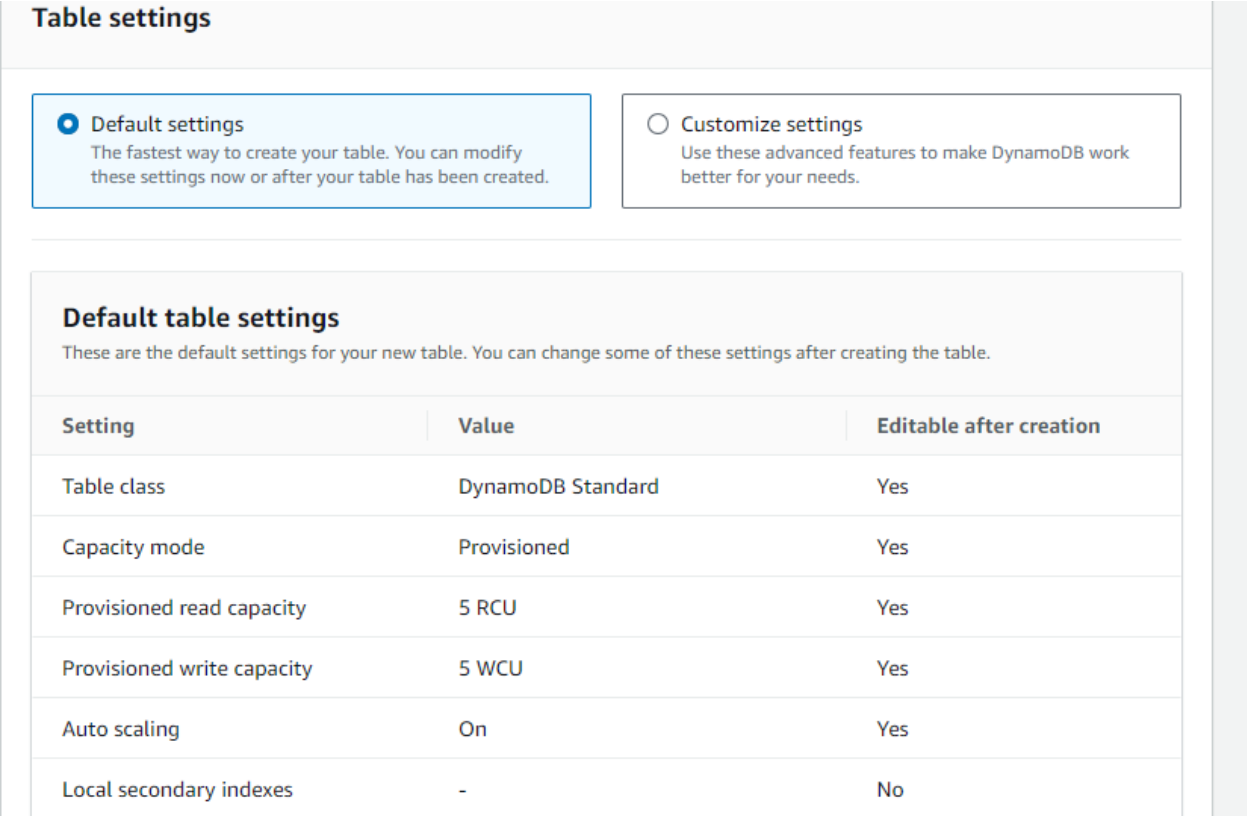


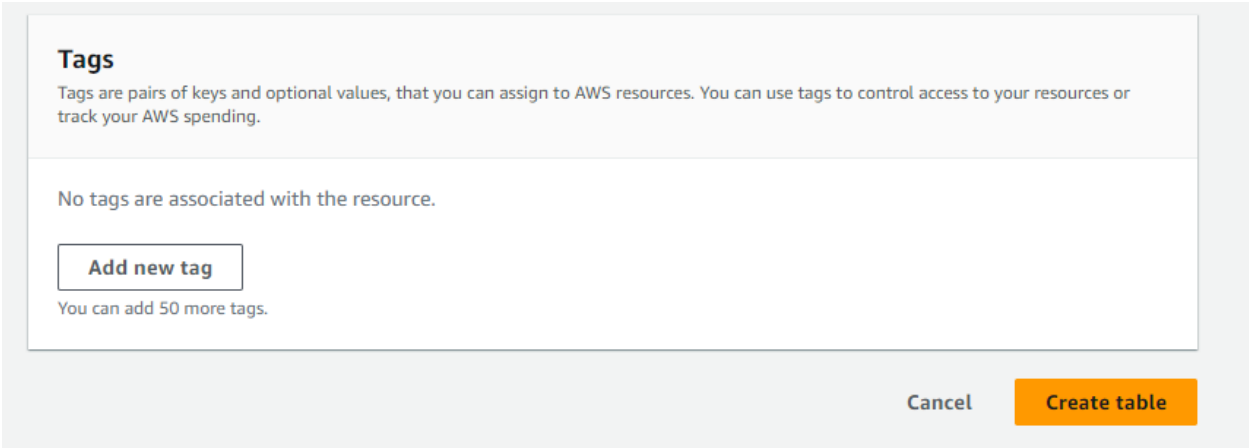




- **Activity 2.2: Create a DynamoDB table for storing registration details and book requests.**
    - Create Users Table with partition key “Email” with type String.







- Follow the same steps to create Appointments Table with appointment_id as the primary key.

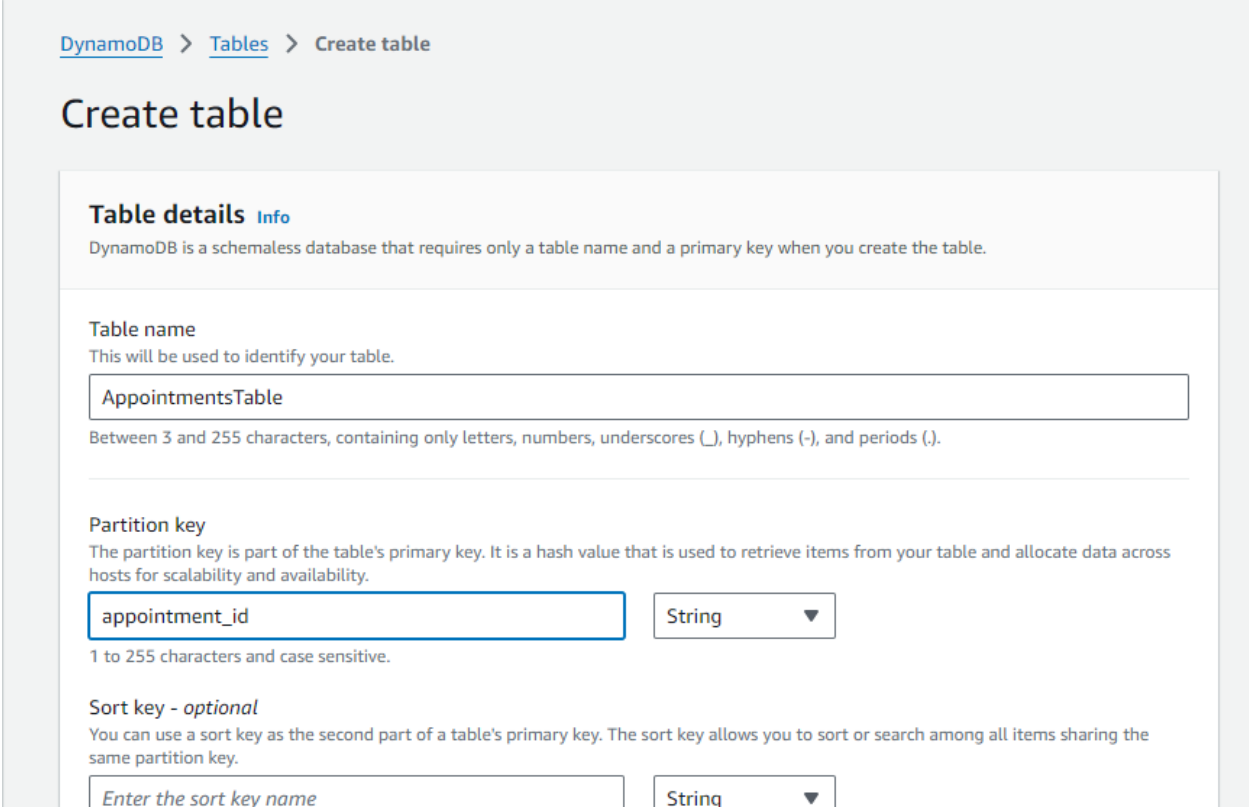


Table settings

- (●) **Default settings**
  The fastest way to create your table. You can modify these settings now or after your table has been created.
- ( ) **Customize settings**
  Use these advanced features to make DynamoDB work better for your needs.

Default table settings

These are the default settings for your new table. You can change some of these settings after creating the table.

Setting	Value	Editable after creation
Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No

Tags

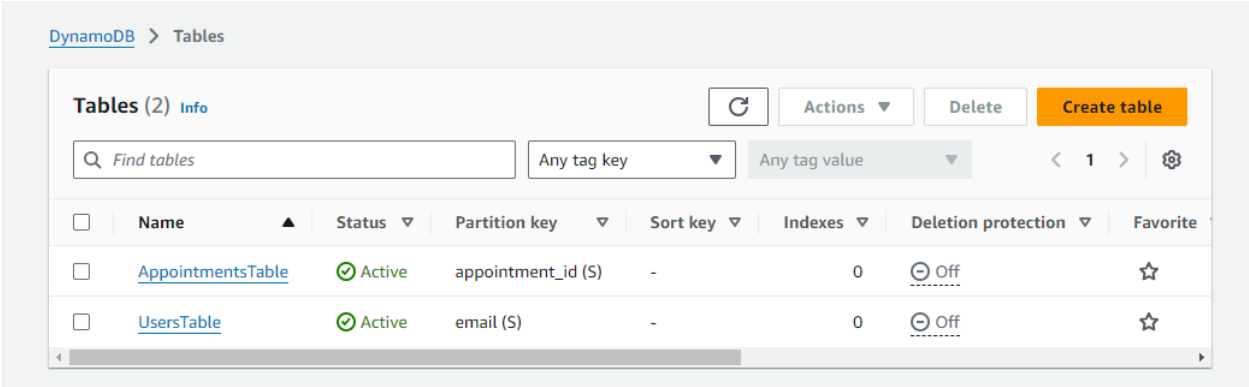
Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

[Add new tag]

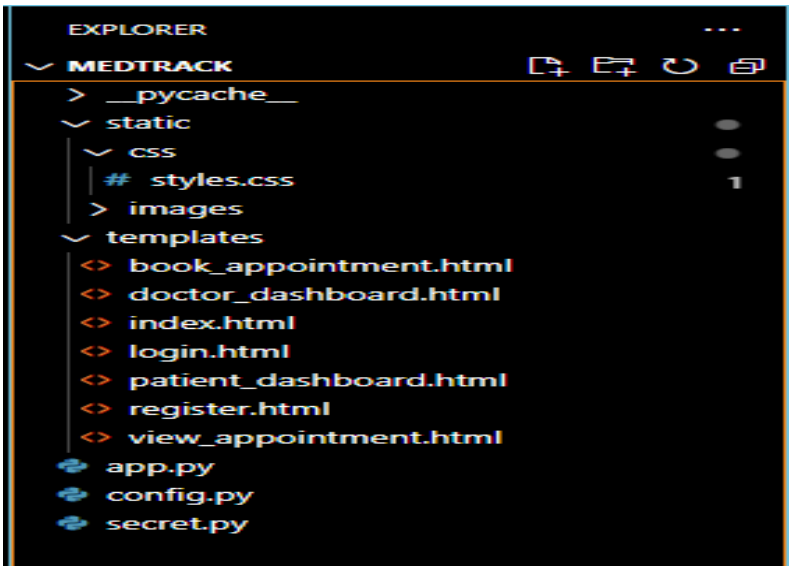
You can add 50 more tags.

[Cancel] [Create table]



Milestone 3: Backend Development and Application Setup

- **Activity 3.1: Develop the backend using Flask**
  - File Explorer Structure



Description of the code:

- **Flask App Initialization**

```
from flask import Flask, request, session, redirect, url_for, render_template, flash
import boto3
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime
import uuid
from config import Config  # Import Config class

```

```python
app = Flask(__name__)
app.config.from_object(Config)
app.secret_key = app.config['SECRET_KEY']

```

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

```python
# Initialize DynamoDB resource
dynamodb = boto3.resource('dynamodb',
                          region_name=app.config['AWS_REGION_NAME'],
                          aws_access_key_id=app.config['AWS_ACCESS_KEY_ID'],
                          aws_secret_access_key=app.config['AWS_SECRET_ACCESS_KEY'])

# Define DynamoDB tables
user_table = dynamodb.Table('UsersTable')
appointment_table = dynamodb.Table('AppointmentsTable')

```

Config.py:

```python
config.py > ...
class Config:
    AWS_ACCESS_KEY_ID = 'AKIA2########OKDF4C'
    AWS_SECRET_ACCESS_KEY = '1u+GQa################/uXAS5zOZD'
    AWS_REGION_NAME = 'ap-south-1'
    SECRET_KEY = '24194554c##################61b6d87eb4de5'

```

Description: The Config class contains configuration settings for the Med Track application, including AWS access credentials and the secret key (Flask) for session management. These settings enable secure access to AWS services like DynamoDB and EC2 while ensuring the application's overall security.

Secret.py:

```python
secret.py > ...
import os
import binascii

# Generate a random secret key
secret_key = binascii.hexlify(os.urandom(24)).decode()
print(secret_key)

```

Description: This code generates a random secret key for use in applications by creating a random sequence of 24 bytes using os.Urandom(24). The bytes are then converted to a hexadecimal string, ensuring a secure key suitable for cryptographic operations.

- **Routes for Web Pages**

1. Home page Routes:

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

```

Description: A route is defined for the home page ('/') that checks if the user is logged in using the is_logged_in() function. If the user is authenticated, they are redirected to the dashboard; otherwise, the index.html template is rendered for unauthenticated users.

2.Register Route:

```python
# Register User (Doctor/Patient)
@app.route('/register', methods=['GET', 'POST'])
def register():
    if is_logged_in():  # Check if already logged in
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
        password = generate_password_hash(request.form['password'])  # Hash password
        age = request.form['age']
        gender = request.form['gender']
        role = request.form['role']  # 'doctor' or 'patient'

        # Add user to DynamoDB
        user_table.put_item(
            Item={
                'email': email,
                'name': name,
                'password': password,  # Store hashed password
                'age': age,
                'gender': gender,
                'role': role,
                'specialization': request.form['specialization'] if role == 'doctor' else None,
            }
        )
        flash('Registration successful. Please log in.', 'success')
        return redirect(url_for('login'))
    return render_template('register.html')
```

Description: A route is defined for user registration ('/register') that handles both GET and POST requests. If the user is already logged in, they are redirected to the dashboard. For POST requests, the route collects form data (name, email, password, age, gender, role) and hashes the password for security. If the role is 'doctor,' a specialization is also collected. The user details are then stored in a DynamoDB table, and upon successful registration, the user is prompted to log in. For GET requests, the register.html template is rendered.

3.Register Route:

```python
# Login User (Doctor/Patient)
@app.route('/login', methods=['GET', 'POST'])
def login():
    if is_logged_in():  # If the user is already logged in, redirect to dashboard
        return redirect(url_for('dashboard'))

    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        role = request.form['role']  # Get the selected role (doctor or patient)

        # Validate user credentials
        user = user_table.get_item(Key={'email': email}).get('Item')

        if user:
            # Check password and role
            if check_password_hash(user['password'], password):  # Use check_password_hash to verify has
                if user['role'] == role:
                    session['email'] = email
                    session['role'] = role  # Store the role in the session
                    flash('Login successful.', 'success')
                    return redirect(url_for('dashboard'))
                else:
                    flash('Invalid role selected.', 'danger')
            else:
                flash('Invalid password.', 'danger')
        else:
            flash('Email not found.', 'danger')

    return render_template('login.html')
```

Description: A route for user login ('/login') handles GET and POST requests. On POST, it validates the email, password, and role by checking credentials stored in DynamoDB, and logs the user in if successful. If already logged in, the user is redirected to the dashboard; otherwise, login.html is rendered.

4.Dashboard Route:

```python
# Dashboard for both Doctors and Patients
@app.route('/dashboard')
def dashboard():
    if not is_logged_in():
        flash('Please log in to continue.', 'danger')
        return redirect(url_for('login'))

    role = session['role']
    email = session['email']

    if role == 'doctor':
        # Show doctor dashboard with list of appointments
        response = appointment_table.scan(
            FilterExpression="#doctor_email = :email",
            ExpressionAttributeNames={"#doctor_email": "doctor_email"},
            ExpressionAttributeValues={":email": email}
        )
        appointments = response['Items']
        return render_template('doctor_dashboard.html', appointments=appointments)

    elif role == 'patient':
        # Show patient dashboard with list of their appointments
        response = appointment_table.scan(
            FilterExpression="#patient_email = :email",
            ExpressionAttributeNames={"#patient_email": "patient_email"},
            ExpressionAttributeValues={":email": email}
        )
        appointments = response['Items']
        return render_template('patient_dashboard.html', appointments=appointments)
```

Description: The dashboard route checks if the user is logged in and redirects them to the login page if not. Depending on the user's role (doctor or patient), it retrieves a list of appointments from DynamoDB using the user's email and renders either the doctor_dashboard.html or patient_dashboard.html template with the relevant appointment data.

5.Book Appointment Route

```python
# Book an Appointment (Patient)
@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'))

    if request.method == 'POST':
        doctor_email = request.form['doctor_email']
        symptoms = request.form['symptoms']
        patient_email = session['email']

        # Create a new appointment
        appointment_id = str(uuid.uuid4())
        appointment_table.put_item(
            Item={
                'appointment_id': appointment_id,
                'doctor_email': doctor_email,
                'patient_email': patient_email,
                'symptoms': symptoms,
                'status': 'pending',
                'appointment_date': str(datetime.now()),
            }
        )
        flash('Appointment booked successfully.', 'success')
        return redirect(url_for('dashboard'))
```

```python
    # Get list of doctors for selection
    response = user_table.scan(
        FilterExpression="#role = :role",
        ExpressionAttributeNames={"#role": "role"},
        ExpressionAttributeValues={":role": 'doctor'}
    )
    doctors = response['Items']
    return render_template('book_appointment.html', doctors=doctors)

```

Description: This route allows patients to book an appointment. If the user is logged in as a patient, they can submit a form with a doctor's email and symptoms. The appointment is then created with a unique ID and saved to DynamoDB. For GET requests, it retrieves a list of available doctors from DynamoDB and renders the book_appointment.html template for selection.

6.View_appointments Route:

```python
# View Appointment (Doctor)
@app.route('/view_appointment/<appointment_id>', methods=['GET', 'POST'])
def view_appointment(appointment_id):
    if not is_logged_in() or session['role'] != 'doctor':
        flash('Only doctors can view appointments.', 'danger')
        return redirect(url_for('login'))

    # Fetch appointment details
    response = appointment_table.get_item(Key={'appointment_id': appointment_id})
    appointment = response.get('Item')

    if request.method == 'POST':
        diagnosis = request.form['diagnosis']
        treatment_plan = request.form['treatment_plan']
        prescription = request.form['prescription']

```

Description: This route allows doctors to view and update appointment details based on the provided appointment_id. If the user is logged in as a doctor, they can retrieve appointment information from DynamoDB and, via POST, submit a diagnosis, treatment plan, and prescription, updating the appointment status to "completed." For GET requests, the appointment data is displayed using the view_appointment.html template.

7.Submit_diagnosis Route:

```python
# Submit Diagnosis (Doctor)
@app.route('/submit_diagnosis/<appointment_id>', methods=['POST'])
def submit_diagnosis(appointment_id):
    diagnosis = request.form['diagnosis']
    treatment_plan = request.form['treatment_plan']
    prescription = request.form['prescription']

    # Update the appointment in the database
    appointment_table.update_item(
        Key={'appointment_id': appointment_id},
        UpdateExpression="SET diagnosis = :d, treatment_plan = :t, prescription = :p, #status = :s",
        ExpressionAttributeNames={"#status": "status"},
        ExpressionAttributeValues={
            ':d': diagnosis,
            ':t': treatment_plan,
            ':p': prescription,
            ':s': 'completed'
        }
    )

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

```

Description: This route allows doctors to submit a diagnosis, treatment plan, and prescription for a specific appointment via POST. The data is updated in DynamoDB by setting the diagnosis, treatment plan, prescription, and changing the appointment status to "completed." After the update, the doctor is redirected to the dashboard with a success message.

8.Logout Route

```
# Logout User
@app.route('/logout', methods=['GET', 'POST'])
def logout():
    if request.method == 'POST':  # Ensure it's a POST request for logout
        session.pop('email', None)
        session.pop('role', None)
        flash('You have been logged out.', 'success')
    return redirect(url_for('index'))
```

Description: This route handles user logout functionality by removing the user's email and role from the session upon receiving a POST request, effectively logging them out. Afterward, the user is redirected to the index page with a flash message indicating that they have been successfully logged out.

Deployment Code:

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

Description: This code snippet serves as the main entry point for the Flask application. When the script is executed directly, it starts the Flask development server in debug mode, allowing for live reloading and detailed error messages, which is useful for development and testing.

Milestone 4: IAM Role Setup

- **Activity 4.1: Create IAM Role.**
    - In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.
    - (Flask)

IAM > Users > Create user

Step 1
Specify user details

Step 2
Set permissions

Step 3
Review and create

Specify user details

User details

User name

med-user

The user name can have up to 64 characters. Valid characters: A-Z, a-z, 0-9, and + = , . @ _ - (hyphen)

☐ Provide user access to the AWS Management Console - *optional*
If you're providing console access to a person, it's a best practice to manage their access in IAM Identity Center.

If you are creating programmatic access through access keys or service-specific credentials for AWS CodeCommit or Amazon Keyspaces, you can generate them after you create this IAM user. Learn more

Cancel | Next

Step 2: Add permissions

Edit

Permissions policy summary

Policy name	Type	Attached as
AmazonDynamoDBFullAccess	AWS managed	Permissions policy

Step 3: Add tags

Add tags - *optional* Info

Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.

No tags associated with the resource.

Add new tag

You can add up to 50 more tags.

Cancel | Previous | Create role

IAM > Users

Users (1/1) Info

An IAM user is an identity with long-term credentials that is used to interact with AWS in an account.

Delete | Create user

Search

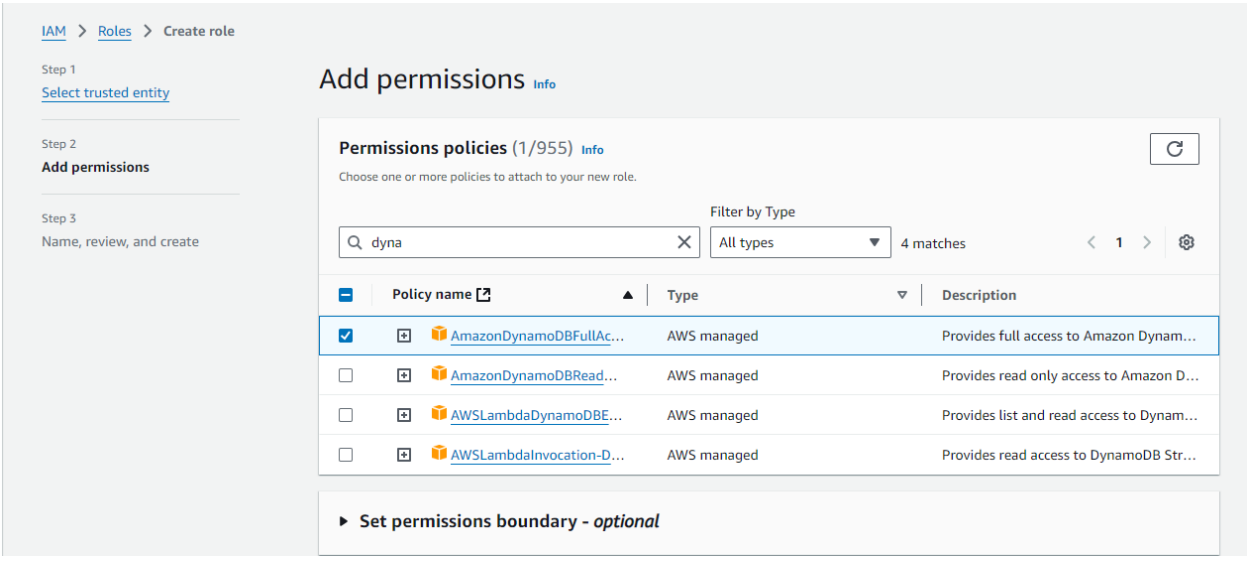
1

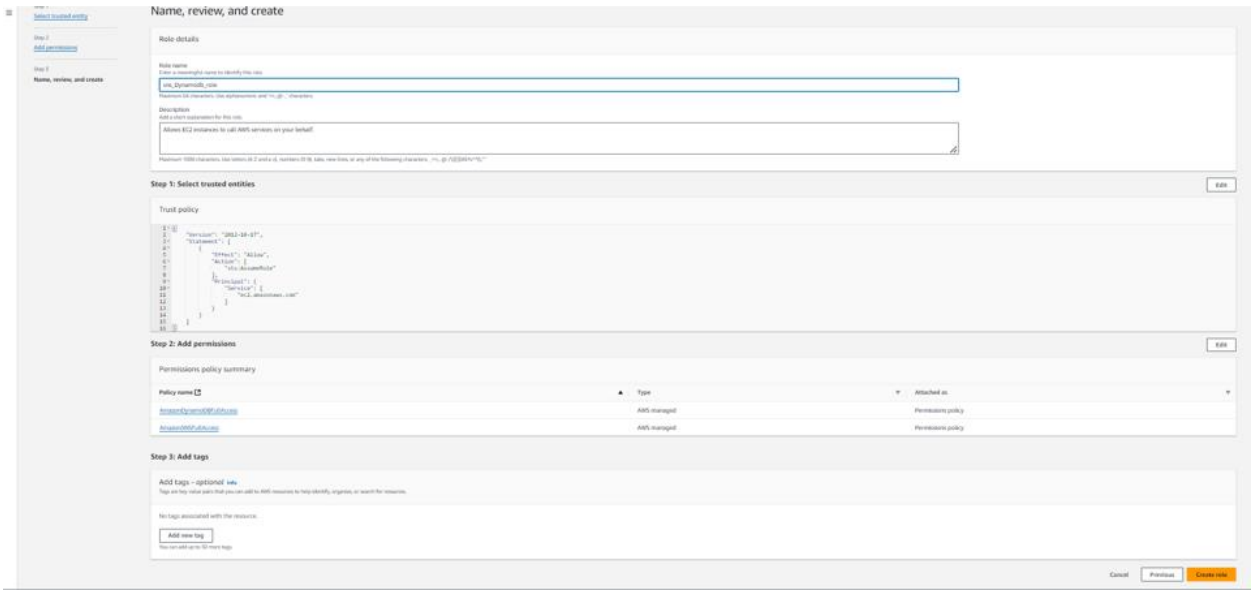
User name	Path	Groups	Last activity	MFA	Password age
med-user	/	0	10 hours ago	-	-

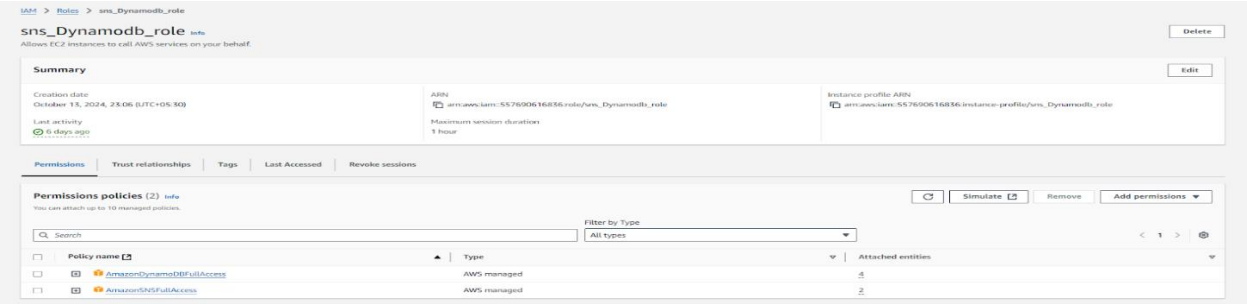
- **Activity 4.2: Attach Policies.**

Attach the following policies to the role:

- Amazon DynamoDB Full Access: Allows EC2 to perform read/write operations on DynamoDB.



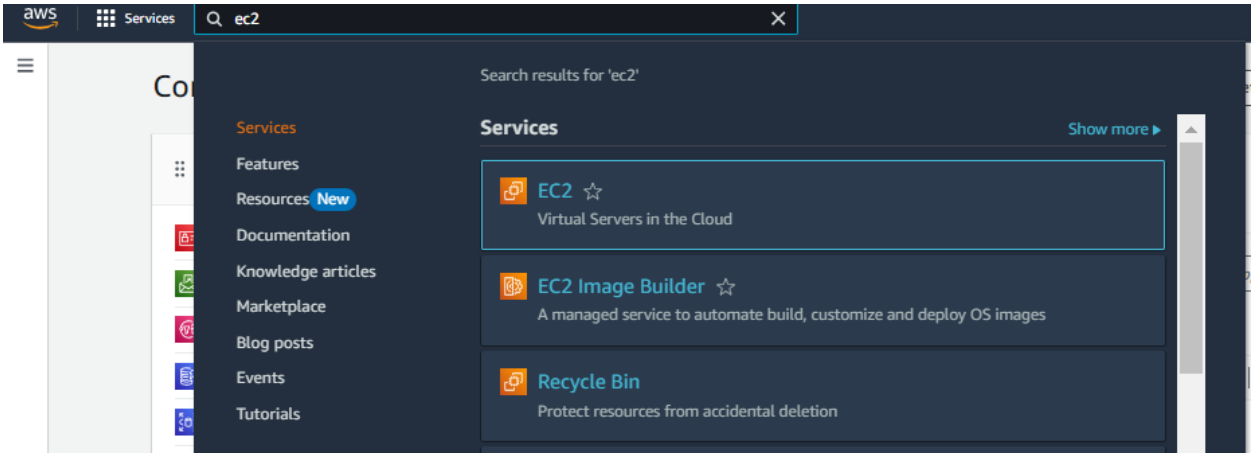




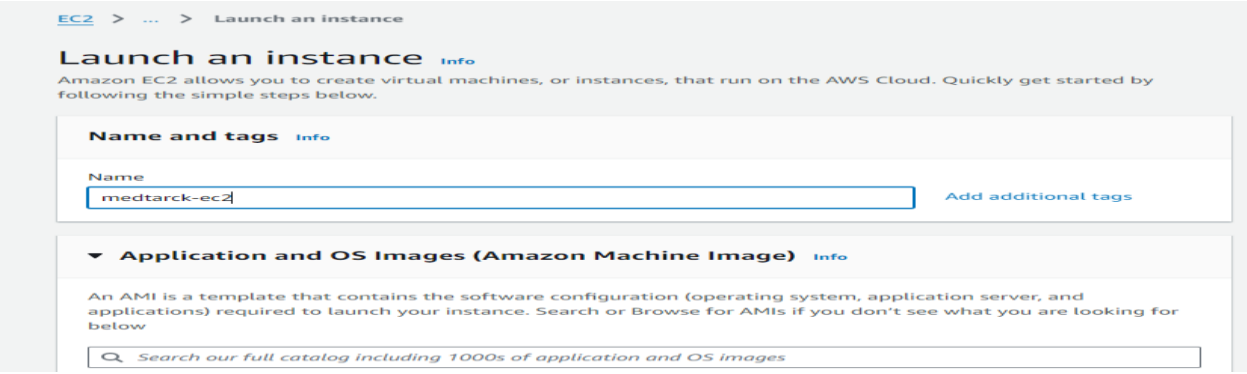
Milestone 5: EC2 Instance Setup

- Choose a Linux-based EC2 instance from the AWS Console to host the Med track application.

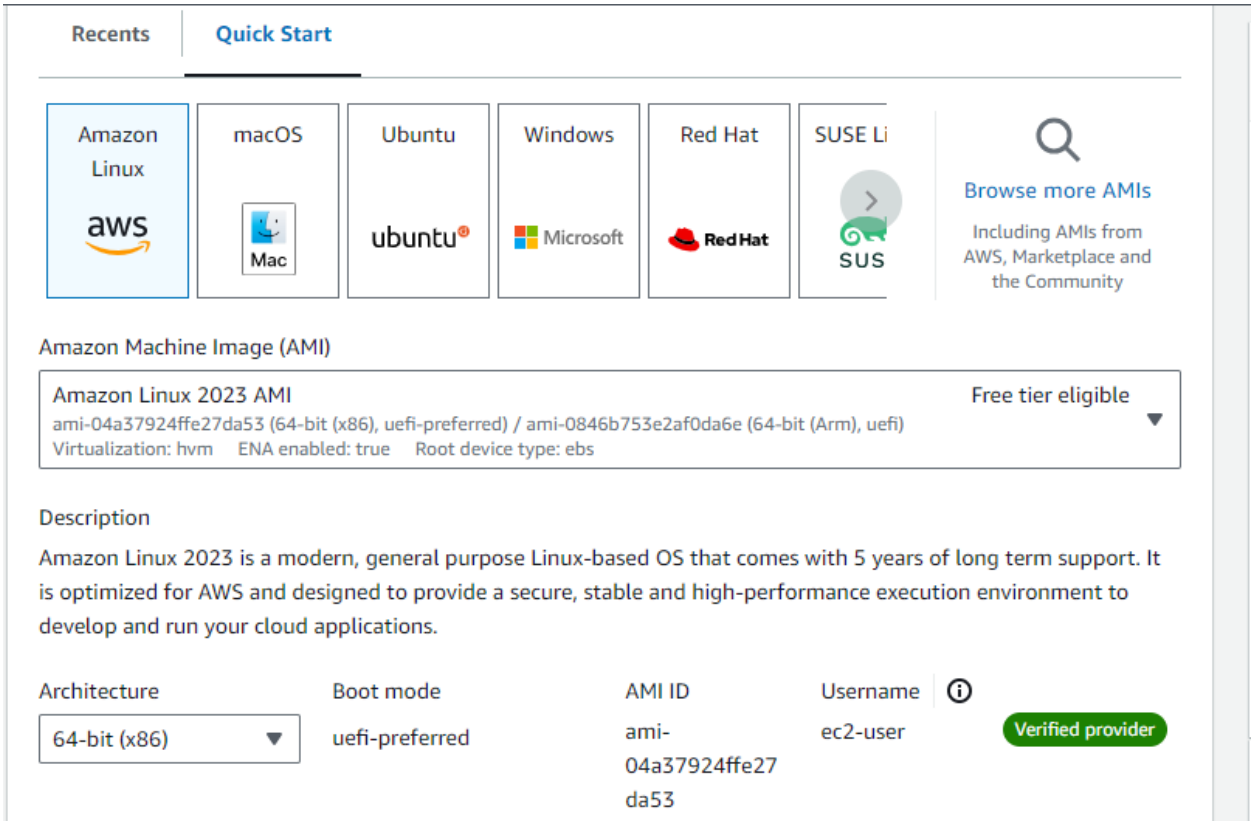
- **Activity 5.1: Launch an EC2 instance to host the Flask application.**
- **Launch EC2 Instance**
  - In the AWS Console, navigate to EC2 and launch a new instance.



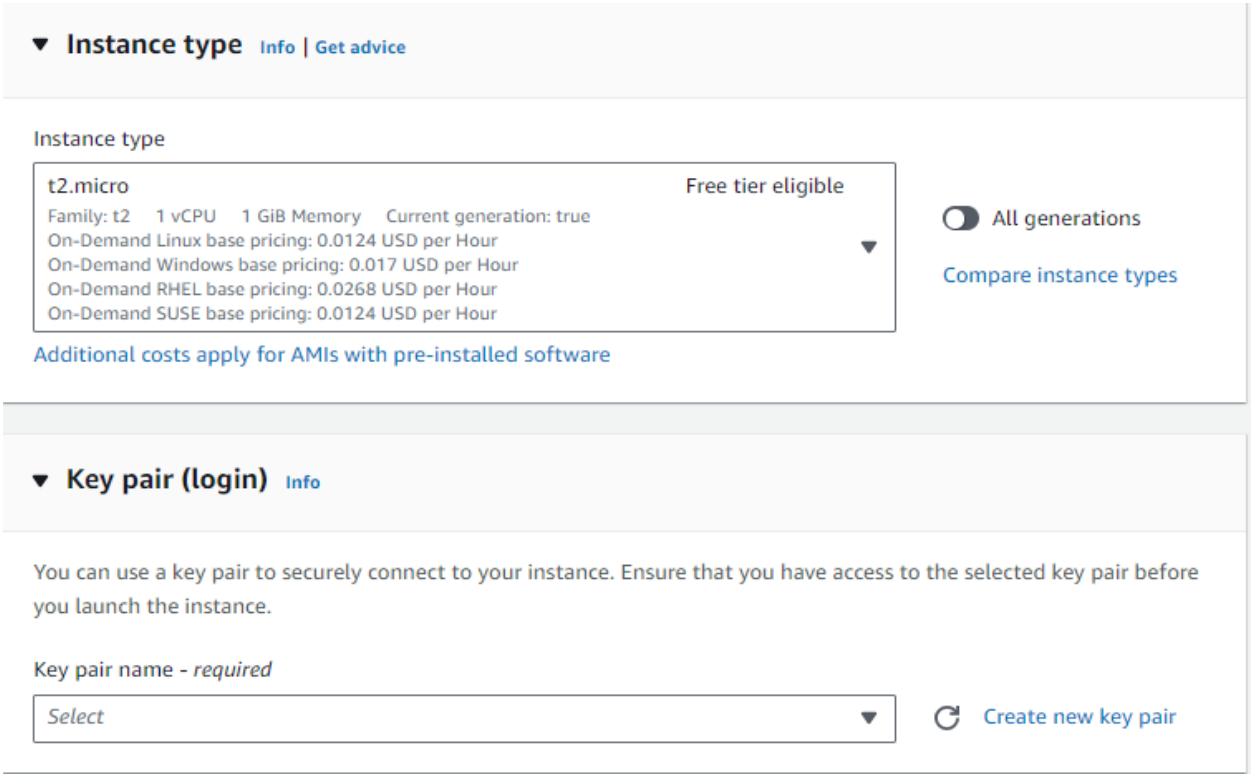
- Click on Launch instance to launch EC2 instance

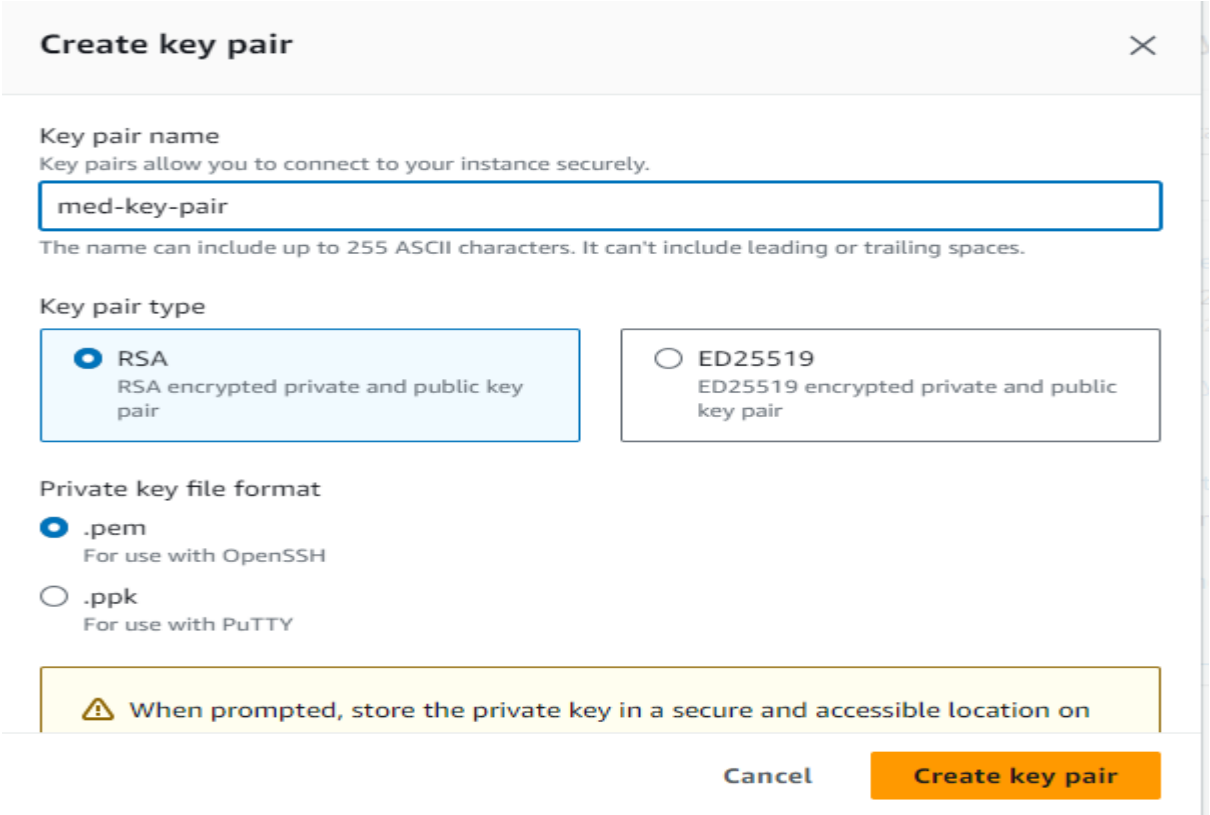


- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

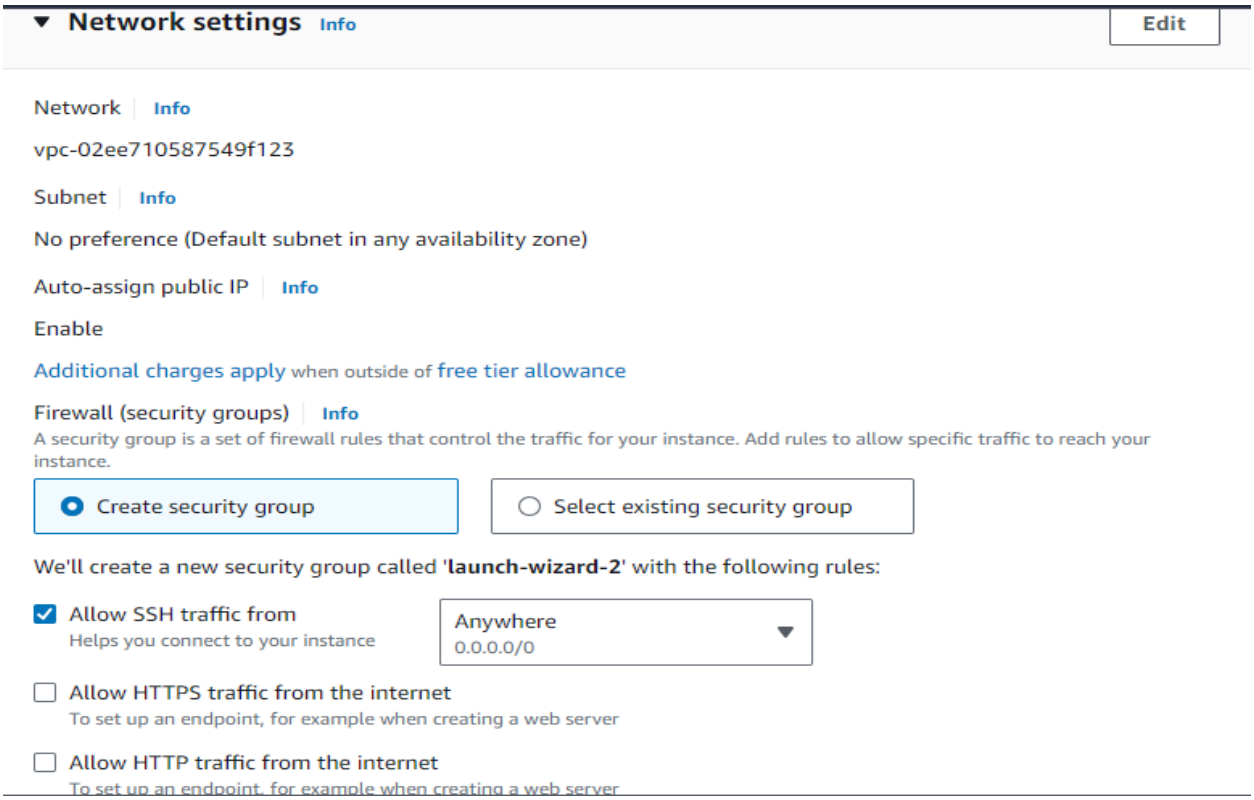


- Create and download the key pair for Server access.





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



Instances (1/2) Info

Last updated less than a minute ago | Connect | Instance state | Actions | Launch instances

Find Instance by attribute or tag (case-sensitive) | All states

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availabili
med-ec2	i-05d97fea7203376de	Running	t2.micro	...	View alarms +	ap-south-
medtarck-ec2	i-0cbd6b6ccb2afc1f4	Running	t2.micro	Initializing	View alarms +	ap-south-

Setting up Inbound and Outbound rules

i-0cbd6b6ccb2afc1f4 (medtarck-ec2)

Details | Status and alarms | Monitoring | Security | Networking | Storage | Tags

Security details

IAM Role
–

Owner ID
741448921864

Launch time
Wed Oct 23 2024 11:23:53 GMT+0530 (India Standard Time)

i-0f6f060a65171c78a (quiz-app)

Inbound rules

Filter rules

Name	Security group rule ID	Port range	Protocol	Source
–	sgr-035f84939e1b2c81b	80	TCP	0.0.0.0/0
–	sgr-0a65c1024fe8d85fe	3306	TCP	0.0.0.0/0
–	sgr-0cde2a0a45ccc0352	22	TCP	0.0.0.0/0
–	sgr-0766684fdd95a7fce	443	TCP	0.0.0.0/0

- Add Type : HTTP > Source : Anywhere
- Add Type : HTTPS > Source : Anywhere

Outbound rules

Filter rules

Name	Security group rule ID	Port range	Protocol	Destination
–	sgr-0aec66d8309fa06b1	All	All	0.0.0.0/0

Milestone 6: Testing and Deployment

- **Activity 6.1: Deploy to EC2**

1. Connect EC2 terminal.
2. Set up any necessary environment variables, including database connection strings.
3. Configure the web server to serve your application.
4. Start your application and ensure it's accessible via the EC2 instance's public IP or domain.
5. Run the below commands on ec2 terminal
6. sudo yum update -y
7. sudo yum install python3 -y
8. sudo pip3 install virtualenv
9. python3 -m venv venv
10. source venv/bin/activate
11. pip install flask

- **Functional Testing**
  - Test the app.py application for functionality, including database interactions and frontend features.
  - Run the Flask app **python3 app.py**
  - It will give you the link

Access the website through:

Access the website through Public IPs:

http:13.164.83.44:5000

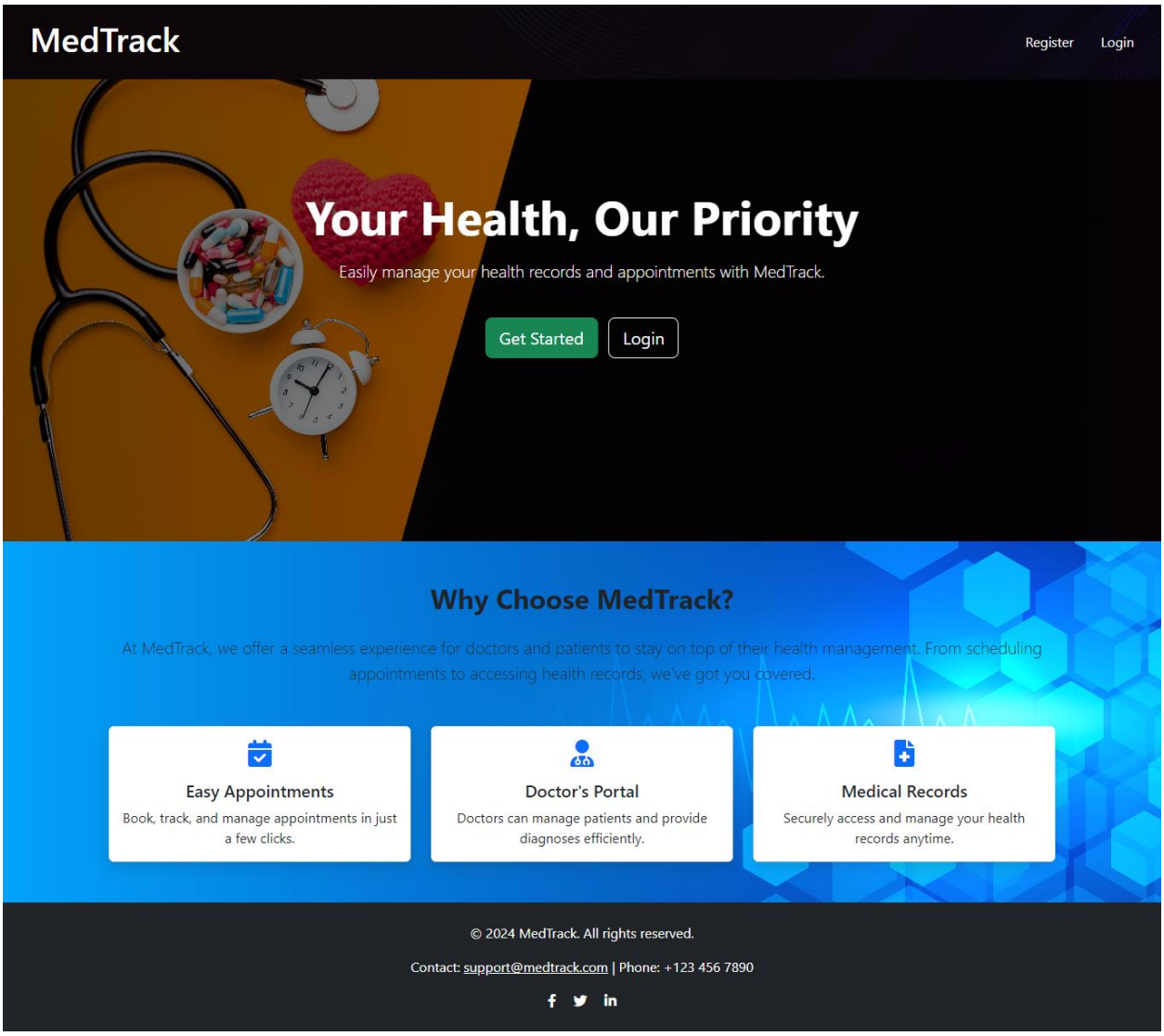
- **Activity 6.2: Deployment**
  - Deploy the application in a production environment, ensuring high availability and performance.

Click on the link above and it will take you to the webpage:

Activity:6.3

- Conduct functional testing to verify user registration, login, (doctor, patient) dashboard, view appointments, book appointments,

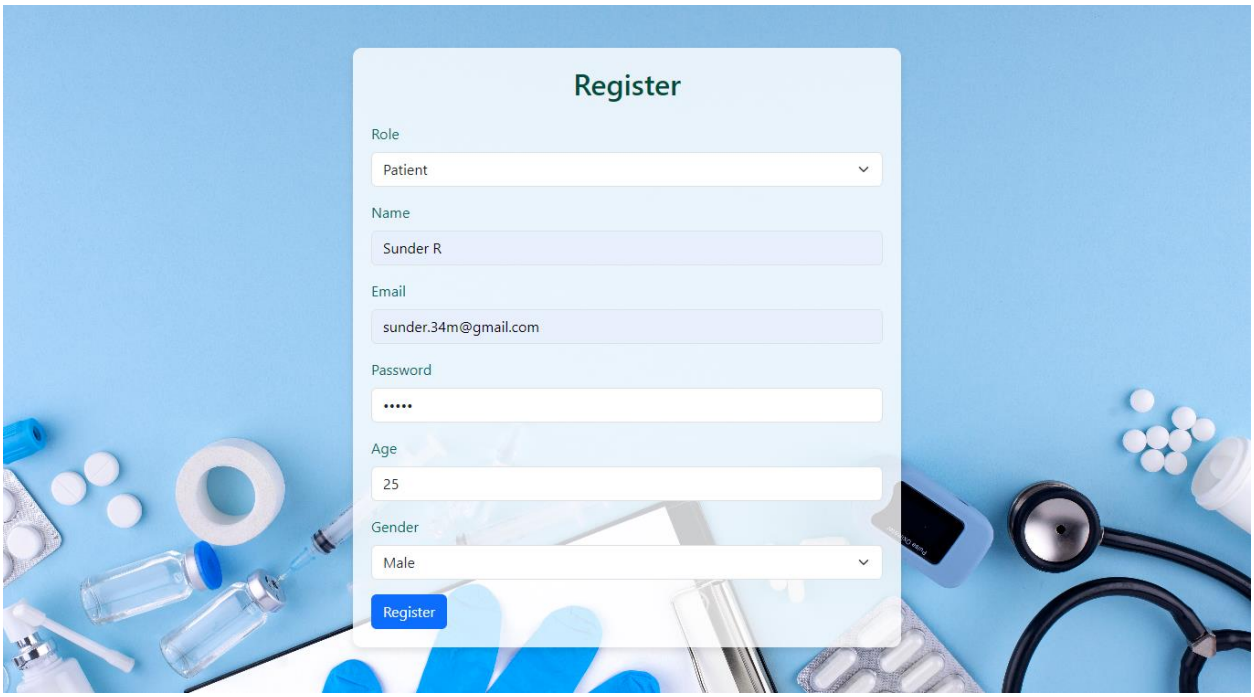
Home Page:



Description:

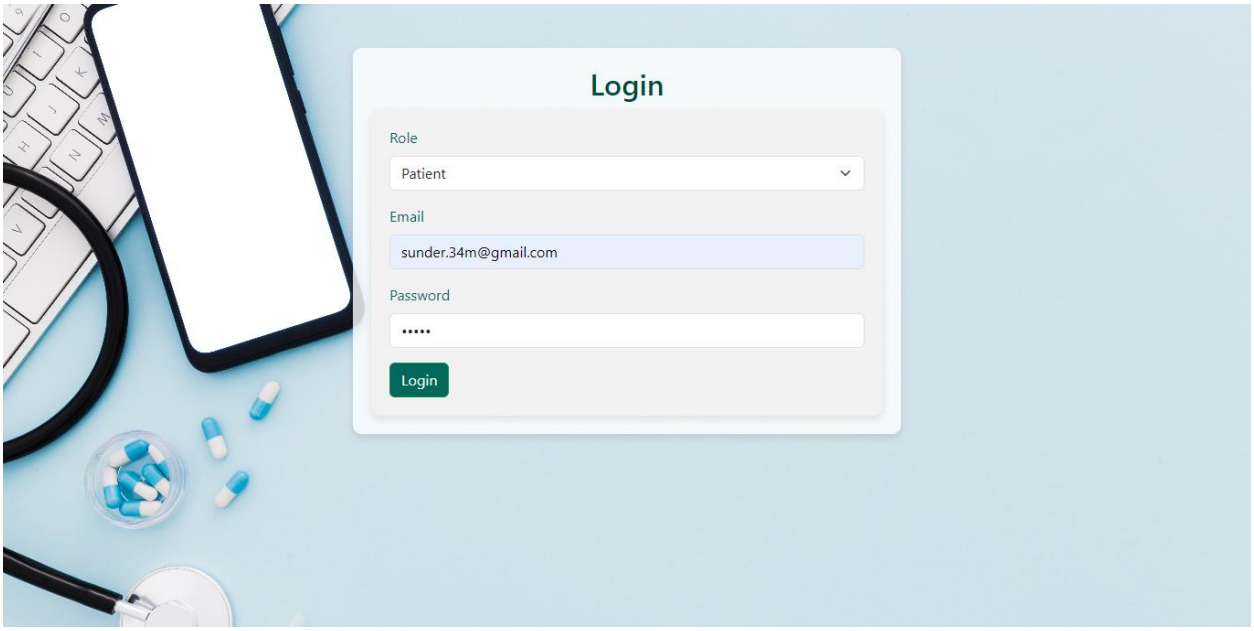
homepage for Med Track, featuring a navigation bar with options for user registration and login. The page includes a hero section promoting the platform's main purpose—helping users manage their health records and appointments—along with an about section detailing key features like easy appointments, a doctor's portal, and secure medical records management.

Register Page (Patient):



Patient Registration: This registration page allows patients to sign up by entering personal details such as name, email, age, and gender. Once registered, patients can access Med Track to manage their appointments and medical records efficiently.

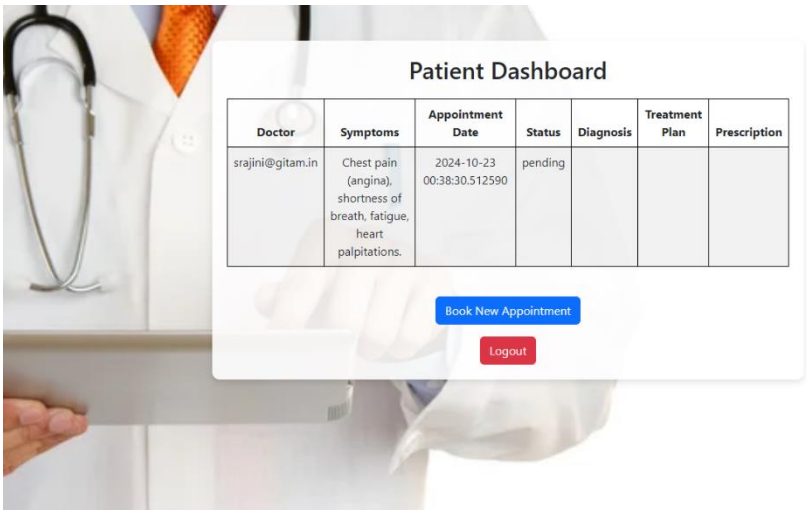
Login Page:



Patient Login:

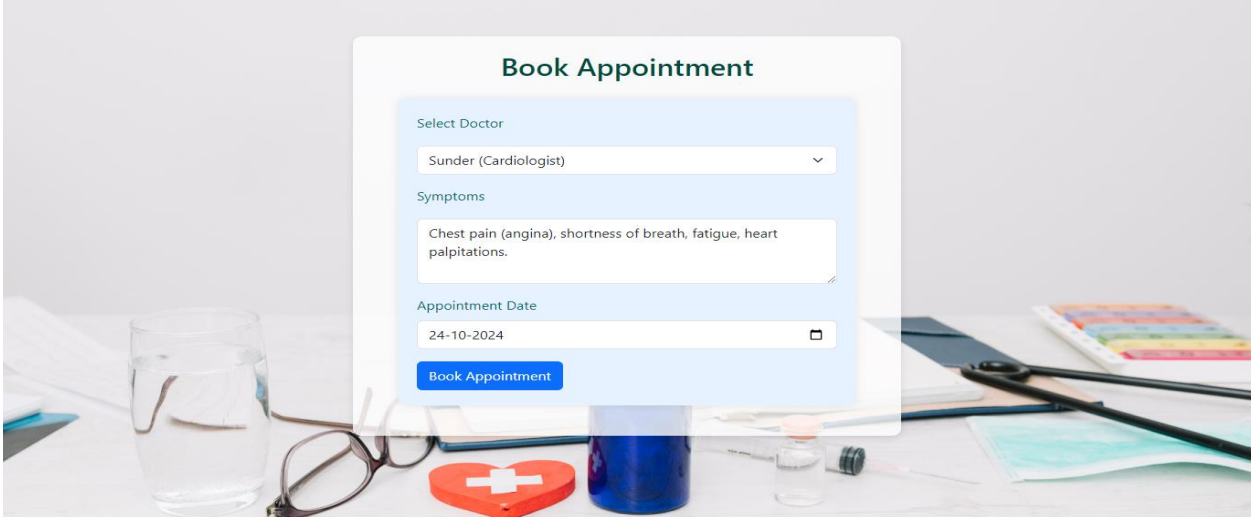
This login page allows patients to securely sign in by selecting their role and entering their email and password. Once authenticated, patients can access their personalized dashboard to manage appointments and view medical records.

Patient dashboard page:



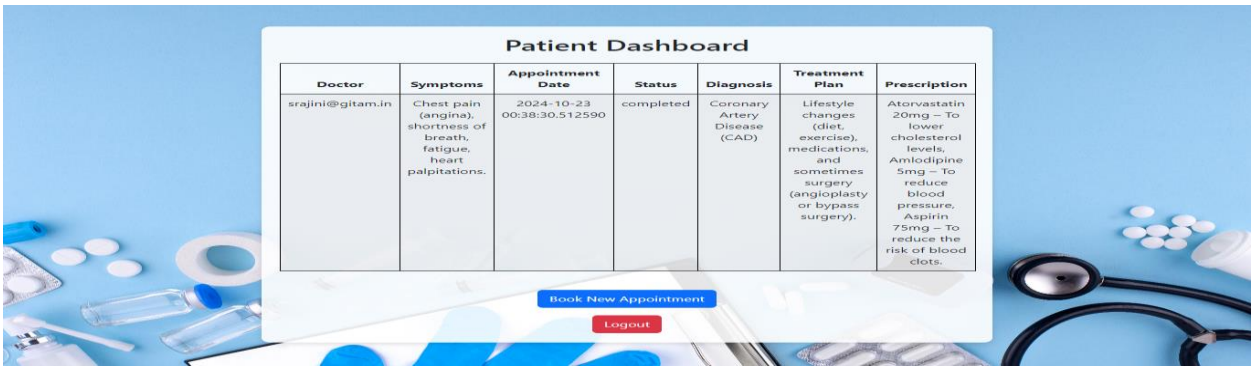
Patient Dashboard: (Before Appointment)

The Patient Dashboard displays detailed appointment information, including the doctor's name, symptoms, appointment date, and the current status of each appointment. Patients can also view diagnoses, treatment plans, and prescriptions, as well as book new appointments or log out from the system.



Book Appointment Page:

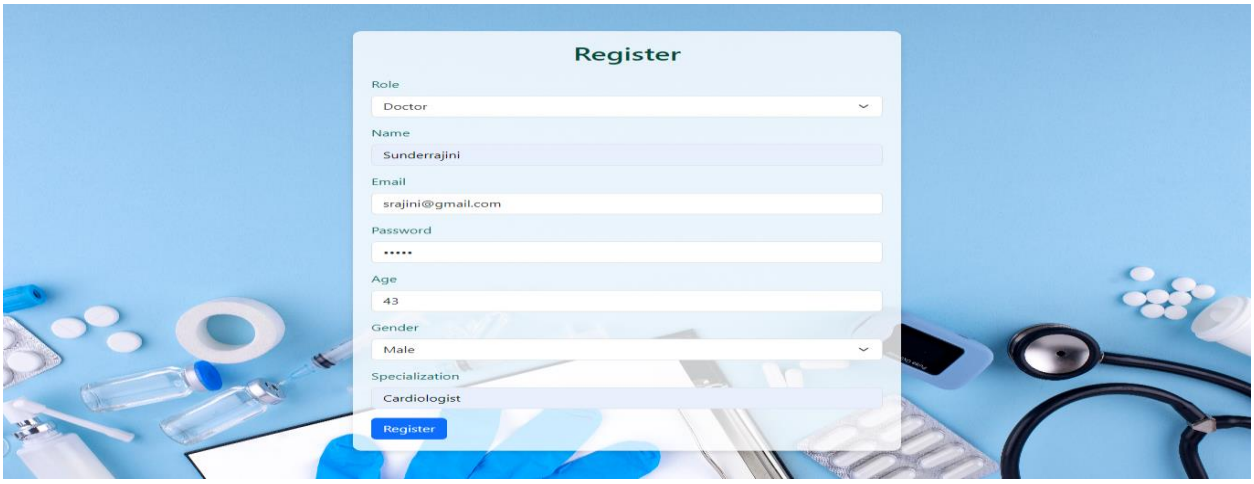
The Book Appointment page allows patients to schedule appointments by selecting a doctor, describing their symptoms, and choosing a preferred date. It provides an intuitive form to facilitate the booking process, ensuring that patients can quickly and efficiently manage their healthcare appointments.



Patient Dashboard: (After Appointment)

The Patient Dashboard displays detailed appointment information, including the doctor's name, symptoms, appointment date, and the current status of each appointment. Patients can also view diagnoses, treatment plans, and prescriptions, as well as book new appointments or log out from the system.

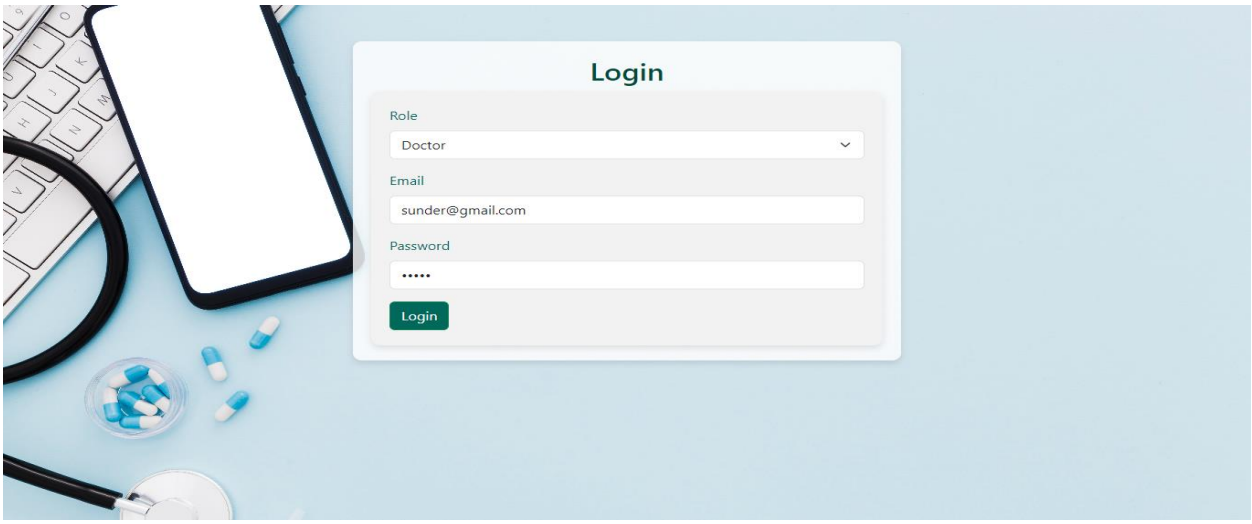
Register Page (Doctor):



Doctor Registration:

For doctors, the registration form includes an additional specialization field that appears when the doctor role is selected. Doctors can register to access patient management features and provide diagnoses through Med Track.

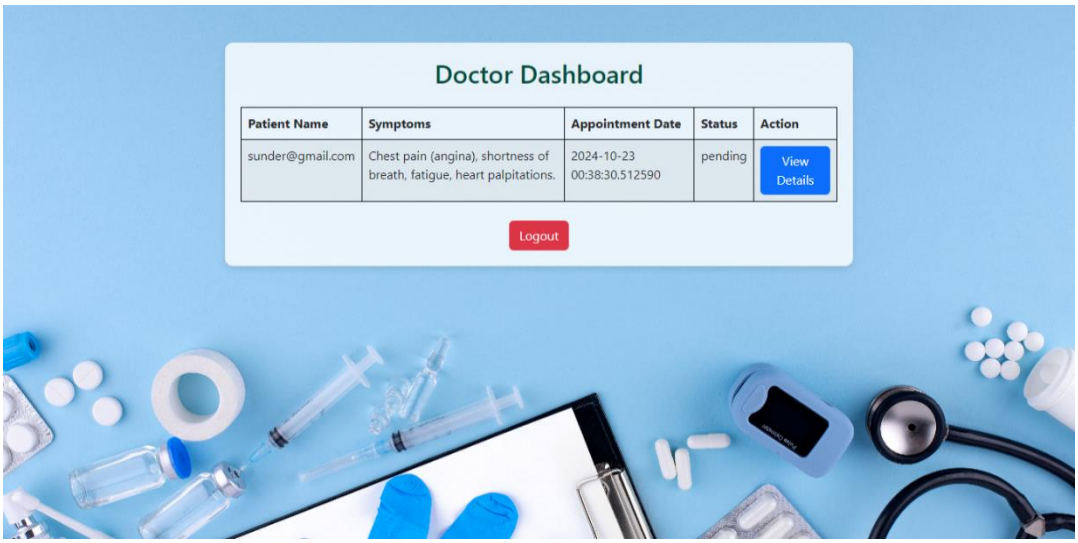
Login Page (Doctor):



Doctor Login:

For doctors, the login form requires role selection, email, and password. Upon successful login, doctors can access the platform to manage patient records, submit diagnoses, and view appointments.

Doctor Dashboard Page:(Before Consulting)

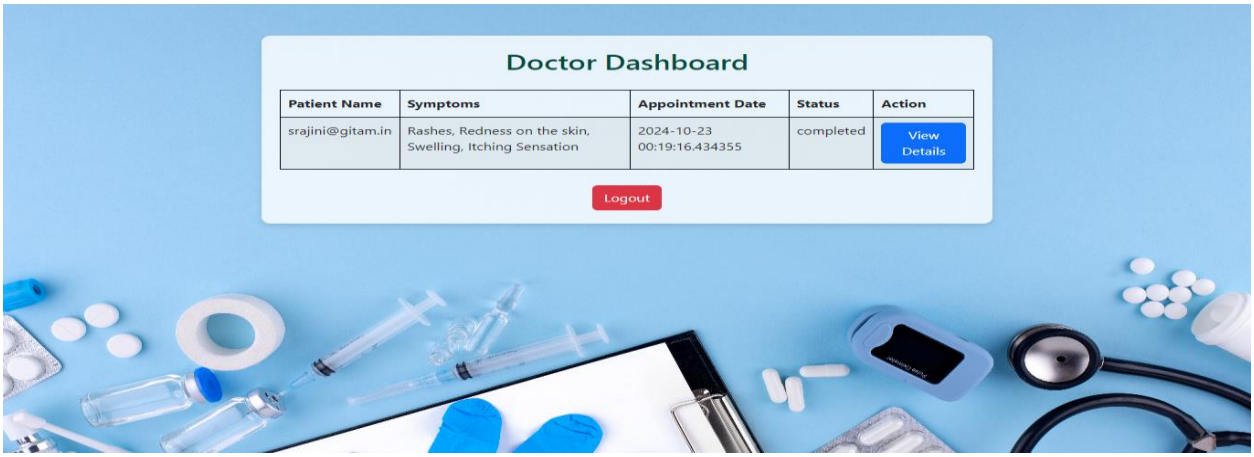


Doctor Dashboard provides an organized view of upcoming appointments, displaying patient details, symptoms, and appointment status in a clean table format. Doctors can click "View Details" to update diagnosis, treatment plans, and prescriptions for each appointment. The interface is user-friendly, with a logout option for easy session management.

View Appointment Page:

The View Appointment page provides a detailed summary of a patient's scheduled appointment, including information about the doctor, symptoms, and the appointment status. It also includes a form for doctors to input the diagnosis, treatment plan, and prescription for the patient after the consultation. This page is designed to streamline the process of updating medical records and managing patient care in an organized and efficient way.

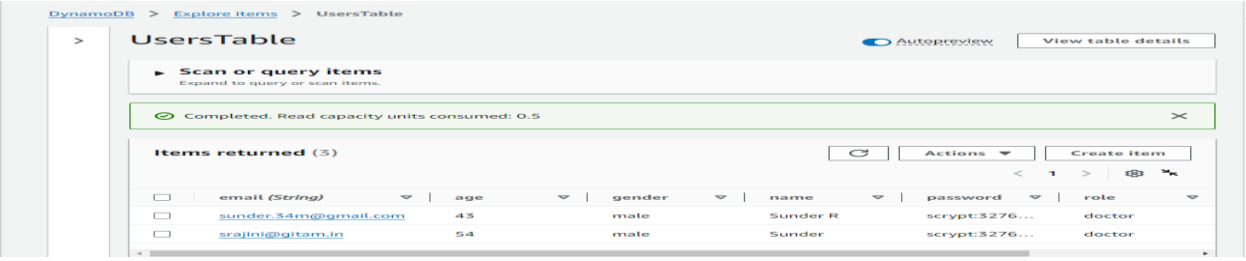
Doctor Dashboard Page:(After Consulting)



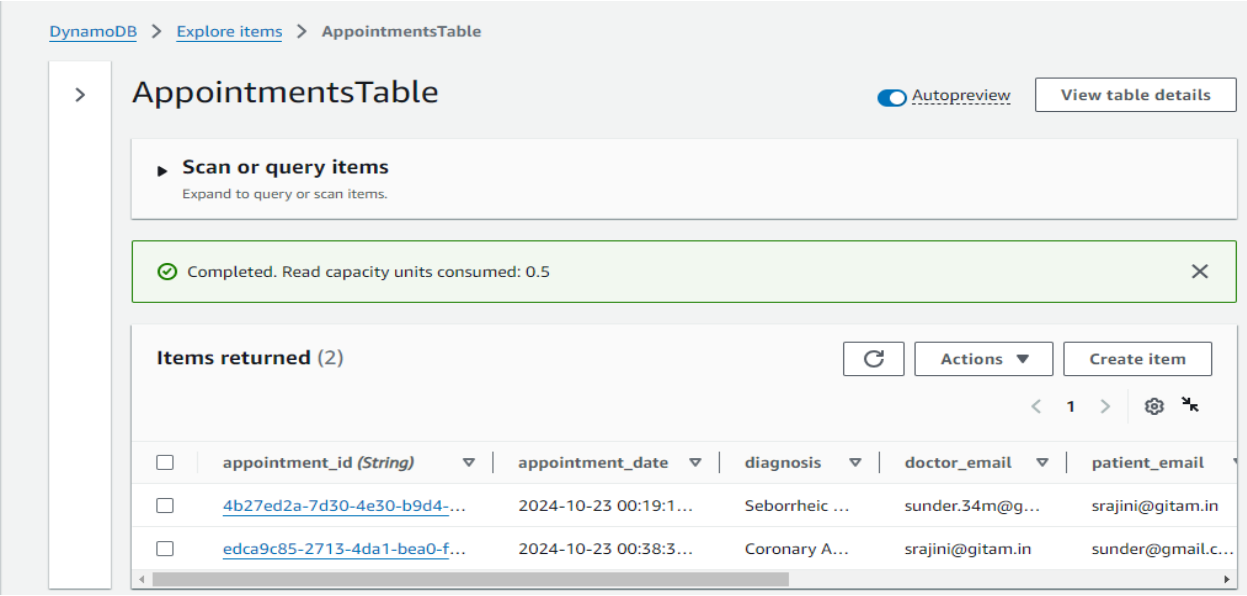
Doctor Dashboard provides an organized view of upcoming appointments, displaying patient details, symptoms, and appointment status in a clean table format. Doctors can click "View Details" to update diagnosis, treatment plans, and prescriptions for each appointment. The interface is user-friendly, with a logout option for easy session management.

DynamoDB up dations:

1. **Users table:**



2.**Appointments table:**



Conclusion: The Med Track application has been successfully developed and deployed using a robust cloud-based architecture with AWS services such as EC2 for hosting, DynamoDB for data management, and IAM roles for secure access control. This platform enhances doctor-patient interactions by enabling patients to book appointments and submit diagnoses while allowing doctors to manage their schedules and access patient records efficiently. The cloud-native approach ensures seamless scalability, accommodating increasing user demand without compromising performance. Comprehensive testing has verified that all functionalities, from user registration to appointment management, operate smoothly. Ultimately, Med Track exemplifies the potential of cloud-based systems to address real-world challenges in the healthcare sector, improving communication and user experience.