Online Complaint: Cloud-Based Online Complaint Registration System using Flask on AWS EC2 and RDS

Project Description:

The **Complaint Management System** is a web-based platform designed to help residents report and track issues like streetlights, water leaks, and road maintenance. This system eliminates manual complaint handling by providing an automated, transparent, and efficient solution.

Built using **Flask (Python), MySQL (Amazon RDS), and AWS services**, it enables users to register, log complaints, and track their progress in real-time. Complaints are assigned to field agents via an **admin dashboard**, ensuring efficient issue resolution.

The system is hosted on **Amazon EC2**, with **Amazon S3** handling complaint-related image storage. **Amazon RDS (MySQL)** securely manages user and complaint data, while **Amazon SNS** sends notifications via email/SMS. **AWS IAM** ensures secure access control, and **Flask-SQLAlchemy** is used for database interactions.

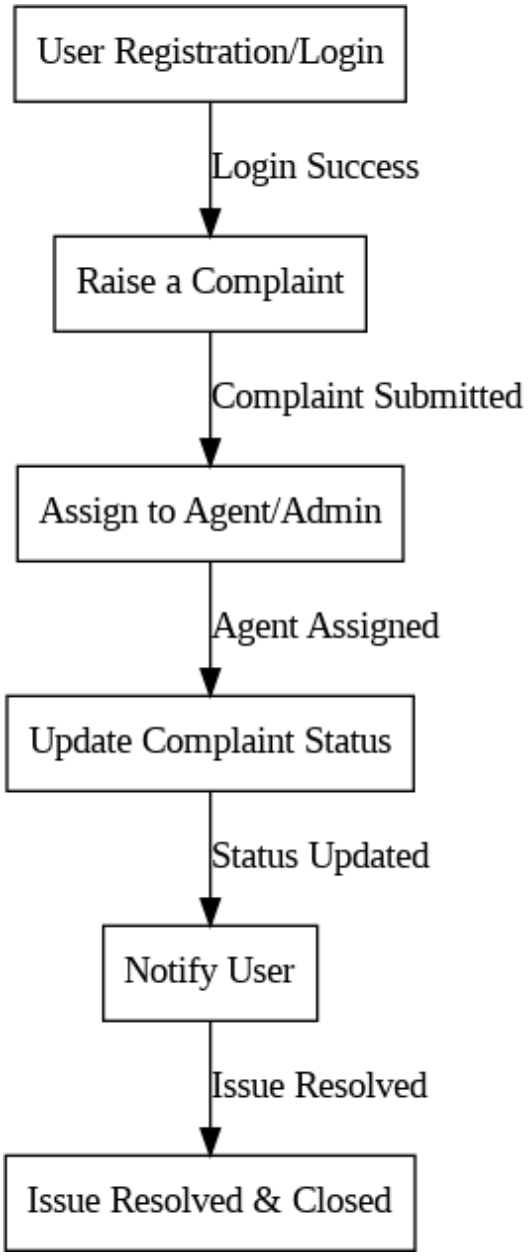
Users receive updates as complaints progress through different stages, from submission to resolution. Agents can update complaint statuses, and users can provide feedback upon issue resolution. This cloud-based solution improves **complaint tracking, response times, and public service efficiency**.

By leveraging AWS and Flask, the system offers a **scalable, reliable, and secure** grievance management platform.

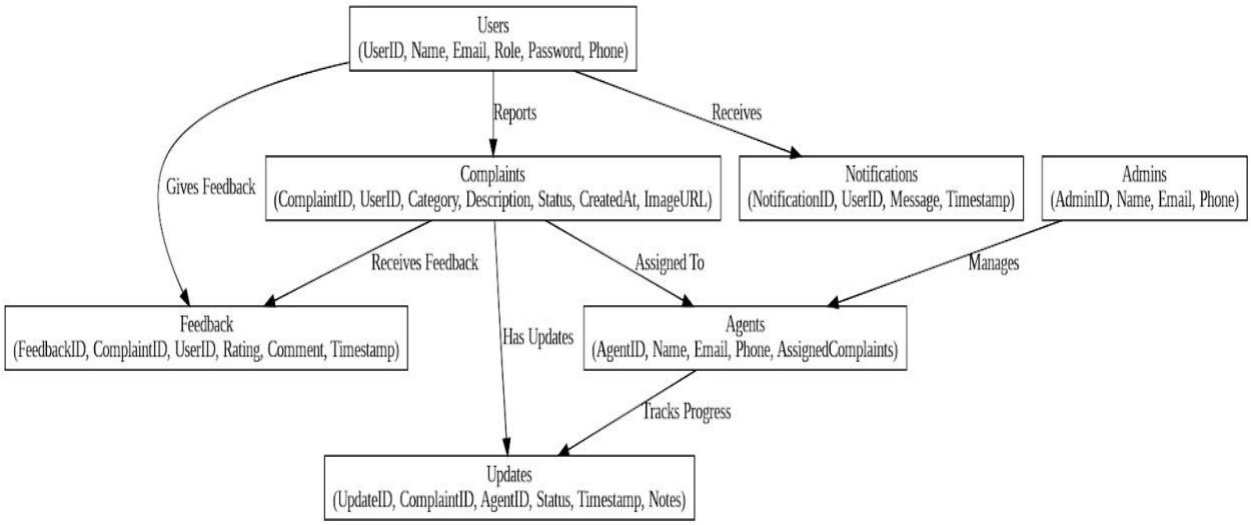
Admin: The main role and responsibilities of the admin is to take care of the whole process. Starting from Admin login, followed by the agent creation and assigning issues to the customer's complaints. Finally, He can track the work assigned to the agent, which will be updated in the database.

User: Users can register for an account. After logging in, they can create a complaint by providing a description of the problem, selecting the type of issue, uploading an image file, and specifying the location of the issue they are facing. Each user will be assigned an agent. They can then view the status of their complaint on their page.

Technical Architecture:



ER Diagram:



Pre-requisites:

1. AWS Account Setup: https://youtu.be/CjKhQoYeR4Q?si=ui8Bvk_M4FfVM-Dh
2. Understanding of IAM: https://youtu.be/gsgdAyGhV0o?si=3qg-bULgkD4LXNvR
3. Knowledge of Amazon EC2: https://youtu.be/8TlukLu11Yo?si=MUj0nEAOESRhHUIz
4. MySQL: https://www.youtube.com/results?search_query=mysql+tutorial
5. RDS connects MySQL: https://www.youtube.com/results?search_query=mysql+connector+for+rds
6. RDS: https://www.youtube.com/live/MPau9c7PT74?si=A8OK-zFGbSKkAFWN

Project Workflow:

1. Project Initialization:

- Define objectives, scope, and KPIs for deploying the Stocker cloud based platform.
- Set up the AWS environment, including EC2 instance configuration and RDS setup
- Outline the use of Flask for backend development and integration of AWS services.

2. EC2 Instance Creation:

- Launch an EC2 instance to host the Stocker application.
- Select the appropriate instance type based on expected traffic and resource requirements.

3. RDS Configuration:

- Set up Amazon RDS for database management with MySQL.
- Configure database instances, including security settings and access controls.

4. Flask Application Deployment:

- Develop and deploy the Stocker application using Flask.
- Transfer application files to the EC2 instance and configure the environment.

5. Testing and Optimization:

- Test the application for functionality, performance, and security.
- Optimize server settings, database configurations, and application performance.

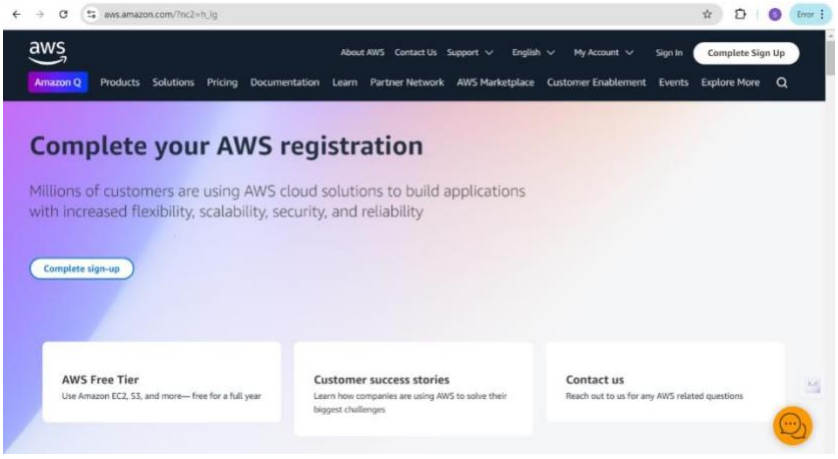
6. Monitoring and Maintenance:

- Implement monitoring tools (e.g., AWS CloudWatch) to track application performance and uptime.
- Regularly update and maintain the application and server to ensure ongoing reliability and scalability.

Milestone 1: AWS Account Setup and Login

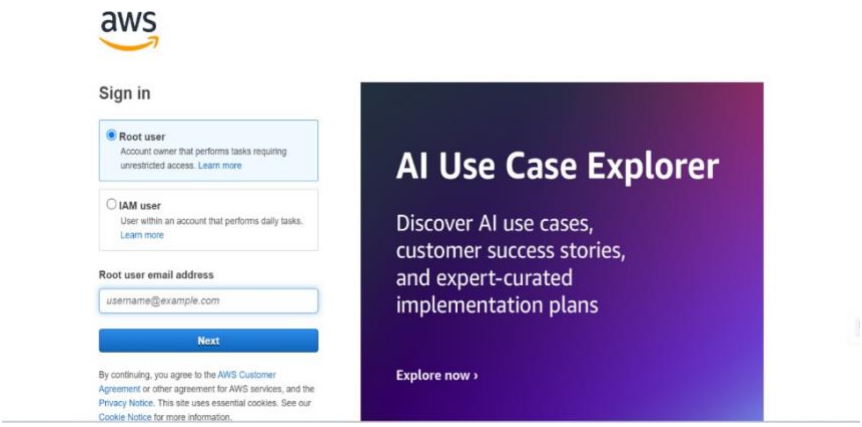
- **Activity 1.1: Create AWS Account**

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



- **Activity 1.2: Login to AWS Management Console**

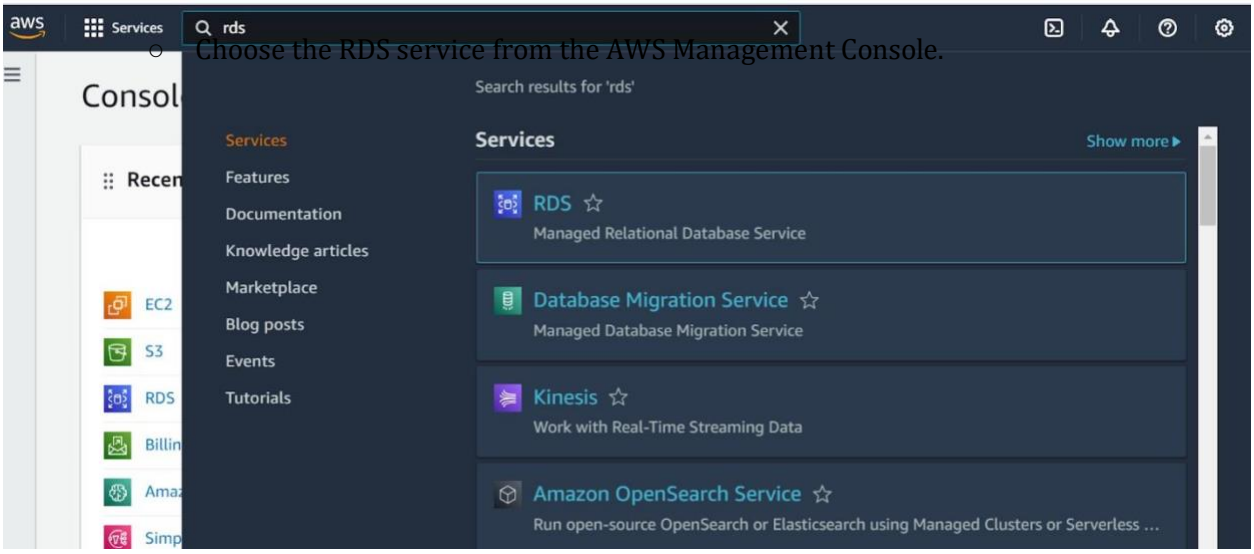
    - Access the AWS Management Console using your login credentials.



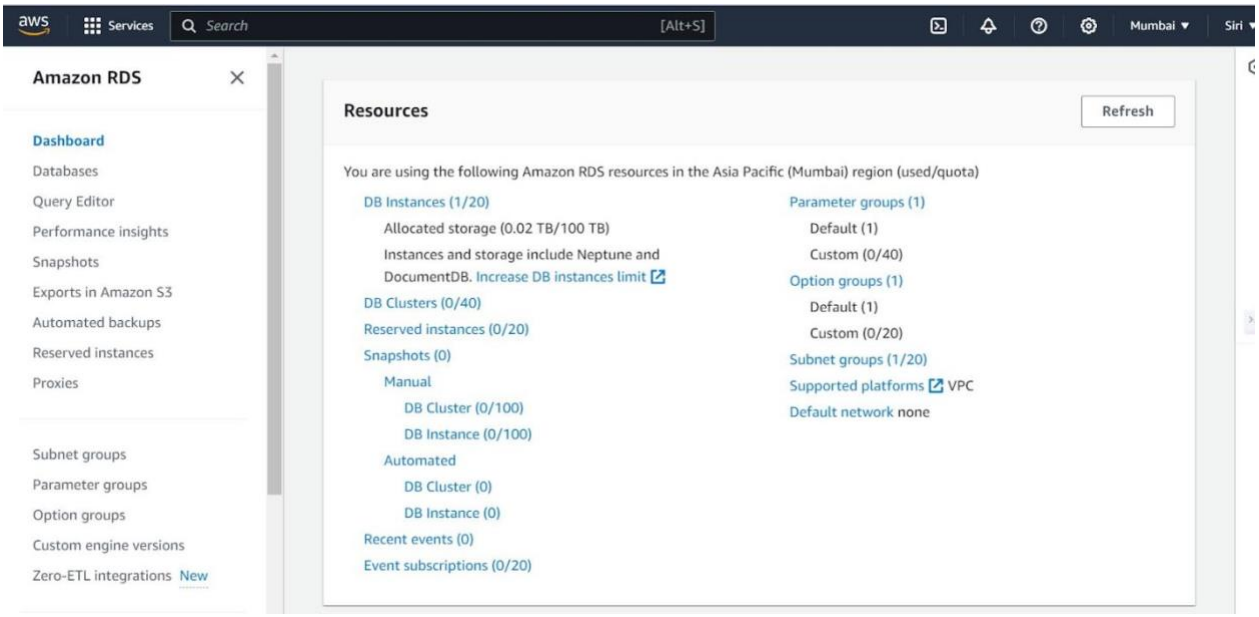
Milestone 2: RDS Database Creation and Setup

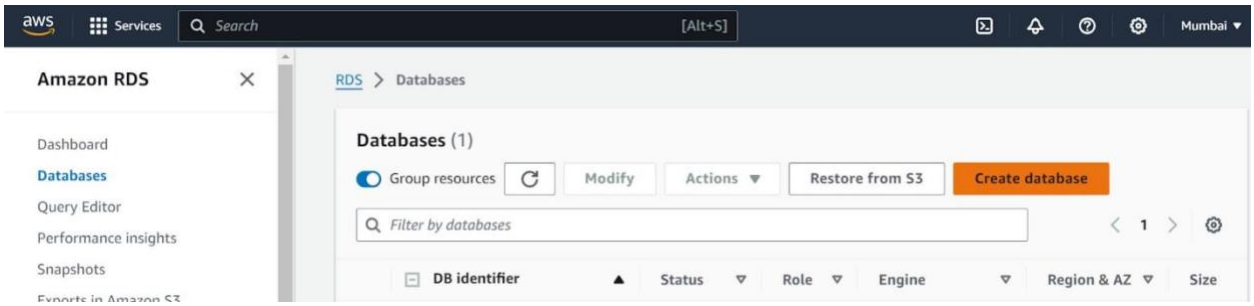
- **Activity 2.1: Create an RDS Instance**

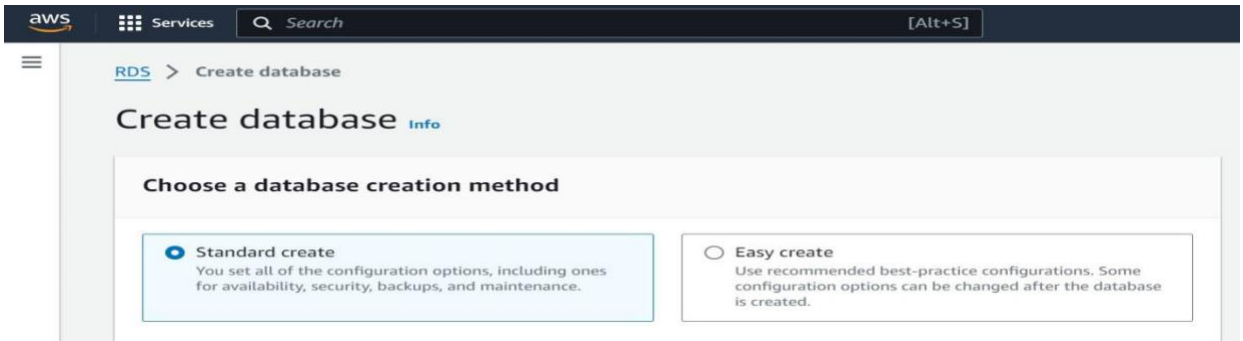
    - Choose the RDS service from the AWS Management Console.

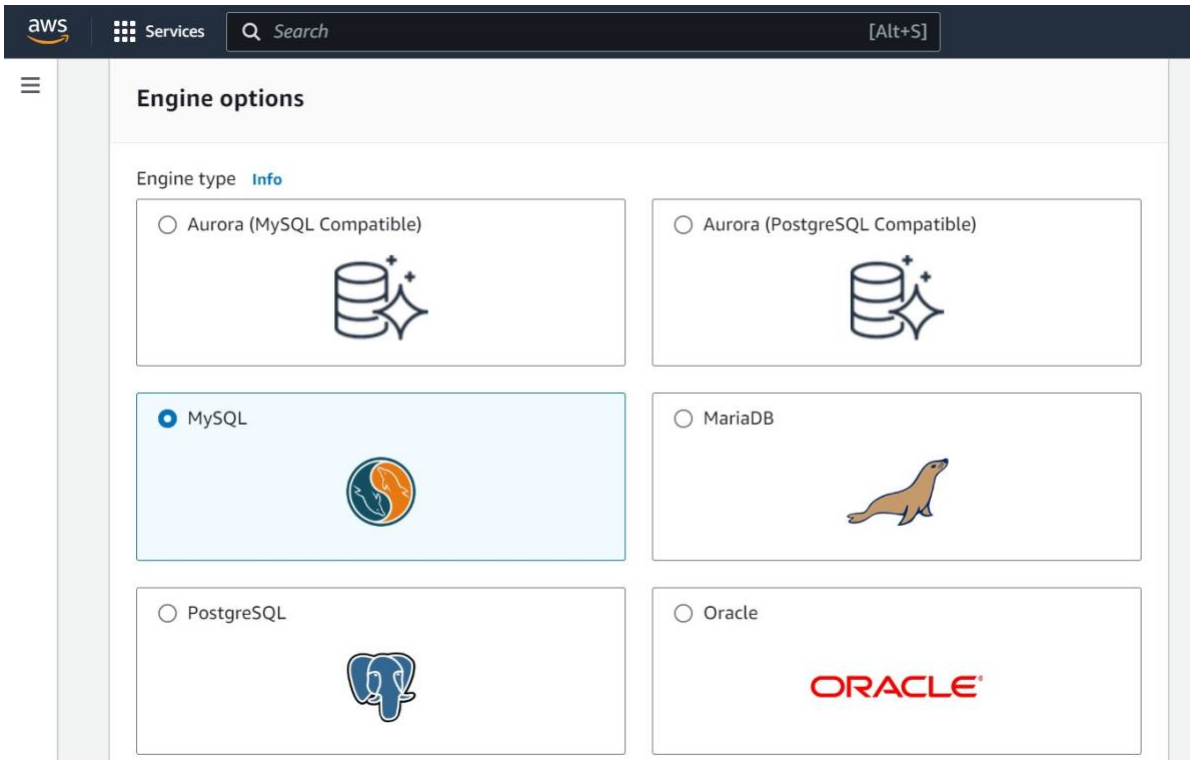


Select MySQL as the database engine, configure the instance settings (e.g., storage, instance class), and launch the RDS instance.









Edition

MySQL Community

Engine version Info

View the engine versions that support the following database features.

Hide filters

Show versions that support the Multi-AZ DB cluster Info

Create a A Multi-AZ DB cluster with one primary DB instance and two readable standby DB instances. Multi-AZ DB clusters provide up to 2x faster transaction commit latency and automatic failover in typically under 35 seconds.

Show versions that support the Amazon RDS Optimized Writes Info

Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.

Engine Version

MySQL 8.0.35

Enable RDS Extended Support Info

Amazon RDS Extended Support is a paid offering. By selecting this option, you consent to being charged for this offering if you are running your database major version past the RDS end of standard support date for that version. Check the end of standard support date for your major version in the RDS for MySQL documentation.

Templates

Choose a sample template to meet your use case.

- ○ **Production**
  Use defaults for high availability and fast, consistent performance.
- ○ **Dev/Test**
  This instance is intended for development use outside of a production environment.
- ◉ **Free tier**
  Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS. **Info**

Availability and durability

Deployment options **Info**

The deployment options below are limited to those supported by the engine you selected above.

- ○ Multi-AZ DB Cluster
  Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.
- ○ Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.
- ◉ Single DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a single DB instance with no standby DB instances.

Settings

DB instance identifier **Info**

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

database-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username **Info**

Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. The first character must be a letter.

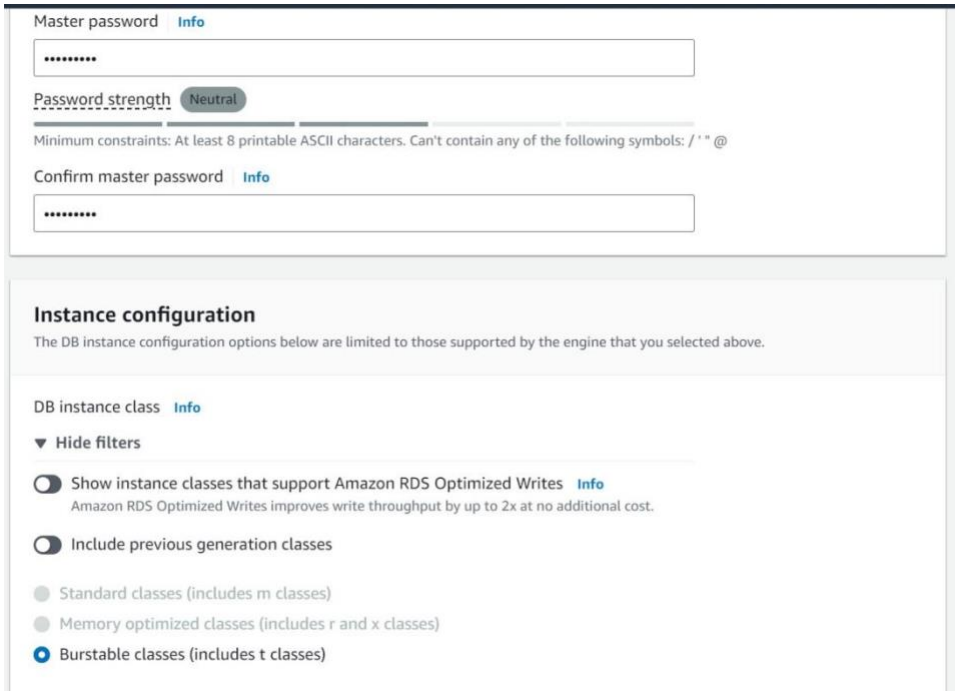
Credentials management

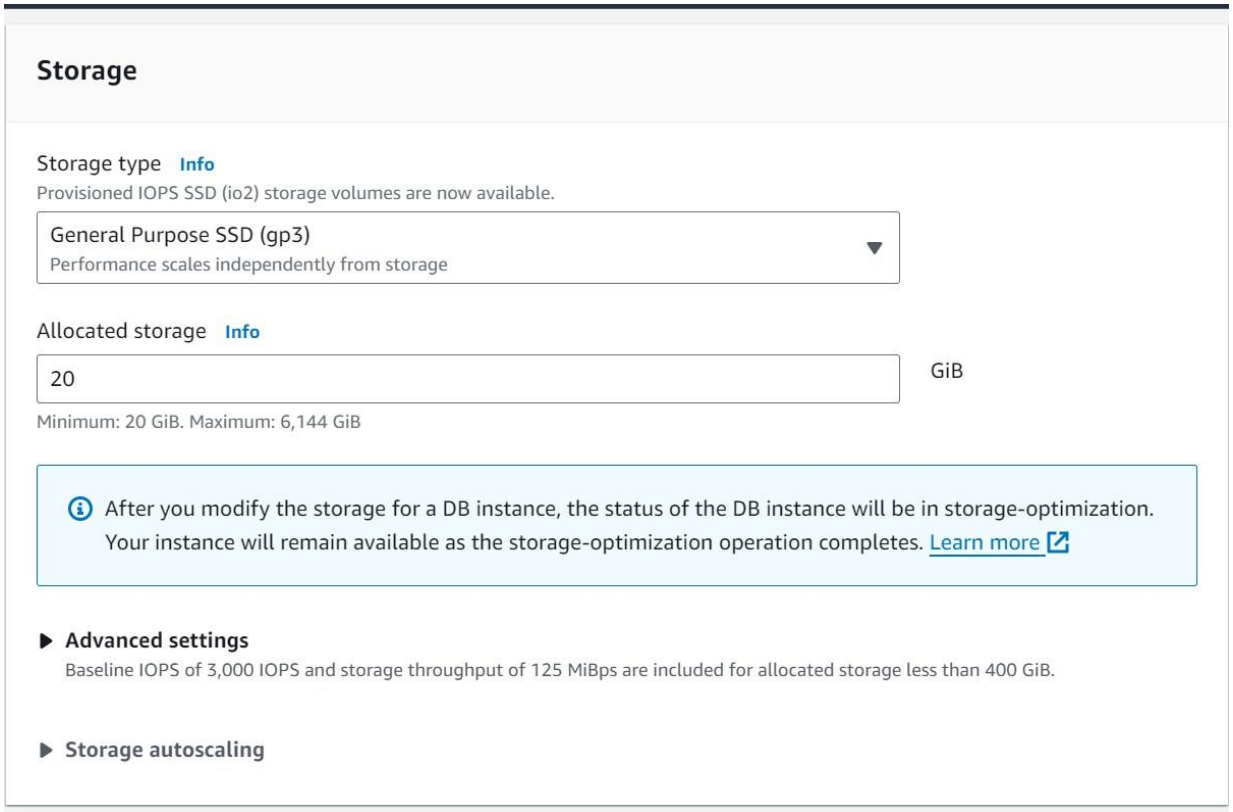
You can use AWS Secrets Manager or manage your master user credentials.

- ○ Managed in AWS Secrets Manager - *most secure*
  RDS generates a password for you and manages it throughout its lifecycle using AWS Secrets Manager.
- ◉ Self managed
  Create your own password or have RDS create a password that you manage.

☐ Auto generate password

Amazon RDS can generate a password for you, or you can specify your own password.


Master password Info
Password strength Neutral
Minimum constraints: At least 8 printable ASCII characters. Can't contain any of the following symbols: / ' " @
Confirm master password Info
Instance configuration
The DB instance configuration options below are limited to those supported by the engine that you selected above.
DB instance class Info
Hide filters
Show instance classes that support Amazon RDS Optimized Writes Info
Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.
Include previous generation classes
Standard classes (includes m classes)
Memory optimized classes (includes r and x classes)
Burstable classes (includes t classes)


Storage
Storage type Info
Provisioned IOPS SSD (io2) storage volumes are now available.
General Purpose SSD (gp3)
Performance scales independently from storage
Allocated storage Info
20
GiB
Minimum: 20 GiB. Maximum: 6,144 GiB
After you modify the storage for a DB instance, the status of the DB instance will be in storage-optimization.
Your instance will remain available as the storage-optimization operation completes. Learn more
Advanced settings
Baseline IOPS of 3,000 IOPS and storage throughput of 125 MiBps are included for allocated storage less than 400 GiB.
Storage autoscaling

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

- (●) Don't connect to an EC2 compute resource
  Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.
- ( ) Connect to an EC2 compute resource
  Set up a connection to an EC2 compute resource for this database.

Network type Info

To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

- (●) IPv4
  Your resources can communicate only over the IPv4 addressing protocol.
- ( ) Dual-stack mode
  Your resources can communicate over IPv4, IPv6, or both.

Virtual private cloud (VPC) Info

Choose the VPC. The VPC defines the virtual networking environment for this DB instance.

vpc-0c9a09cc4c0ef9dd5
3 Subnets, 3 Availability Zones

Only VPCs with a corresponding DB subnet group are listed.

ⓘ After a database is created, you can't change its VPC.

Public access Info

- (•) Yes
  RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.
- ( ) No
  RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

- ( ) Choose existing
  Choose existing VPC security groups
- (•) Create new
  Create new VPC security group

New VPC security group name

new security group

Availability Zone Info

No preference

RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

- [ ] Create an RDS Proxy Info
  RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see Amazon RDS Proxy pricing.

Database authentication

Database authentication options Info

- (•) Password authentication
  Authenticates using database passwords.
- ( ) Password and IAM database authentication
  Authenticates using the database password and user credentials through AWS IAM users and roles.
- ( ) Password and Kerberos authentication
  Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

Monitoring

- [ ] Enable Enhanced Monitoring
  Enabling Enhanced Monitoring metrics are useful when you want to see how different processes or threads use the CPU.

▸ Additional configuration

Database options, encryption turned on, backup turned on, backtrack turned off, maintenance, CloudWatch Logs, delete protection turned off.

This billing estimate is based on on-demand usage as described in Amazon RDS Pricing. Estimate does not include costs for backup storage, IOs (if applicable), or data transfer.

Estimate your monthly costs for the DB Instance using the AWS Simple Monthly Calculator.

Estimated monthly costs

The Amazon RDS Free Tier is available to you for 12 months. Each calendar month, the free tier will allow you to use the Amazon RDS resources listed below for free:

- 750 hrs of Amazon RDS in a Single-AZ db.t2.micro, db.t3.micro or db.t4g.micro Instance.
- 20 GB of General Purpose Storage (SSD).
- 20 GB for automated backup storage and any user-initiated DB Snapshots.

Learn more about AWS Free Tier.

When your free usage expires or if your application use exceeds the free usage tiers, you simply pay standard, pay-as-you-go service rates as described in the Amazon RDS Pricing page.

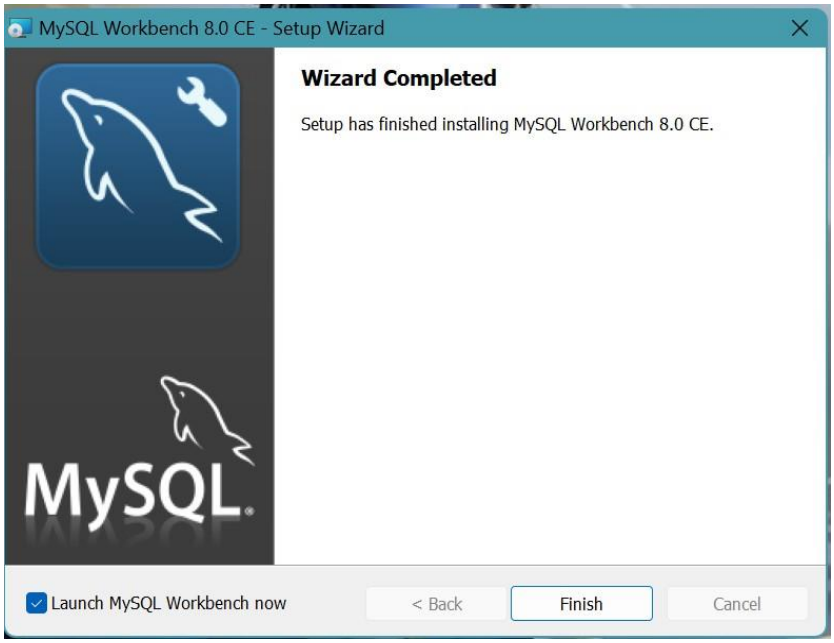
You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

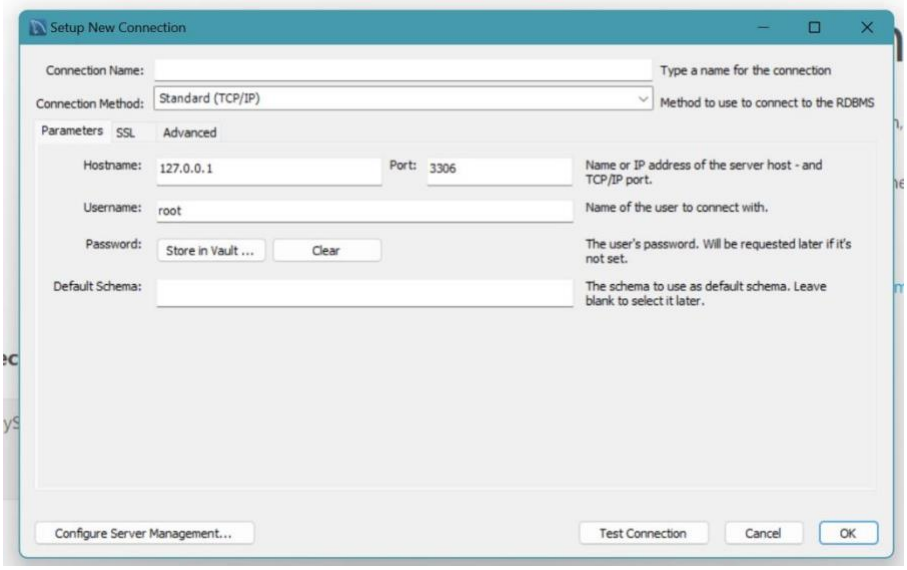
Cancel | Create database

- **Activity 2.2: Configure Database Access**
    - Set up security groups, create database credentials, and configure access policies to ensure secure connectivity to the database.

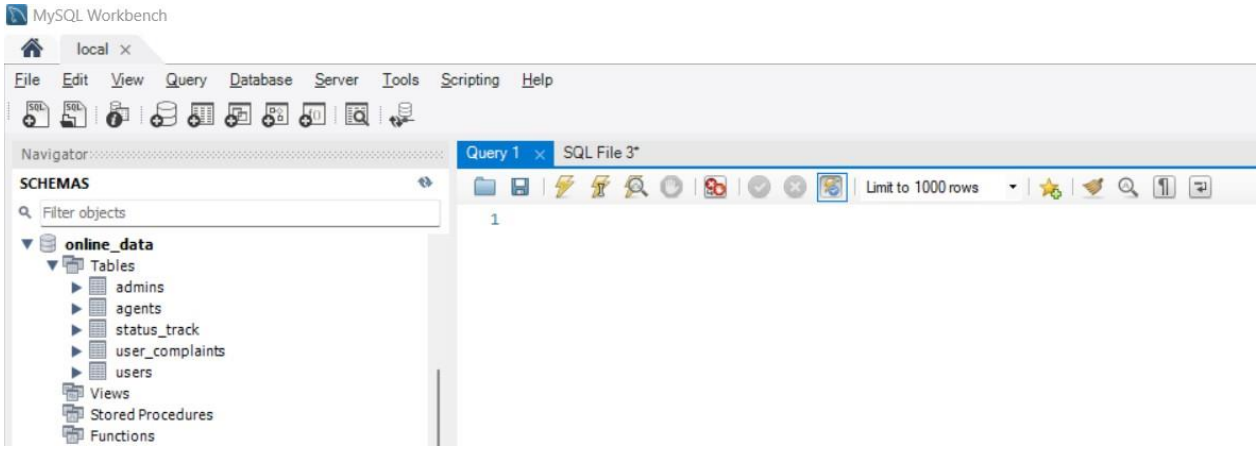
- **Activity 2.3: Install MySQL Workbench**
    - Download and install MySQL Workbench on your local machine for database management.

    - Connect to the RDS instance via MySQL Workbench using the endpoint and credentials from AW





- Give a connection name.
- Copy the endpoint from the RDS database that is created in AWS and paste it into **Hostname.**
- Write the username, enter the password, and click **Test Connection**.
- Once the connection is successful, you'll be welcomed with this interface



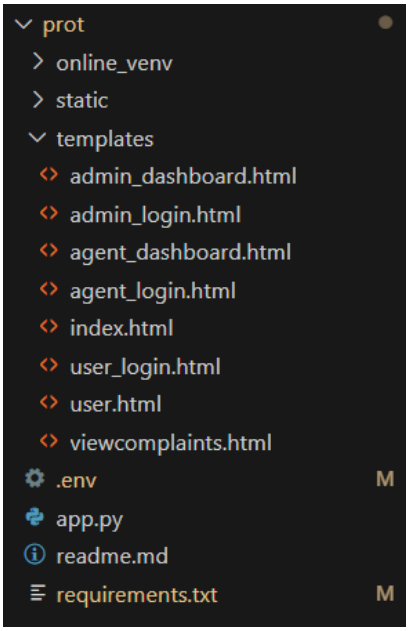
Activity 2.4: Create the database and the required tables.

- **Create a basic database schema for Online Complaint Management**

```
CREATE DATABASE online_data;
USE online_data;
```

Milestone 3: Frontend Development and Application Setup

- **Activity 3.1: Build the Frontend**
  - Develop HTML, CSS, and Python-based Flask application files for Online Complaint Registration frontend interface.



- **Activity 3.2: Integrate Application with RDS**
  - Connect app.py (Flask application) to the MySQL RDS database by configuring database connection settings and verifying connectivity.

Description of the code:

1 Imports and Setup:

The script starts by importing necessary libraries and setting up the Flask application:

- Flask: A web framework for creating web applications in Python.

- PyMySQL: A library for working with databases in Python.

- Flask-Login: Handles user sessions for logging in and out.

- Other libraries: For making HTTP requests, working with dates, and handling passwords.

```python
from flask import Flask, jsonify, render_template, request, redirect, url_for, flash, session
import pymysql # type: ignore
from werkzeug.utils import secure_filename
import bcrypt
from dotenv import load_dotenv
import os

# Load environment variables from the .env file
load_dotenv()


app = Flask(__name__)
```

2 Database Configuration: The script sets up the database connection and configures the application:

- It uses environment variables to securely store database credentials and API keys.

- The database is set up to use MySQL.

- A secret key is set for the application (used for security purposes).

```python
app = Flask(__name__)
app.secret_key = 'your_secret_key'


db_config = {
    'host': os.getenv('DB_HOST'),
    'user': os.getenv('DB_USER'),
    'password': os.getenv('DB_PASSWORD'),
    'database': os.getenv('DB_NAME')
}
```

3 Database Models: We have created several tables in the database online_data:

1. users: Stores user details like name, email, password, and account creation timestamp.

2. agents: Stores agent information including name, role, email, and password.

3. admins: Stores admin information, including name, role, email, and password.

4. user_complaints: Records complaints submitted by users, including details such as issue type, description, and location. It references the `users` table.

5. status_track: Tracks the status of a complaint, including when the status was updated, who accepted it, and the action taken. It references the `user_complaints` table.

```
-- Create the users table
-- CREATE TABLE users (
--     id INT AUTO_INCREMENT PRIMARY KEY,
--     name VARCHAR(255) NOT NULL,
--     email VARCHAR(255) NOT NULL UNIQUE,
--     password VARCHAR(255) NOT NULL,
--     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
-- );

-- -- Create the user_complaints table
-- CREATE TABLE user_complaints (
--     complaint_id SERIAL PRIMARY KEY,
--     issue_type VARCHAR(255) NOT NULL,
--     image_path VARCHAR(255),
--     description TEXT NOT NULL,
--     address VARCHAR(255) NOT NULL,
--     latitude VARCHAR(50),
--     longitude VARCHAR(50),
--     user_id INT NOT NULL,  -- Foreign key referencing users
--     submitted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
--     FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
-- );
```

```
-- CREATE TABLE agents (
--     id INT AUTO_INCREMENT PRIMARY KEY,
--     name VARCHAR(255) NOT NULL,
--     role VARCHAR(255) NOT NULL,
--     email VARCHAR(255) NOT NULL UNIQUE,
--     password VARCHAR(255) NOT NULL,
--     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
-- );

-- CREATE TABLE admins (
--     id INT AUTO_INCREMENT PRIMARY KEY,
--     name VARCHAR(100) NOT NULL,
--     role VARCHAR(50),
--     email VARCHAR(100) UNIQUE NOT NULL,
--     password VARCHAR(255) NOT NULL
-- );

-- CREATE TABLE status_track (
--     track_id INT PRIMARY KEY AUTO_INCREMENT,
--     complaint_id BIGINT UNSIGNED, -- Foreign key linking to user_complaints, ensure it matches the type
--     status VARCHAR(50) NOT NULL, -- Status of the complaint: e.g., "Pending", "In Progress", "Resolved"
--     updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- To track when the status was updated
--     accepted_by INT, -- ID of the admin or agent who accepted the complaint
--     session_id VARCHAR(255), -- To track the session that processed the action
--     action_taken VARCHAR(50), -- e.g., "Accepted", "Declined"
--     FOREIGN KEY (complaint_id) REFERENCES user_complaints(complaint_id) ON DELETE CASCADE
-- );
```

4 **Routes, `/`: The home page.**: The script defines several routes (URLs) for the web application:

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

5 **`/login`**: Handles user login.

```
@app.route('/user/login', methods=['GET', 'POST'])
def user_login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password'].encode('utf-8')

        connection = get_db_connection()
        if connection:
            cursor = connection.cursor()
            try:
                cursor.execute("SELECT id, password FROM users WHERE email = %s", (email,))
                result = cursor.fetchone()

                if result and bcrypt.checkpw(password, result[1].encode('utf-8')):
                    session['user_id'] = result[0]  # Store user ID in session
                    # flash('User login successful!', 'success')
                    return redirect(url_for('user'))
                else:
                    flash('Invalid credentials. Please try again.', 'error')
            finally:
                cursor.close()
                connection.close()
        else:
            flash('Database connection error. Please try again later.', 'error')

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

6 - `/register`: New users can create an account.

```python
@app.route('/user/signup', methods=['GET', 'POST'])
def user_signup():
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
        password = request.form['password'].encode('utf-8')
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt()).decode('utf-8')

        connection = get_db_connection()
        if connection:
            cursor = connection.cursor()
            try:
                cursor.execute("INSERT INTO users (name, email, password) VALUES (%s, %s, %s)", (name, email, hashed_password))
                connection.commit()
                flash('User account created successfully!', 'success')
                return redirect(url_for('user_login'))
            except pymysql.MySQLError as e:
                flash('Error creating account. Email may already exist.', 'error')
                print(f"Error: {e}")
            finally:
                cursor.close()
                connection.close()
        else:
            flash('Database connection error. Please try again later.', 'error')

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

7 - **`/logout`:** Logs out the current user.

```python
@app.route('/user/logout')
def user_logout():
    session.pop('user_id', None)  # Clear the user ID from the session
    flash('You have been logged out.', 'success')
    return redirect(url_for('index'))
```

Similarly, it goes for the 'agents' and 'admins' table.

8 - **`/viewcomplaints`:** This route seems to be used for viewing complaints, although the function is incomplete. It might display the complaints submitted by a user after they log in.

```python
@app.route('/viewcomplaints', methods=['GET', 'POST'])
def viewcomplaints():
    if 'user_id' not in session:  # Check if user is logged in
        flash('You need to log in to access this page.', 'error')
        return redirect(url_for('login'))  # Redirect to login page if not logged in

    user_id = session.get('user_id')  # Fetch the logged-in user's ID

    connection = get_db_connection()  # Establish DB connection
    if connection:
        cursor = connection.cursor()  # Use dictionary cursor for named access
        try:
            # SQL query to fetch complaints details
            query = """      You, 15 hours ago • Initial commit
                SELECT
                    uc.complaint_id AS "Sr No",
                    uc.image_path AS "Your Uploaded Image",
                    'Image will be uploaded soon' AS "Image After Completion of work",
                    uc.issue_type AS "Title",
                    uc.description AS "Description",
                    IFNULL(a.name, 'Not Assigned') AS "Agent Name",
                    st.status AS "Work Status",
                    'Feedback Placeholder' AS "Feedback"
                FROM
                    user_complaints uc
                LEFT JOIN
                    status_track st ON uc.complaint_id = st.complaint_id
                LEFT JOIN
                    agents a ON st.accepted_by = a.id
                WHERE
                    uc.user_id = %s;  # Filter by the logged-in user's ID
            """
```

```python
            cursor.execute(query, (user_id,))  # Execute the query with the user_id parameter
            rows = cursor.fetchall()  # Fetch all the rows
            cursor.close()  # Close the cursor
            connection.close()  # Close the connection
        except Exception as e:
            print(f"An error occurred: {e}")
            flash('An error occurred while fetching the complaints.', 'error')
            rows = []  # Fallback to empty rows if there's an error
    else:
        rows = []  # Fallback to empty rows if connection fails

    # Render the 'viewcomplaints.html' template and pass the fetched rows
    return render_template('viewcomplaints.html', rows=rows)
```

9 - `/user`: This route allows logged-in users to submit complaints by uploading an image and providing other details like title, description, and location (latitude and longitude). The information is stored in the database and the user is redirected to the index page.

```python
@app.route('/user', methods=['GET', 'POST'])
def user():
    if 'user_id' not in session:  # Check if user is logged in
        flash('You need to log in to access this page.', 'error')
        return redirect(url_for('user_login'))

    if request.method == 'POST':
        # Handle file upload
        if 'image' not in request.files:
            flash('No file part', 'error')
            return redirect(request.url)

        file = request.files['image']

        if file.filename == '':
            flash('No selected file', 'error')
            return redirect(request.url)

        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            file.save(file_path)

            # Get other form data
            title = request.form.get('title')
            description = request.form.get('description')
            address = request.form.get('address')
            lat = request.form.get('lat')
            lon = request.form.get('lon')

            # Retrieve the user_id from session
            user_id = session.get('user_id')
```

```python
        # Save the details in the database
        connection = get_db_connection()
        if connection:
            cursor = connection.cursor()
            try:
                cursor.execute("""
                    INSERT INTO user_complaints (issue_type, image_path, description, address, latitude, longitude, user_id)
                    VALUES (%s, %s, %s, %s, %s, %s, %s)""",
                    (title, file_path, description, address, lat, lon, user_id))  # Include user_id here
                connection.commit()
                flash(f'Thanks for the {title} update!', 'success')  # Flash message with title
            except pymysql.MySQLError as e:
                flash('Error submitting complaint. Please try again.', 'error')
                print(f"Error: {e}")
            finally:
                cursor.close()
                connection.close()
        else:
            flash('Database connection error. Please try again later.', 'error')

        return redirect(url_for('index'))  # Redirect to the index page

    else:
        flash('Invalid file format. Please upload a valid image.', 'error')
        return redirect(request.url)  # Return to the same page on file upload error

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

10 - /agent_dashboard`: This route displays the agent dashboard, where agents can view assigned complaints. It retrieves complaints and associated user information from the database.

```python
@app.route('/agent_dashboard')
def agent_dashboard():
    if 'agent' not in session:  # Check if agent is logged in
        flash('You need to log in to access this page.', 'error')
        return redirect(url_for('agent_login'))

    connection = get_db_connection()
    if connection:
        cursor = connection.cursor()
        cursor.execute("""
            SELECT u.name AS user_name, c.issue_type, c.image_path
            FROM users u
            JOIN user_complaints c ON u.id = c.user_id;
        """)
        ag_dash = cursor.fetchall()
        print("Agent Dashboard",ag_dash)

        cursor.close()
        connection.close()
    else:
        ag_dash = []

    return render_template('agent_dashboard.html', ag_dash=ag_dash)
```

11 - **/admin_dashboard`:** Displays the admin dashboard, where admins can view user complaints and assign agents. It pulls data from the database, including users, complaints, and agent names.

```python
@app.route('/admin_dashboard')
def admin_dashboard():
    if 'admin' not in session:  # Check if admin is logged in
        flash('You need to log in to access this page.', 'error')
        return redirect(url_for('admin_login'))

    connection = get_db_connection()
    if connection:
        cursor = connection.cursor()
        cursor.execute("""
            SELECT u.name AS user_name, c.issue_type, c.image_path
            FROM users u
            JOIN user_complaints c ON u.id = c.user_id;
        """)
        users = cursor.fetchall()
        print("Admin Dashboard",users)

        # Fetch agent names
        cursor.execute("SELECT name FROM agents;")
        agents = cursor.fetchall()
        print(agents)

        cursor.close()
        connection.close()
    else:
        users = []
        agents = []

    return render_template('admin_dashboard.html', users=users, agents=agents)
```

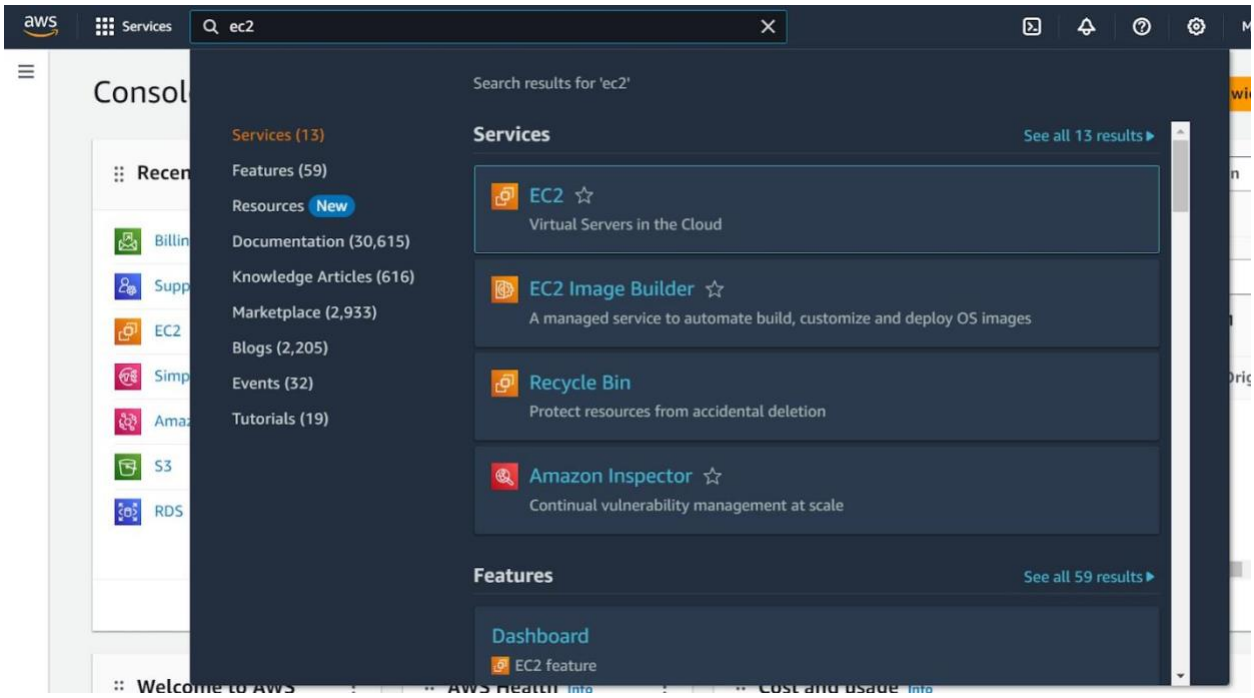
12 - **`/assign-agent`:** This route assigns an agent to a user complaint. It accepts a JSON object with the `userId` and `agentName`, updates the database, and returns a JSON response indicating success.

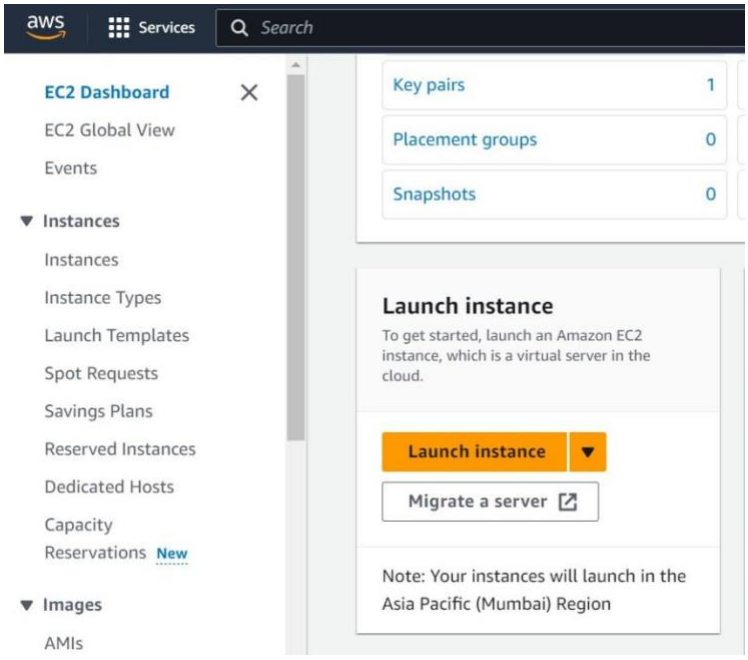
```python
@app.route('/assign-agent', methods=['POST'])
def assign_agent():
    data = request.get_json()
    user_id = data.get('userId')
    agent_name = data.get('agentName')

    # Here you would update your database or data structure with the assignment
    print(f"Assigned {agent_name} to user ID: {user_id}")

    return jsonify({"success": True})
```

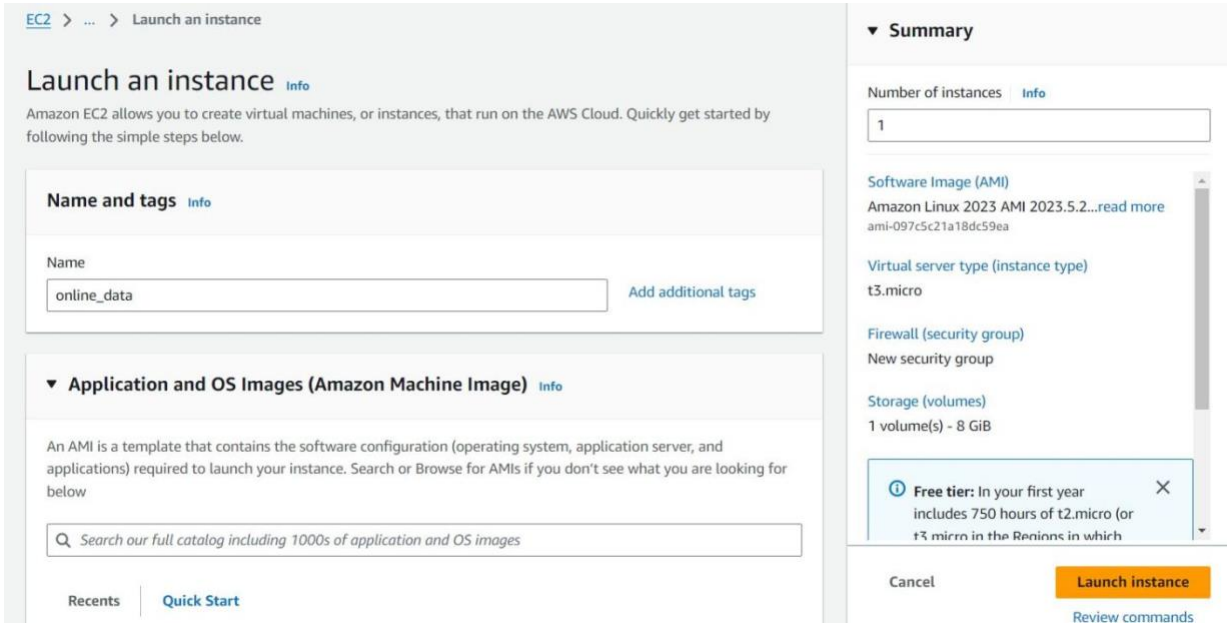
Milestone 4: EC2 Instance Setup

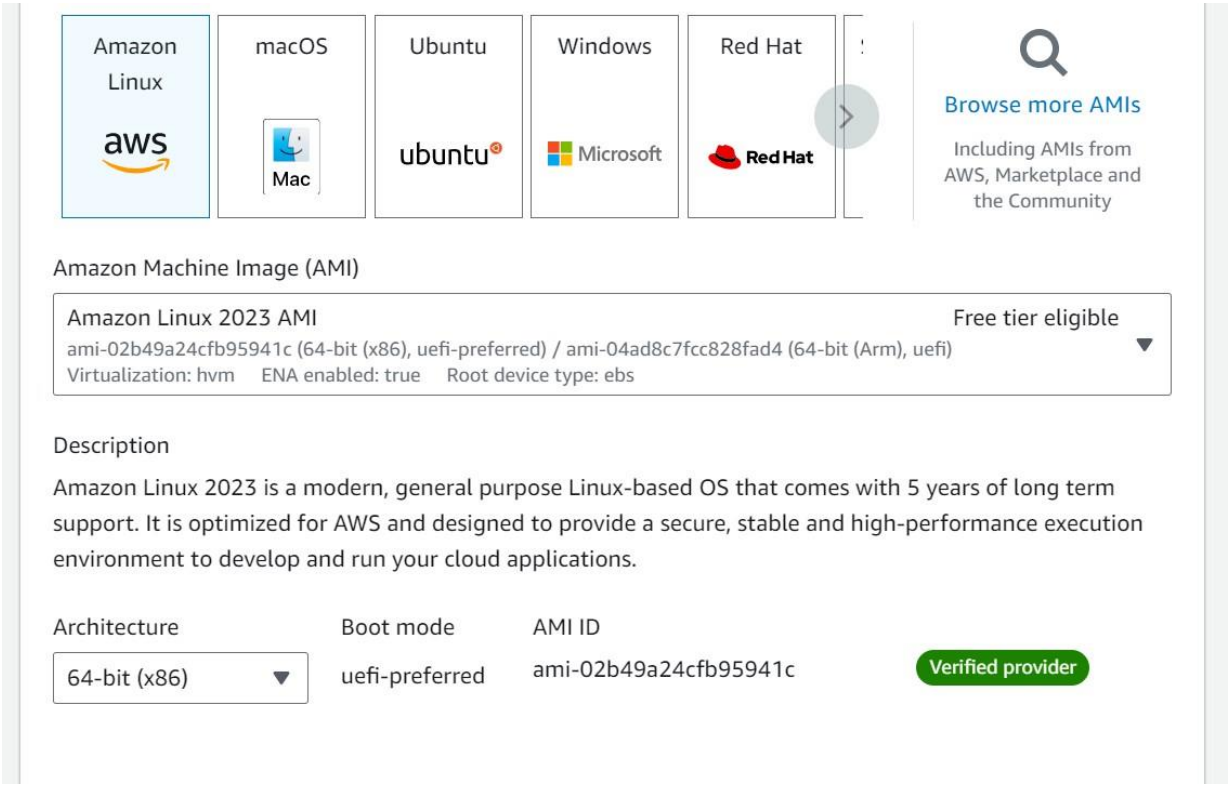
- **Activity 4.1: Launch EC2 Instance**

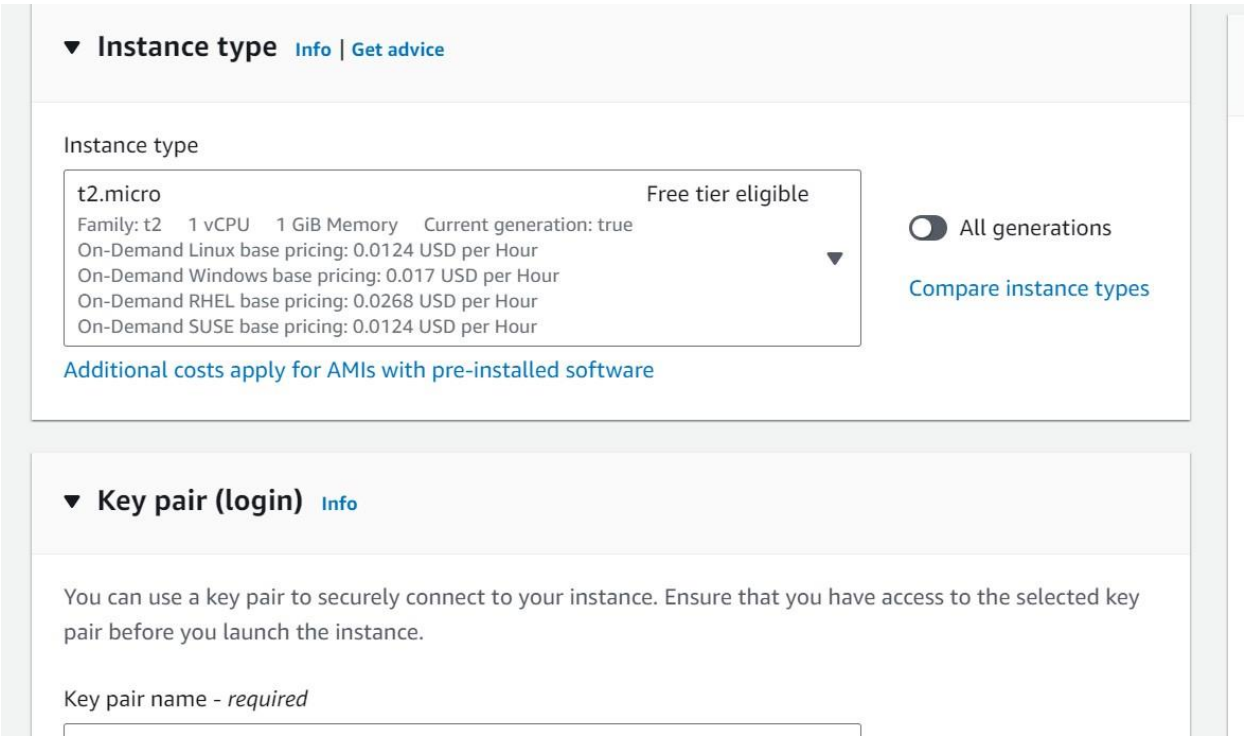
  Choose a Linux-based EC2 instance from the AWS Console to host the Stocker application.




aws
Services
Search
EC2 Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations New
Images
AMIs
Key pairs 1
Placement groups 0
Snapshots 0
Launch instance
To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.
Launch instance
Migrate a server
Note: Your instances will launch in the Asia Pacific (Mumbai) Region

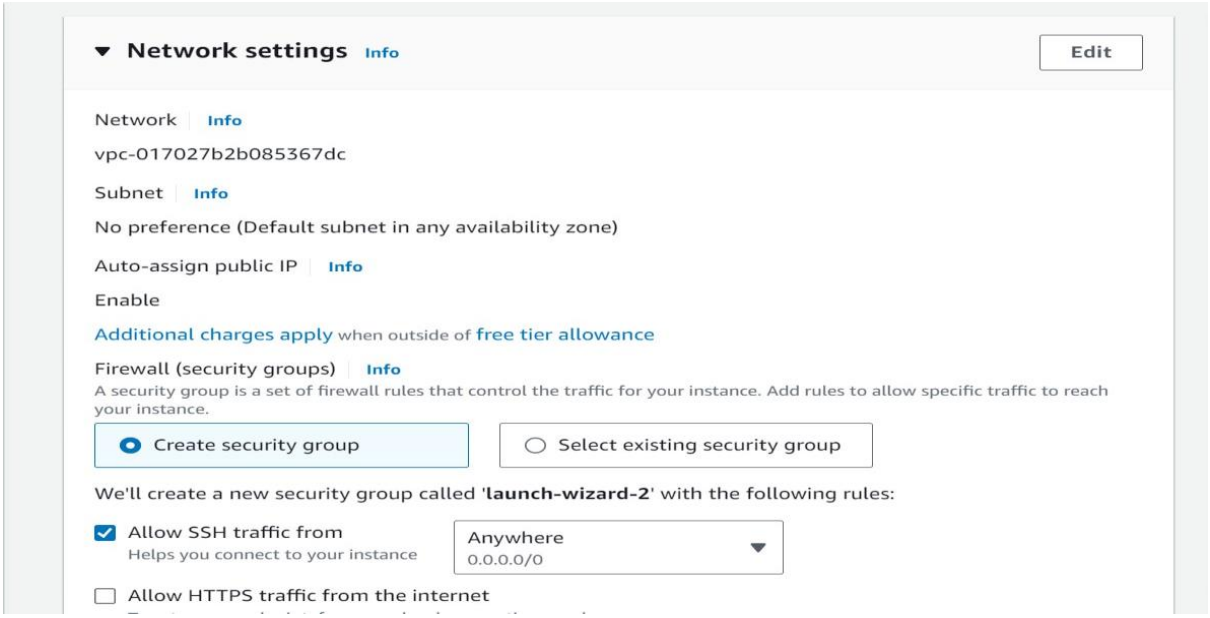

EC2 > ... > Launch an instance
Launch an instance Info
Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.
Name and tags Info
Name
online_data
Add additional tags
Application and OS Images (Amazon Machine Image) Info
An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below
Search our full catalog including 1000s of application and OS images
Recents
Quick Start
Summary
Number of instances Info
1
Software Image (AMI)
Amazon Linux 2023 AMI 2023.5.2...read more
ami-097c5c21a18dc59ea
Virtual server type (instance type)
t3.micro
Firewall (security group)
New security group
Storage (volumes)
1 volume(s) - 8 GiB
Free tier: In your first year includes 750 hours of t2.micro (or
Cancel
Launch instance
Review commands


Amazon Linux
aws
macOS
Mac
Ubuntu
ubuntu
Windows
Microsoft
Red Hat
Red Hat
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community
Amazon Machine Image (AMI)
Amazon Linux 2023 AMI
Free tier eligible
ami-02b49a24cfb95941c (64-bit (x86), uefi-preferred) / ami-04ad8c7fcc828fad4 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs
Description
Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.
Architecture
Boot mode
AMI ID
64-bit (x86)
uefi-preferred
ami-02b49a24cfb95941c
Verified provider

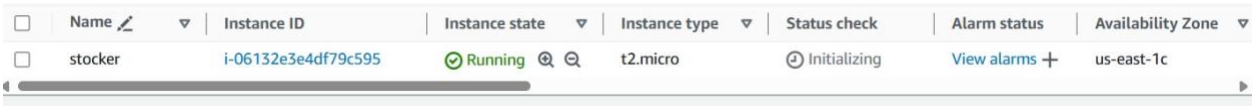


Activity 4.2: Configure Network Settings

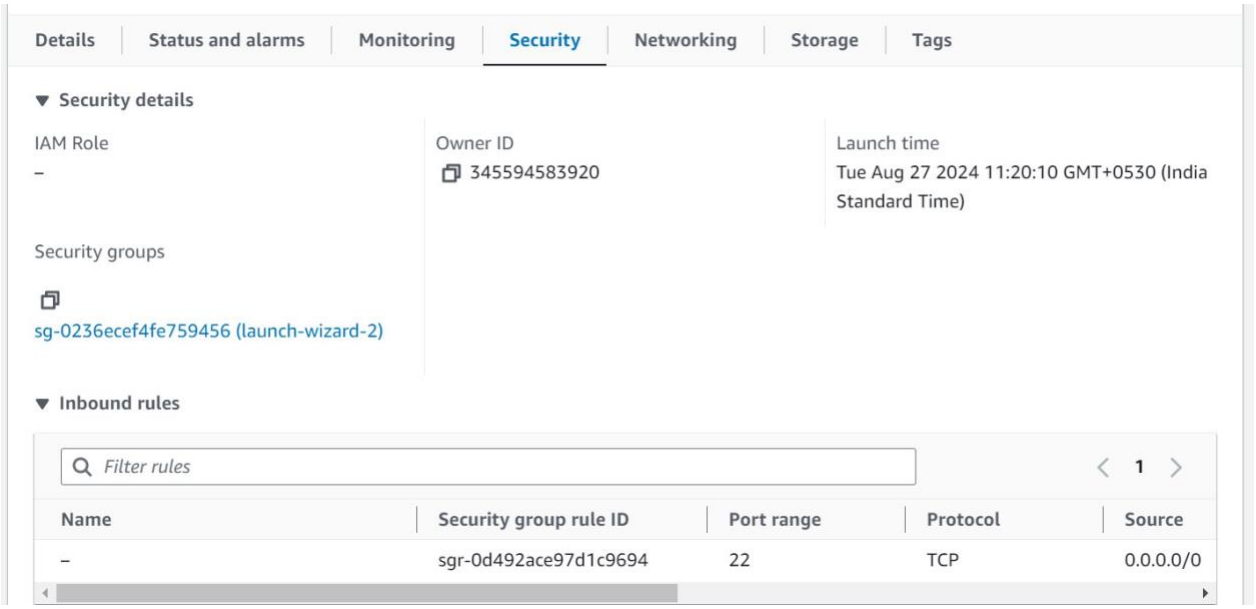
Set up the security group to allow HTTP, HTTPS, and SSH traffic.

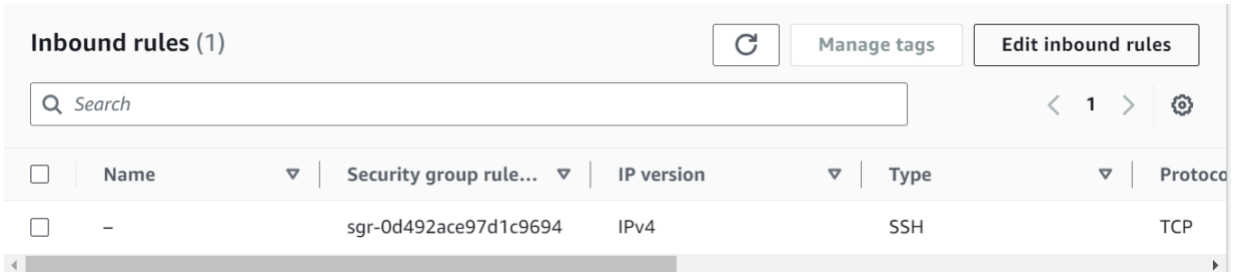


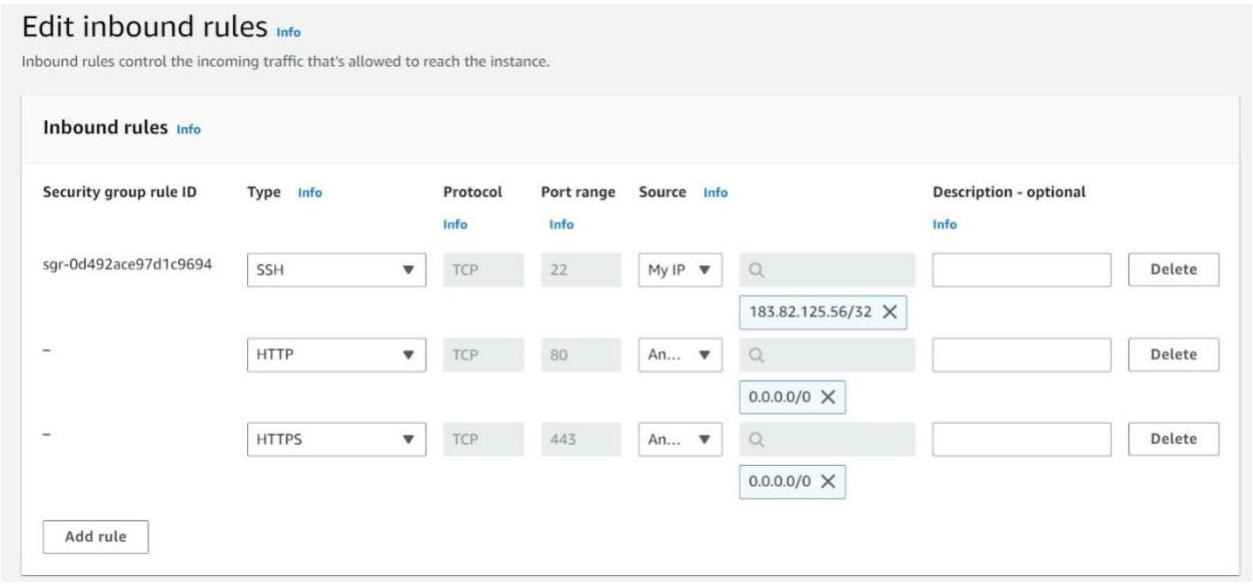
- Create and download the key pair for SSH access.



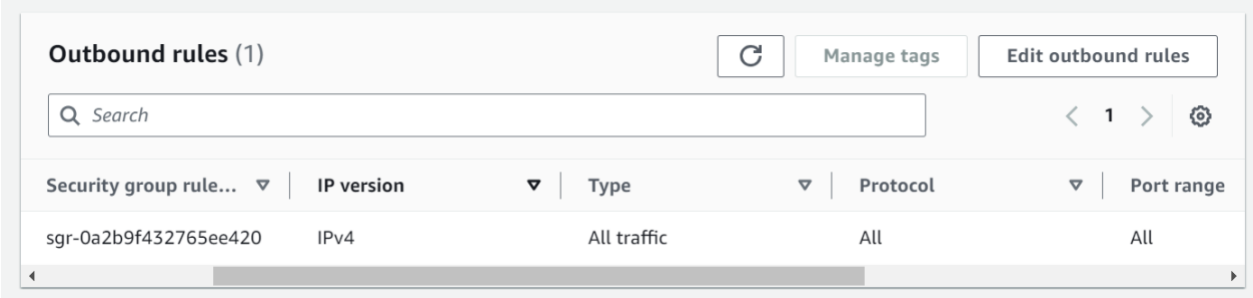
Setting up Inbound and Outbound rules







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

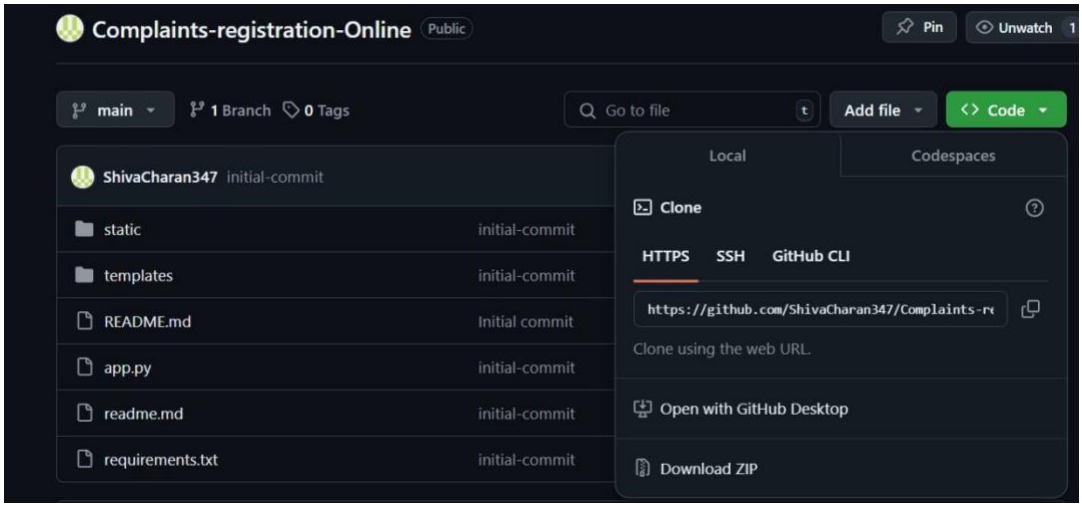


Milestone 5: Testing and Deployment

Activity 5.1: Deploy to EC2

1. Transfer your application code to the EC2 instance.
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
7. sudo yum install git
8. export DB_USER=your_db_username
9. export DB_PASSWORD=your_db_password

10. export DB_HOST=your_rds_endpoint
11. git clone git-repo-link



12. sudo yum install pip
13. pip install flask flask_login flask_sqlalchemy werkzeug pymysql
14. cd your-dir
15. python3 app.py

- **Activity 5.2 : Launch Flask Application**
  - Run the Flask app on the EC2 instance

Milestone 6: Testing and Deployment

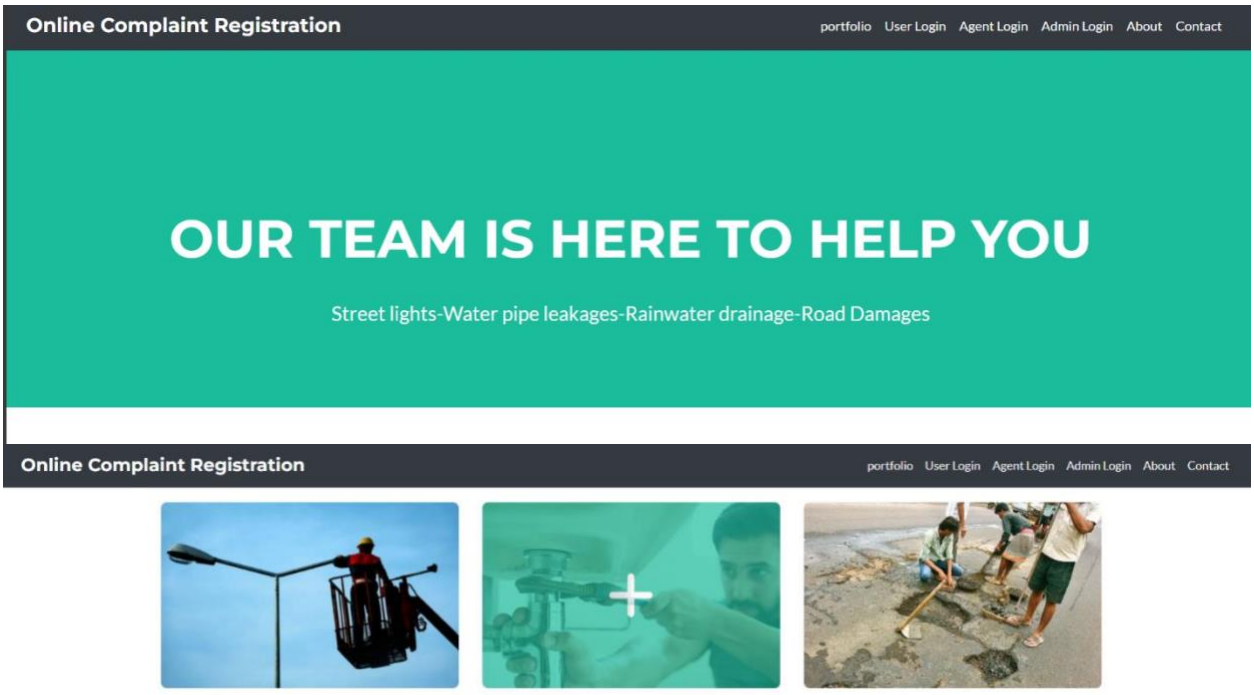
- **Activity 6.1: Functional Testing**
  - Test the Telesupport Hub application for functionality, including database interactions and frontend features.
  - Run the Flask app **python3 app.py**
  - It will give you the link

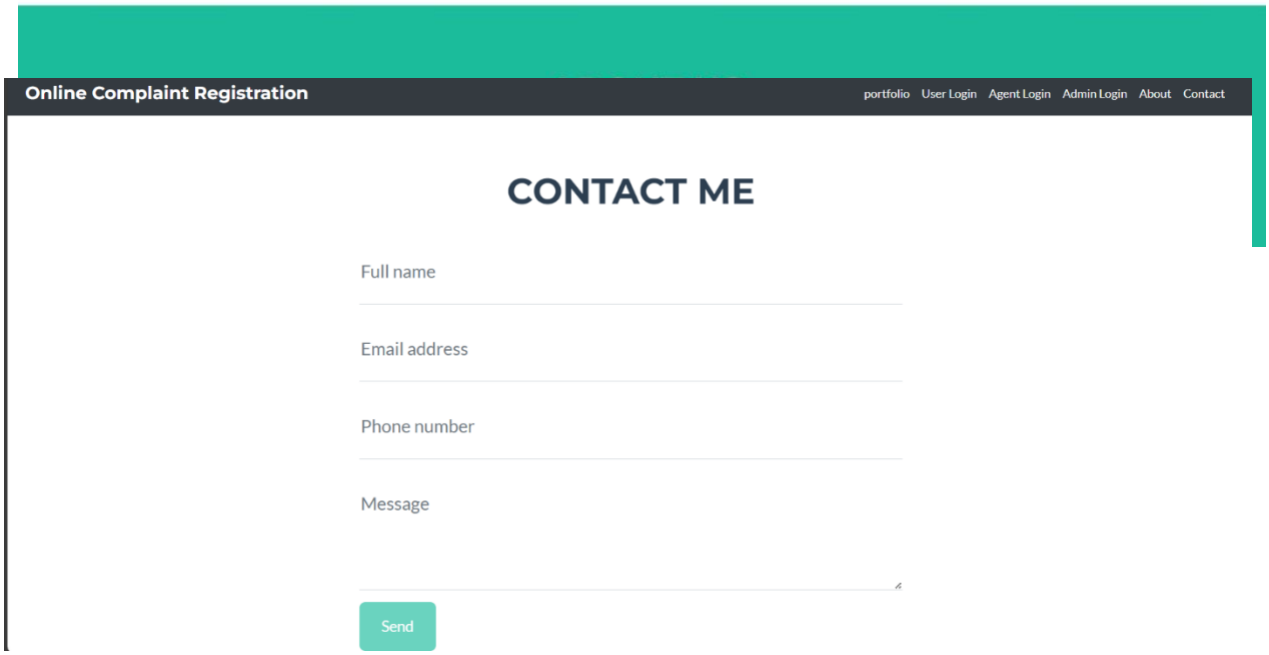
```
Serving Flask app 'app'
Debug mode: on
RNING: This is a development server. Do not use it in a production deployment. Use a p
Running on http://127.0.0.1:5000
ess CTRL+C to quit
Restarting with stat
Debugger is active!
Debugger PIN: 968-863-937
```

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

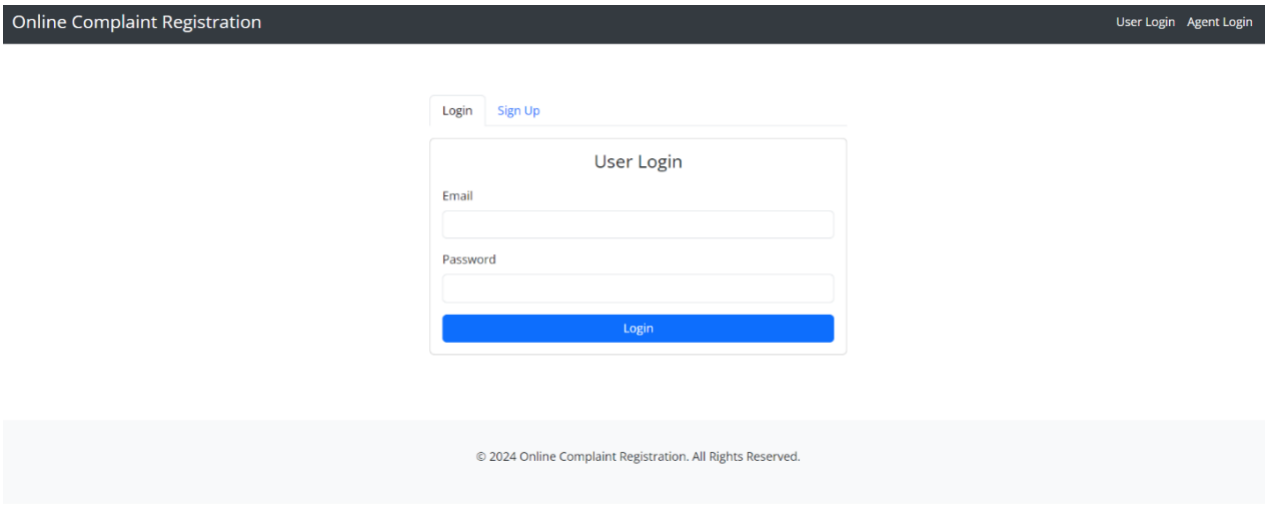
Click on the link above and it will take you to the webpage:

home:

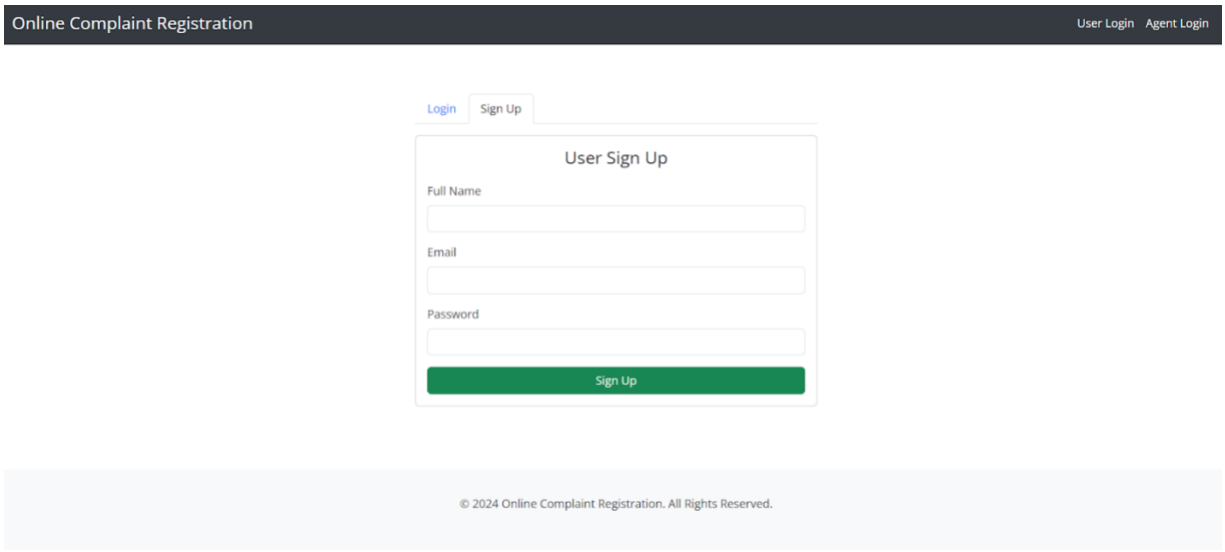


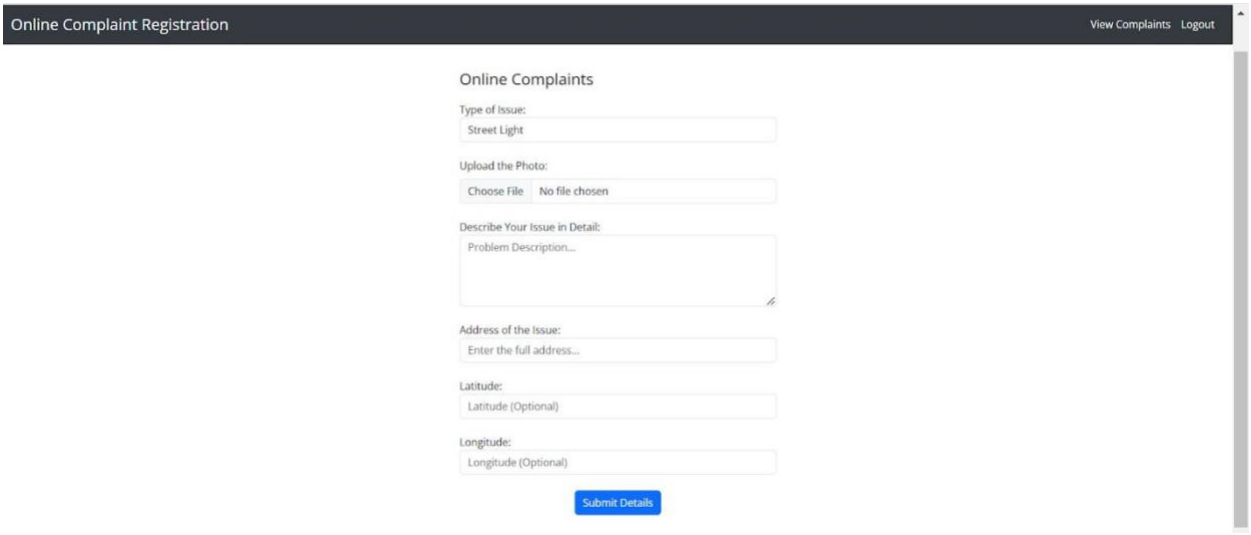


User Login:

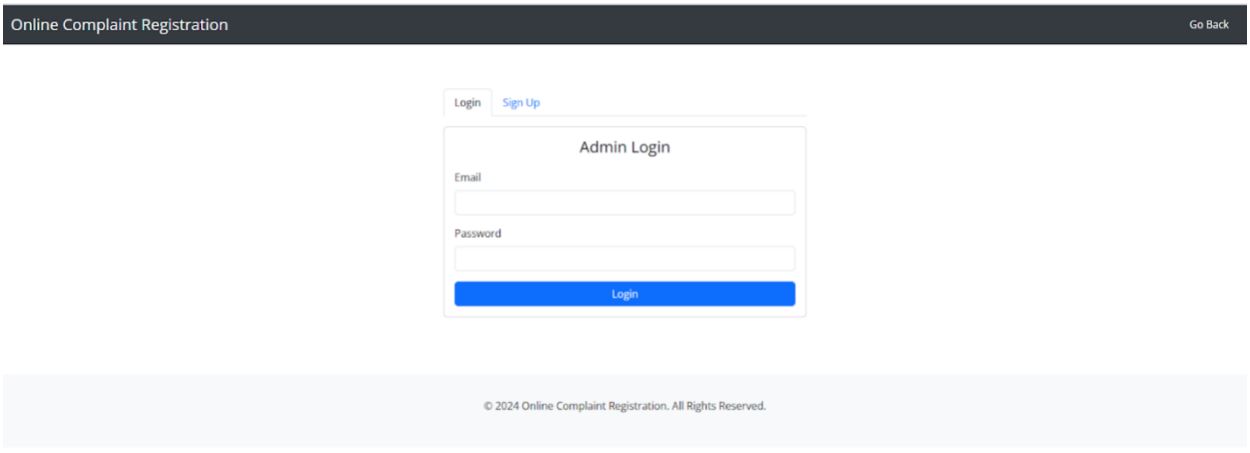


Registration:

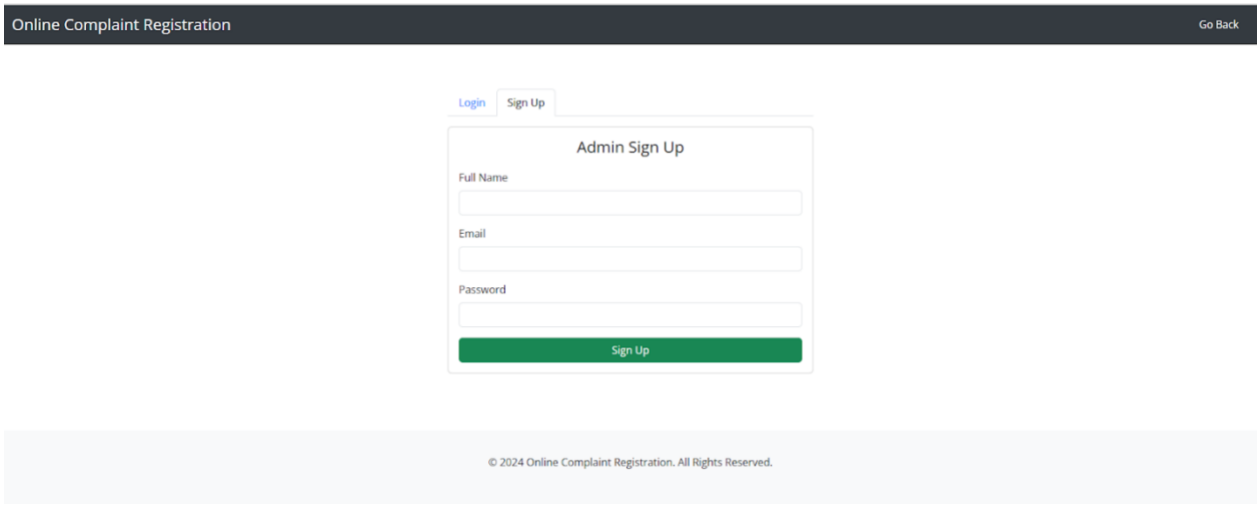




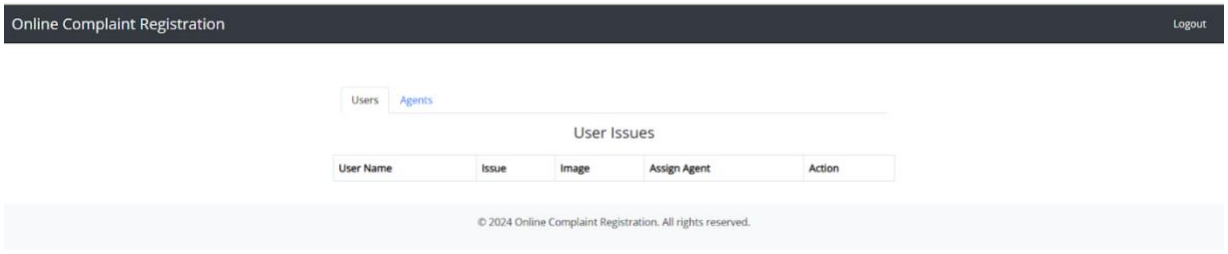
Admin Login:

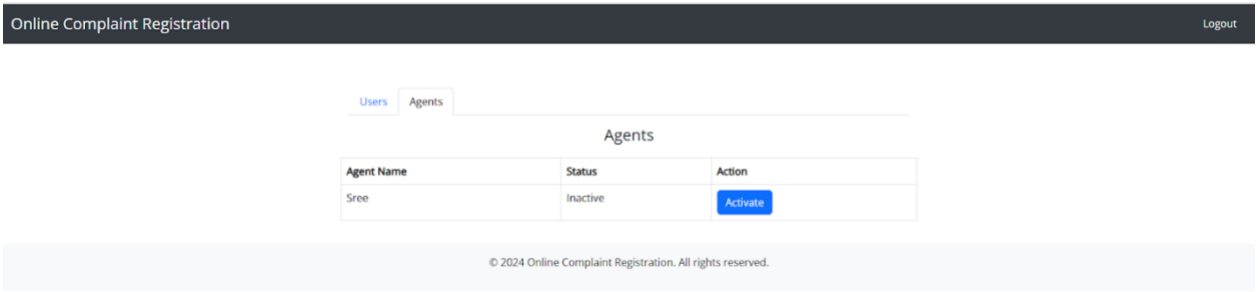


Registration:

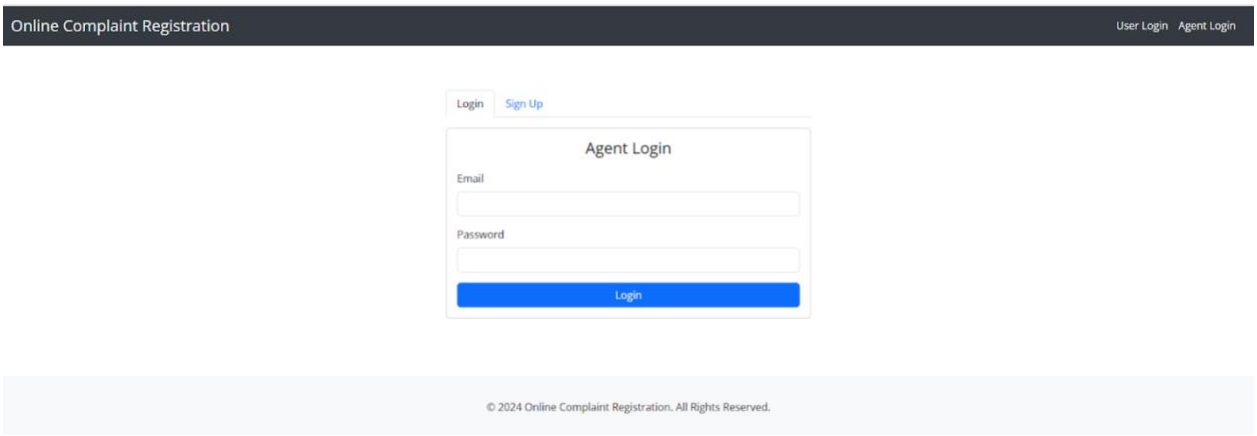

Online Complaint Registration
Go Back
Login
Sign Up
Admin Sign Up
Full Name
Email
Password
Sign Up
© 2024 Online Complaint Registration. All Rights Reserved.

--

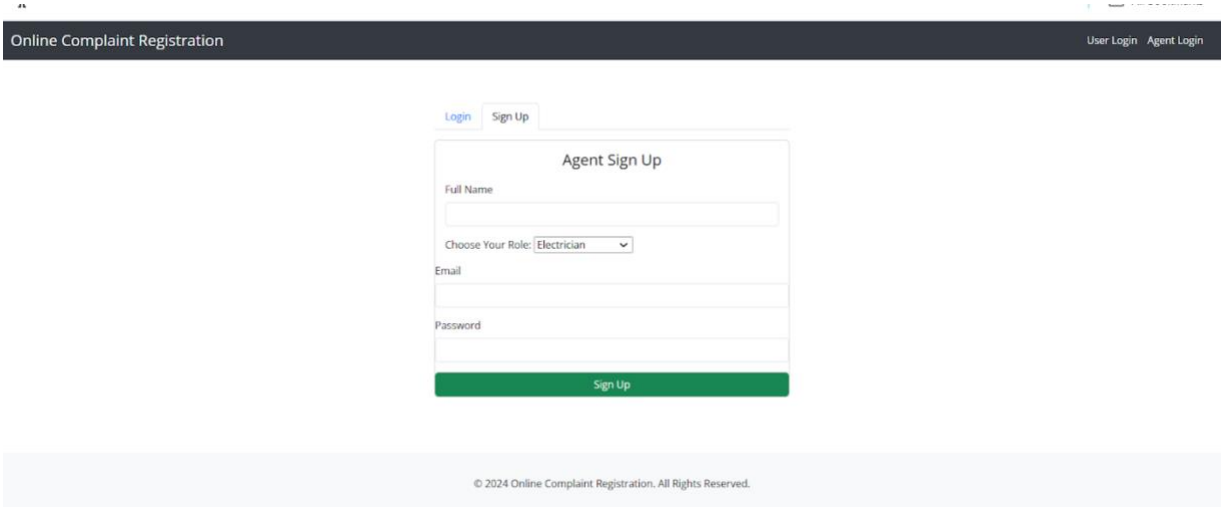

Online Complaint Registration
Logout
Users
Agents
User Issues
User Name
Issue
Image
Assign Agent
Action
© 2024 Online Complaint Registration. All rights reserved.

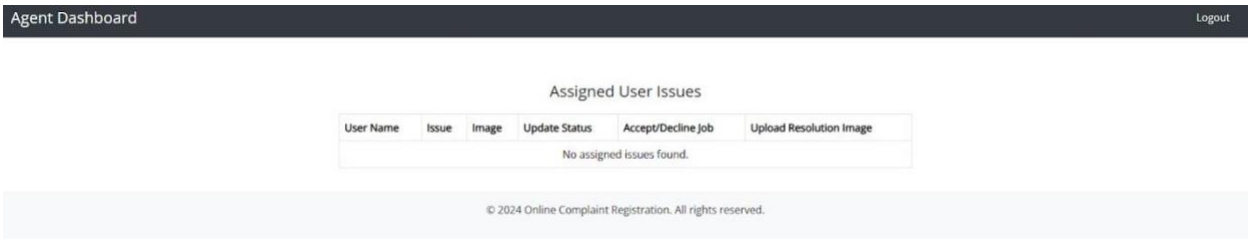


Agent login:



Registration:





Milestone 7: Monitoring and Optimization

- **Activity 7.1: Performance Monitoring**
  - Set up AWS CloudWatch for monitoring EC2 and RDS performance metrics.
  - Implement alerts and notifications for critical performance thresholds.
- **Activity 7.2: Optimization**
  - Optimize the server and database configurations based on monitoring results, including adjusting instance types and query optimization.

Conclusion:

The **Online Complaints Management and Registration** project exemplifies the development of a robust and scalable Complaint Management using AWS. By leveraging Flask alongside MySQL RDS, it provides efficient data management and secure backend operations. The deployment on AWS EC2 allows for flexible scaling and reliable performance. Each phase, from database configuration to frontend design and deployment, was meticulously executed to ensure a high-quality user experience. Additionally, AWS CloudWatch is utilized to monitor application performance and maintain operational efficiency. Overall, the project showcases how AWS services can effectively support dynamic and scalable customer service solutions in the telecom industry.