Citizen AI- Intelligent Citizen Engagement Platform

Project Documentation

1. Introduction:

● Project title :CITIZEN AI – INTELLIGENT CITIZEN ENGAGEMENT PLATFORM
● Team member : BHUVANESHWARAN . K
● Team member : DAVID RAJA . Y
● Team member : DIVYADHARSHAN . B
● Team member : SAMUEL . C

2. Project overview:

- **Purpose:**

To revolutionize government-citizen interaction by providing a 24/7 AI-powered platform for information access, feedback submission, and sentiment analysis,thereby improving transparency, efficiency, and public trust.Citizen Sentiment Analysis in Citizen AI is a core feature designed to understand the public's feelings about government services and related topics.

Citizen AI was developed to solve these challenges by introducing a real-time AI assistant capable of:

- Answering citizen questions with human-like responses using IBM Granite models.
- Analyzing citizen feedback to detect sentiments such as satisfaction or dissatisfaction.
- Presenting actionable insights on a dynamic dashboard.

Enhancing government responsiveness, transparency, and public trust through AI-powered interactions

➢ Features:

Conversational Interface:- Real-time Q&A about government services.

Sentiment Analysis: Automatically classifies citizen feedback as Positive, Neutral, or Negative.
Dynamic Dashboards: Provides both citizens and government officials with visual insights from the collected data.
User Authentication: Separate login systems for citizens and government administrators.
Feedback & Concern System: Allows citizens to report issues and provide feedback seamlessly.

3.Key Points:

➢ Problem Statement

- Citizens face delays in receiving information or reporting issues.
- Feedback collected by government departments is rarely analyzed in real time.

➢ Proposed Solution

- **Citizens:** To ask questions, provide feedback, and get quick answers.
- **Government Officials:** To monitor public sentiment and improve services.
- **Policy Makers:** To use data insights for better governance decisions.
- **Developers/Admins:** To manage system operations and improve AI performance.

➢ Target Users

- **City Residents** – To report civic issues easily and get eco-advice.
- **City Administrators –** To monitor feedback, analyze KPIs, and respond to anomalies.

- **Urban Planners –** To summarize long documents and make informed policy decisions.
- **Teachers & Students –** To explore sustainability practices via the Eco Tips Generator.
- **Government Departments** – For utility monitoring and forecasting resource demands

.

➢ Expected Outcomes

- A responsive, intelligent chatbot available 24/7 for public queries.
- Real-time analysis of citizen feedback to detect trends and issues.
- Dashboards that visualize public mood and interaction frequency.
- Enhanced digital governance through AI, leading to improved public satisfaction and trust.

4. Folder Structure

```
CitizenAI/
├── app.py # Main Flask application and core logic
├── requirements.txt # Python dependencies
├── .env # Environment variables (ignored by git)
├── citizen_ai.db # SQLite database (created automatically)
├── 3.py # Script to create a government admin user
│
├── static/ # Static files (CSS, JS, Images)
│ └── styles.css # (Your base.html expects this file)
│
└── templates/ # HTML Templates
├── index.html # Landing page
├── login.html # Login page with citizen/govt tabs
├── register.html # User registration page
├── dashboard.html # Citizen personal dashboard
├── admin_dashboard.html # Government admin dashboard
├── chat.html # AI conversation interface
├── feedback.html # Feedback submission page
├── about.html # Project information page
├── services.html # (Referenced, ensure it exists)
└── base.html # Base template
```

To start the project:

1. Ensure prerequisites are met.
2. Install dependencies via `pip install -r requirements.txt`.
3. Configure your `.env` file with API keys.
4. Execute `python app.py`.
5. Navigate to `http://localhost:5001` in your browser.

5.Technical Requirements:

The technical implementation of Citizen AI relies on modern, scalable, and secure architecture. The backend is built using Python with the Flask framework, allowing for modular API routing and clean project structure. The frontend is implemented using HTML, CSS, and JavaScript to create responsive user interfaces, including the chatbot, feedback form, and analytics dashboard. AI functionality is powered by IBM Granite, which handles natural language understanding and response generation. The system connects to a relational database such as SQLite or PostgreSQL to store user feedback, sentiment scores, and session logs. Configuration settings, including API keys and database URIs, are securely managed using .env files and a centralized config.py module. To ensure reliability and performance, the application is expected to maintain low response latency (under 3 seconds per query) and support concurrent users through optimized backend routes and efficient frontend rendering. Basic security measures like input validation and environment-based configuration are implemented to protect user data and prevent vulnerabilities. The system also includes provisions for unit and integration testing to maintain code quality and long-term maintainability.

Key points:

➢ Technical Requirements:

- Backend developed using **Flask (Python)** with RESTful routing.
- Frontend created using **HTML, CSS, and JavaScript** for interactive UI.
- AI integration handled via **IBM Granite API** for NLP and chat responses.
- Database setup using **SQLite or PostgreSQL** for structured data storage.
- Sensitive data and configuration managed through .env and config.py.

- Chat latency kept under **3 seconds** per response for smooth UX.
- Scalable and modular design to support multiple users and future features.
- Input sanitization and secure API key handling for data protection.
- Unit and integration tests included to ensure quality and reliability.

➢ Functional Requirements:

- Enable real-time chatbot interaction using IBM Granite.
- Accept citizen queries and provide human-like responses.
- Allow feedback submission through a web form interface.
- Automatically analyze sentiment (Positive, Neutral, Negative) from feedback.
- Display sentiment insights and user trends on a dynamic dashboard.
- Provide personalized and context-aware replies based on session history.
- Store all user queries, feedback, and sentiment results in a secure database.

6.LLM Integration (IBM Watsonx Granite):

➢ **Model:**

WatsonxLLM` from the `langchain_ibm` library.

➢ **Functionality:**

Processes user queries in natural language and generates informative, context-aware responses specific to Indian government services. The integration includes a fallback mechanism to ensure responses are always provided.

ML Modules:

➢ **Sentiment Analysis Module:**

A rule-based classifier (`analyse_sentiment` function) that uses keyword matching (lists of positive/negative words) to determine the sentiment of user feedback. It returns a sentiment label (Positive, Neutral, Negative) and a confidence score.

- **Data Management:**

Database: SQLite (for development and demonstration purposes).

Tables:

- users: Stores citizen and government user credentials and profiles.
- sessions: Manages user login sessions.
- chat_history: Logs all AI conversations.
- sentiment_analysis: Stores all submitted feedback and its classified sentiment.

7.Architecture

The system follows a client-server model with a clear separation of concerns.

- **Frontend:-**

Technology: HTML5, CSS3, JavaScript

Framework: Jinja2 Templating Engine (integrated with Flask)

Libraries: Chart.js for data visualization, Font Awesome for icons.

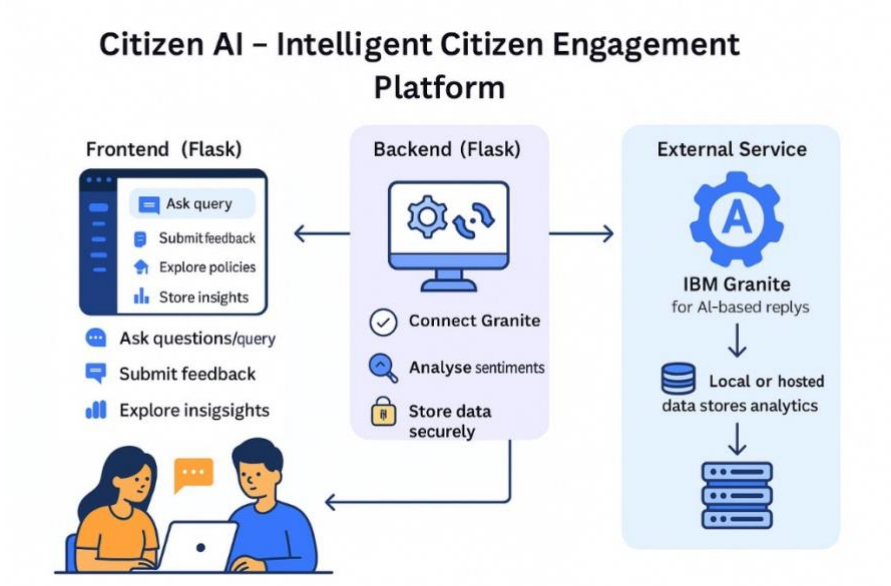
Purpose: Provides the user interface for citizens to interact with the AI assistant and for

government officials to view analytics. The frontend is served dynamically by the Flask backend.

- **Backend:**

Technology: Python
Framework: Flask (Web Server Gateway Interface)
Purpose: Handles HTTP requests, serves web pages, manages user sessions, processes form data, and contains all application logic.

➢ System Architecture Diagram:



➢ User flow:

- A **citizen** opens the platform and interacts with the chatbot by typing a query.
- The message is sent to the backend, where IBM Granite processes it and returns a response.
- The citizen may also submit feedback using a form, which is analyzed for sentiment and stored in the database.
- An **admin** logs into the dashboard to monitor overall sentiment trends and feedback analytics.
- The system tracks session history, enabling context-aware, more personalized responses for returning or active users.

8.User Interface

The UI is modern, responsive, and designed with a government-service aesthetic (using blue
color schemes, official badges)

Key pages include:

- **Landing Page (`index.html`):**
  Promotional introduction with calls-to-action.
- **Login/Register Pages:**

Clean forms with password strength indicators and separate tabs for citizen/government login.

- **Citizen Dashboard:**

Personal overview of feedback history, sentiment stats,and recent chats.

- **Government Dashboard:**

Admin panel showing all users, all feedback, and advanced sentiment analytics with charts.

- **Chat Interface:**

A simple chat-style interface for interacting with the AI.

- **Feedback Page:**

A form for submitting textual feedback.

Key Points:

➢ **Technology Stack Used:**

- **Backend Framework:** Python with Flask (for modular API routing)
- **Frontend Technologies:** HTML, CSS, JavaScript (with Bootstrap)
- **AI Integration:** IBM Granite LLM (for chatbot and contextual responses)
- **Sentiment Analysis:** Custom Python module using NLP techniques
- **Database:** SQLite (development) / PostgreSQL (production-ready)
- **Configuration & Secrets:** .env file and config.py for environment-based setup
- **Version Control:** Git and GitHub
- **Deployment (Optional):** Docker, cloud hosting options

➢ Development Process:

The development of Citizen AI followed a structured and modular approach. The project was divided into independent yet connected components, ensuring flexibility and easier debugging. Each major feature was implemented and tested step by step, starting from backend setup to AI integration and frontend user experience. Below are the key stages of the development process:

➢ **Flask Project Initialization:**

- Created the basic Flask structure with app.py, config.py, and .env for secure settings.
- Installed necessary dependencies including Flask, IBM Granite SDK, and SQLAlchemy.

➢ **Chatbot Module Implementation:**

- Developed chat_routes.py to handle incoming chat messages.
- Integrated IBM Granite API through granite_interface.py to fetch AI responses.
- Designed chat.html and chat.js for real-time user interaction on the frontend.

➢ **Feedback & Sentiment Analysis:**

- Created a feedback form and handled submissions via feedback_routes.py.
- Analyzed sentiment using the analyse_sentiment() function in sentiment_analysis.py.
- Stored feedback and sentiment data into the database using SQLAlchemy models.

➢ **Dashboard Development:**

- Built dashboard_routes.py to serve sentiment and feedback metrics.
- Designed dashboard.html with embedded charts for real-time analytics using JavaScript.
- Displayed sentiment counts and citizen engagement trends for admin users.

➢ **Contextual Response Enhancement:**

- Implemented session tracking to maintain chat context.

- Used helpers.py to store previous interactions and send enriched prompts to Granite for more relevant responses.

➢ **Database Integration:**

- Defined all database schemas (User, Feedback, Sentiment) in models.py.
- Created init_db.py for easy initialization and reset during testing.
- Connected the application to SQLite/PostgreSQL for persistent data handling.

9.Challenges & Fixes:

- **API Integration Issues:**
  Initial connection to IBM Granite API failed due to incorrect headers and key usage. Fixed by properly loading environment variables and validating API structure.
- **Slow Chat Response:**
  LLM responses were delayed due to large prompts. Solved by optimizing the prompt format and reducing input token length.
- **Frontend and Backend Sync:**
  Chat UI failed to reflect real-time responses. Resolved by properly handling async calls and using consistent JSON structures.
- **Sentiment Misclassification:**
  Feedback was inaccurately tagged. Improved model accuracy by refining classification logic and preprocessing input text.
- **UI Freezing & Form Reloads:**
  Long LLM calls caused UI lag. Handled with asynchronous JS and form handling improvements.
- **Database Connection Errors:**
  Faced schema mismatches during early testing. Resolved by standardizing model definitions and resetting the database using init_db.py.

UI/UX Consideration:

The Citizen AI platform is designed with a clean, responsive, and user-friendly interface to ensure ease of access for users of all digital literacy levels. The chatbot interface provides real-time feedback using smooth interactions and toast notifications to confirm actions. The layout uses Bootstrap for responsive design, ensuring compatibility across devices such as desktops, tablets, and mobile phones.

High-contrast colors and readable font sizes are used to support accessibility, and input fields are clearly labeled for better usability. The dashboard presents key metrics through intuitive charts and minimal clutter, allowing administrators to quickly understand public sentiment and engagement. The overall design prioritizes simplicity, clarity, and functionality to enhance user engagement and trust.

10.FUNCTIONAL & PERFOMANCE TESTING

The primary objective of the Functional and Performance Testing phase is to ensure that all features of the Citizen AI platform work as intended and that the system performs reliably under expected user conditions. This phase involved testing each module individually and then in combination, checking for accuracy, responsiveness, and stability. The goal was to identify and resolve any bugs, inconsistencies, or performance bottlenecks before deployment.

Key Points:

➢ Test Cases Executed:

- **Chatbot:**
  Verified real-time responses from IBM Granite for various citizen queries.
- **Feedback Submission:**
  Checked form handling, data storage, and confirmation messages.

- **Sentiment Analysis:**
  Tested classification as Positive, Neutral, or Negative for diverse feedback inputs.
- **Dashboard:**
  Validated data visualization, sentiment graphs, and trend accuracy.
- **Contextual Chat:**
  Ensured conversation history is maintained and used for relevant replies.
- **Database:**
  Confirmed data integrity, proper schema usage, and reliable storage.
- **Performance:**
  Measured response times (<3s), smooth UI experience, and multi-user handling.

➢ Bug Fixes & Improvements:

Several issues were identified and fixed during testing:

- **Granite API Delay:** Resolved by reducing prompt size and optimizing token usage.
- **Incorrect Sentiment Labels:** Refined logic in analyse_sentiment() to improve accuracy.
- **Frontend-Backend Mismatch:** Standardized API responses to maintain consistent data formats.
- **Chart Rendering Errors:** Fixed parsing logic for feedback timestamps and values.
- **UI Freezing on Long Responses:** Implemented asynchronous handling and loading indicators.

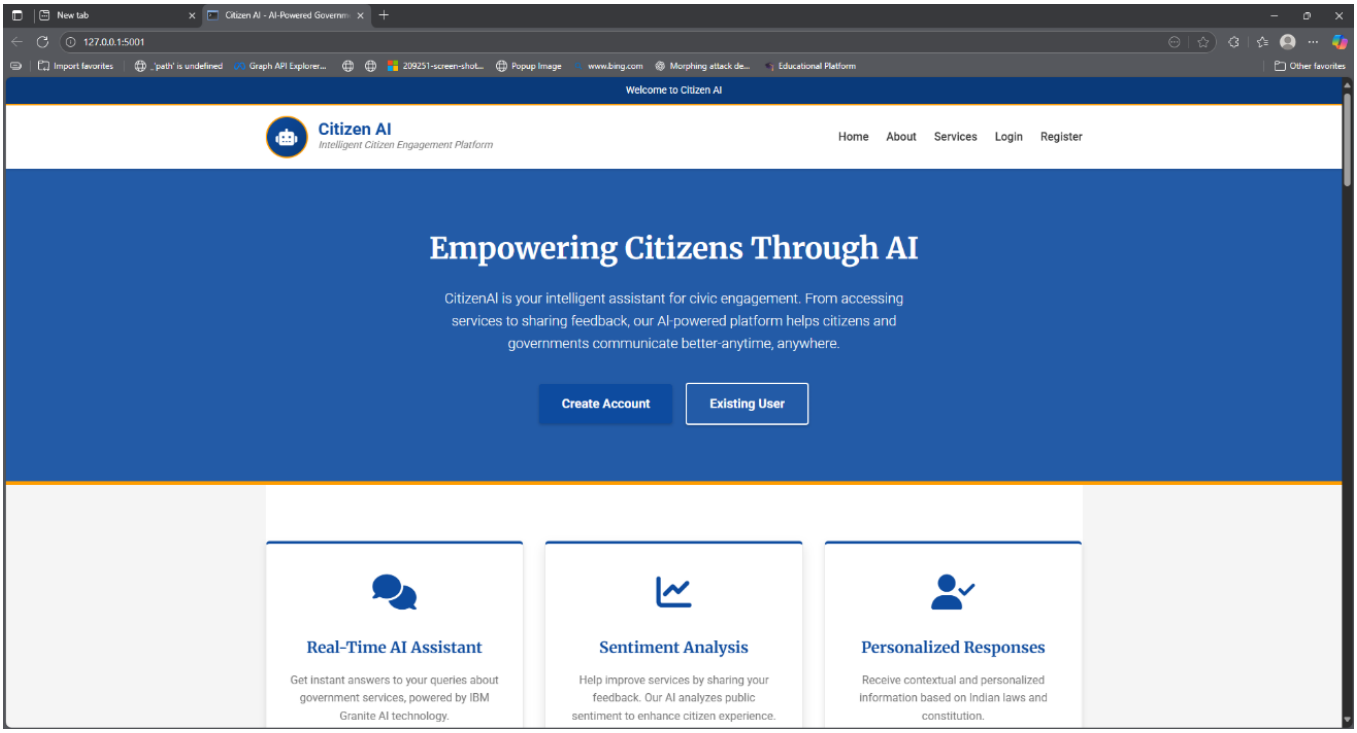
➢ Final Validation:

After thorough testing, the Citizen AI platform was confirmed to:

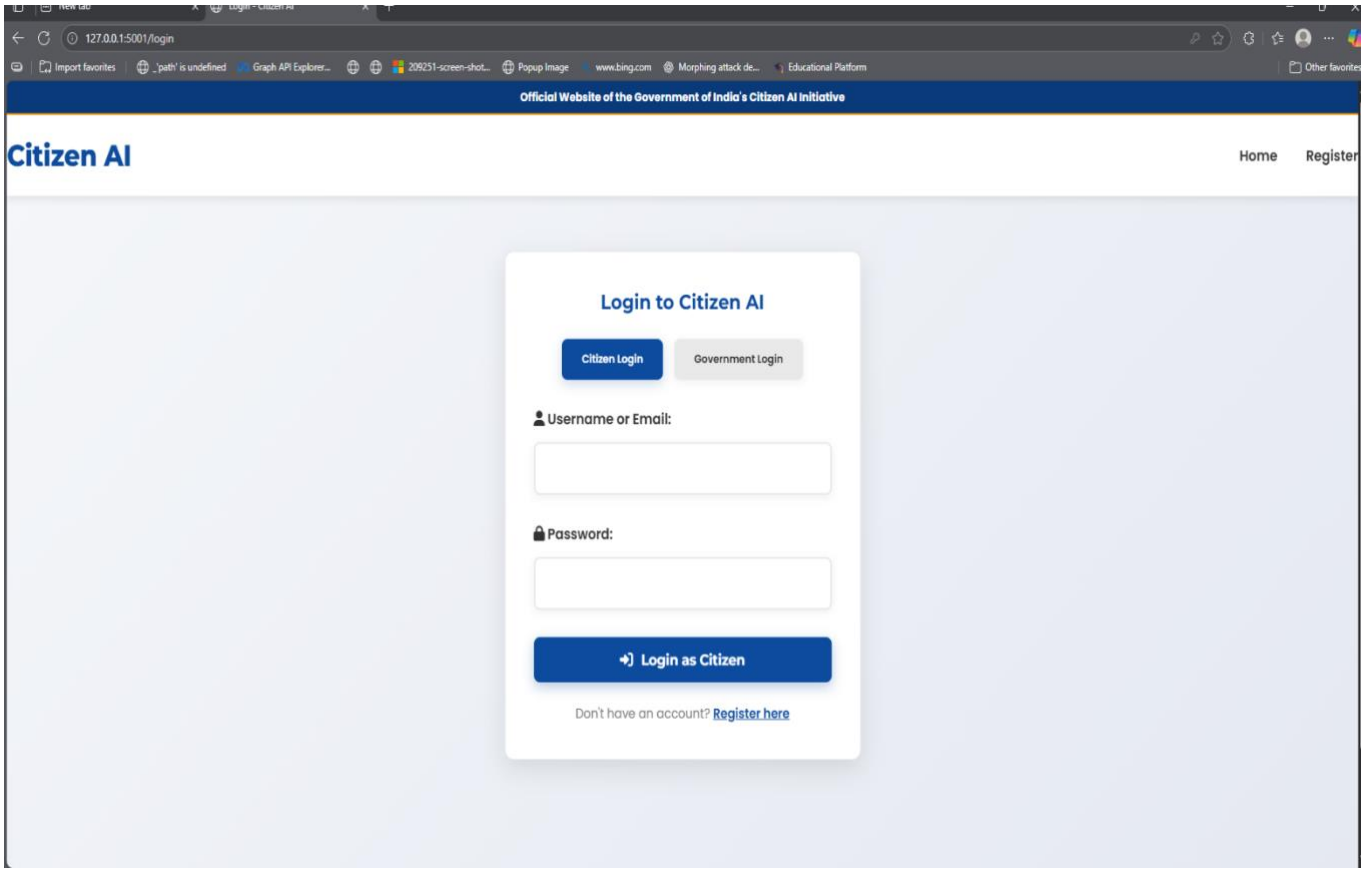
- Handle real-time citizen interaction effectively.
- Process and store feedback accurately.
- Display sentiment trends and analytics correctly.
- Respond quickly and scale to multiple users.
- Meet the functional requirements defined in the planning phase.

11.Deployment:

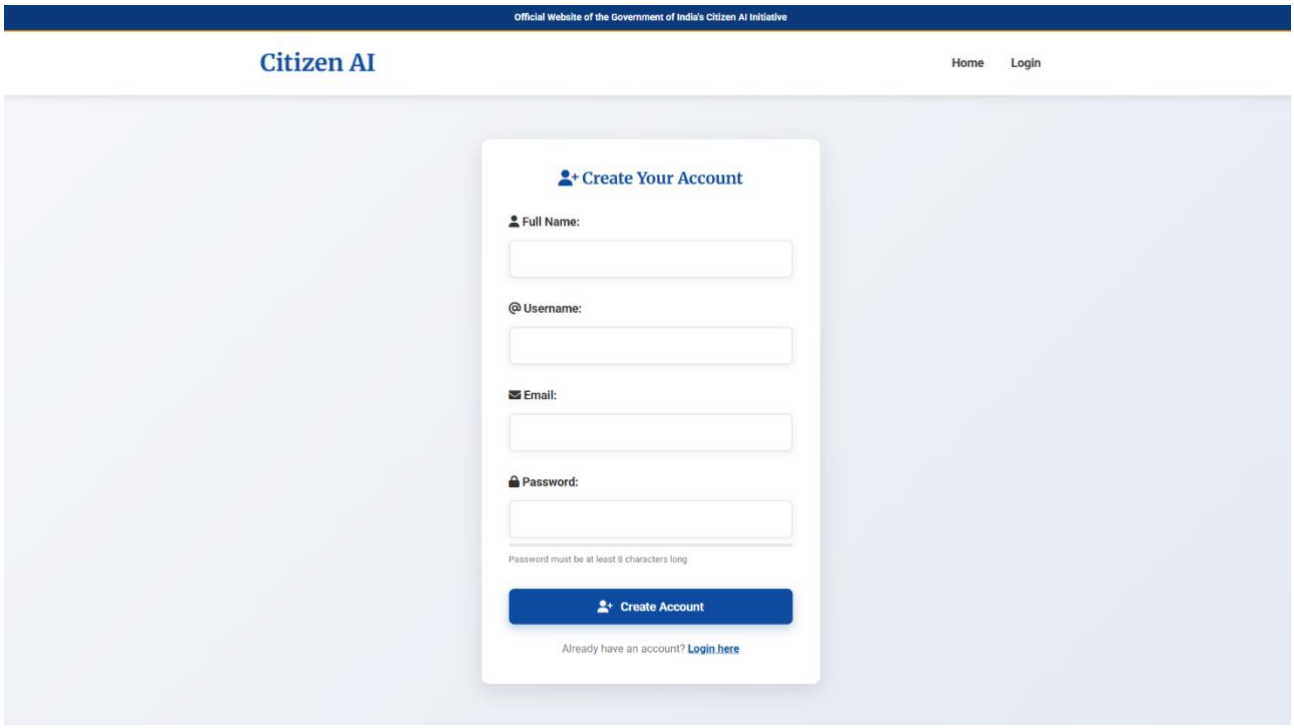
Home page:



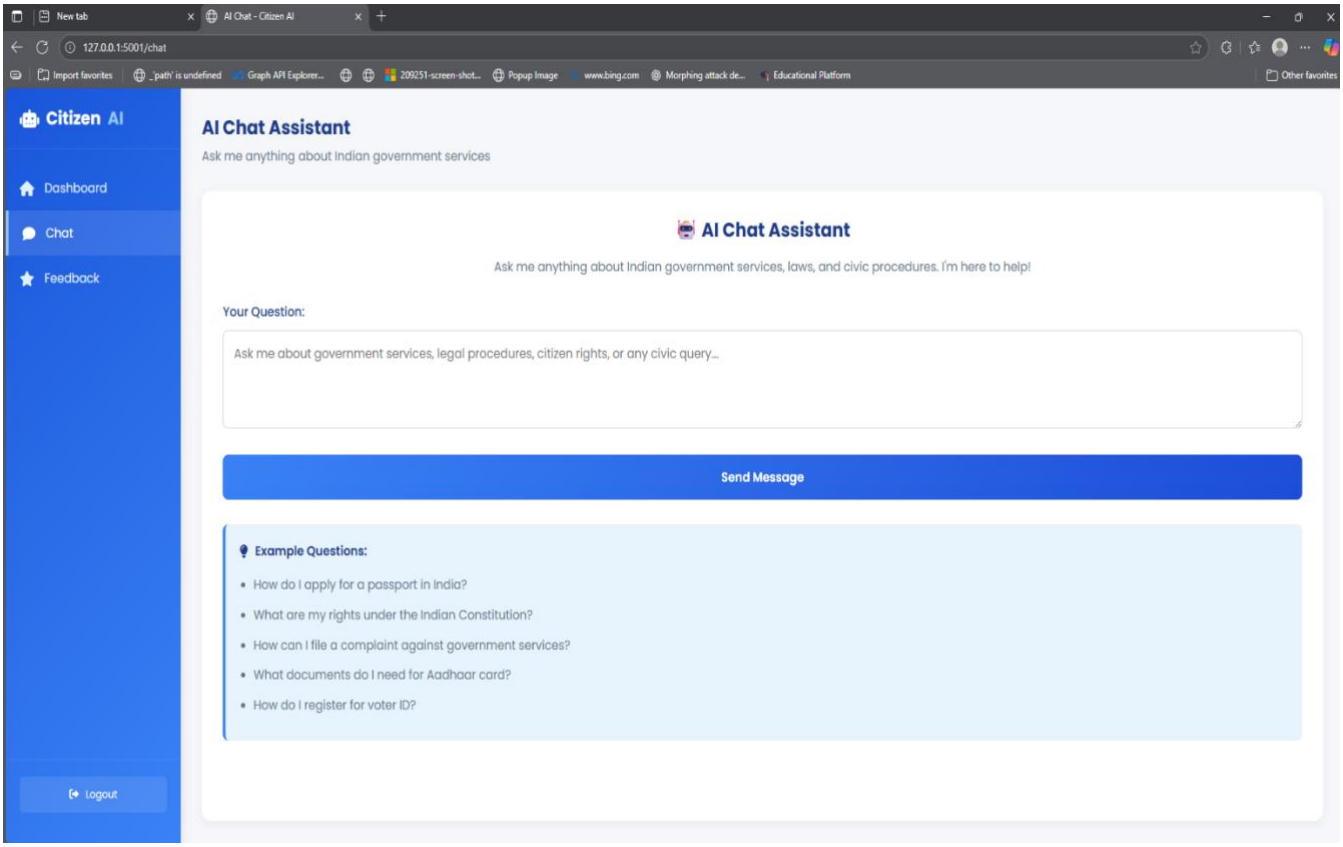
Citizen-login page:



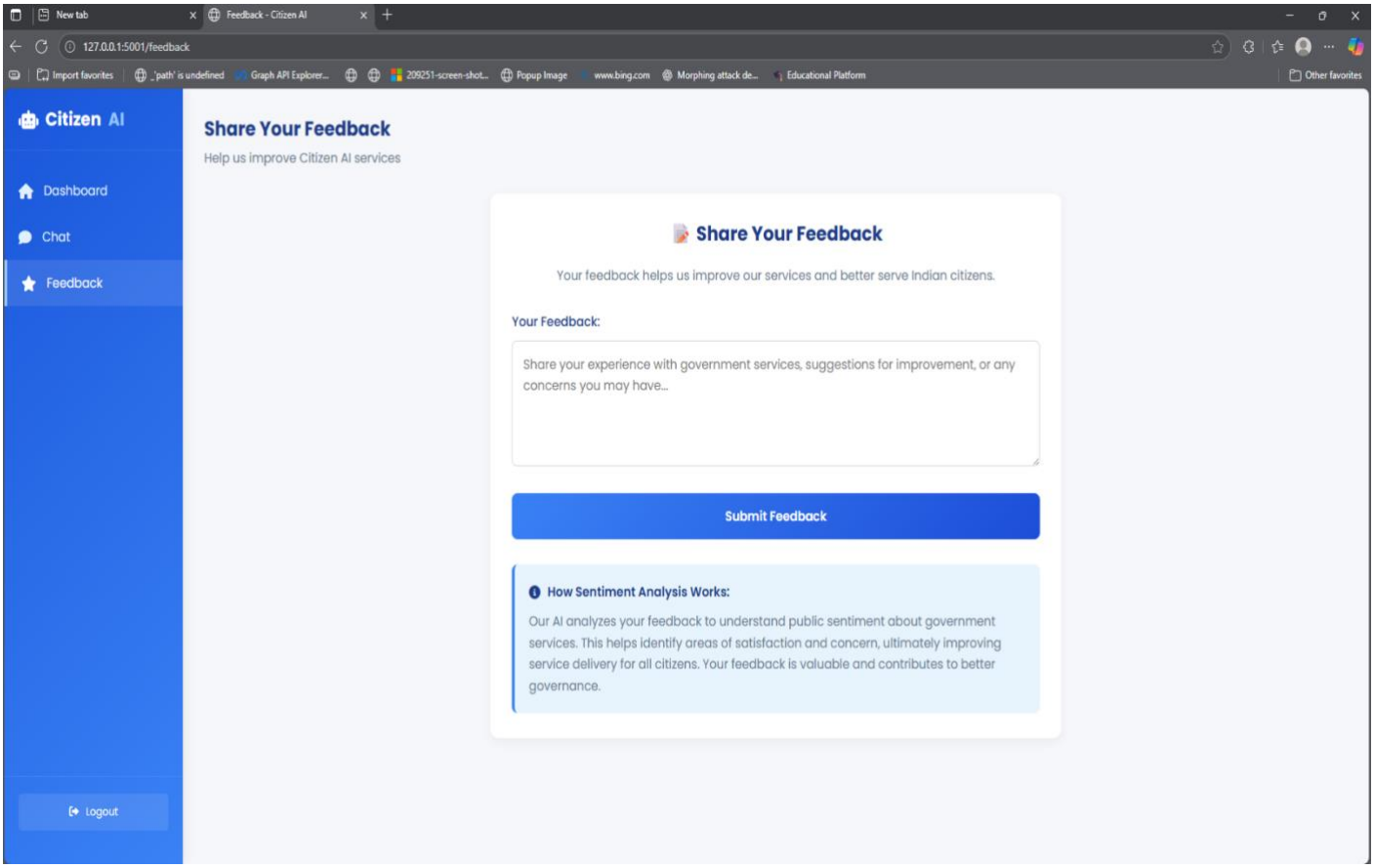
Citizen-Register page:



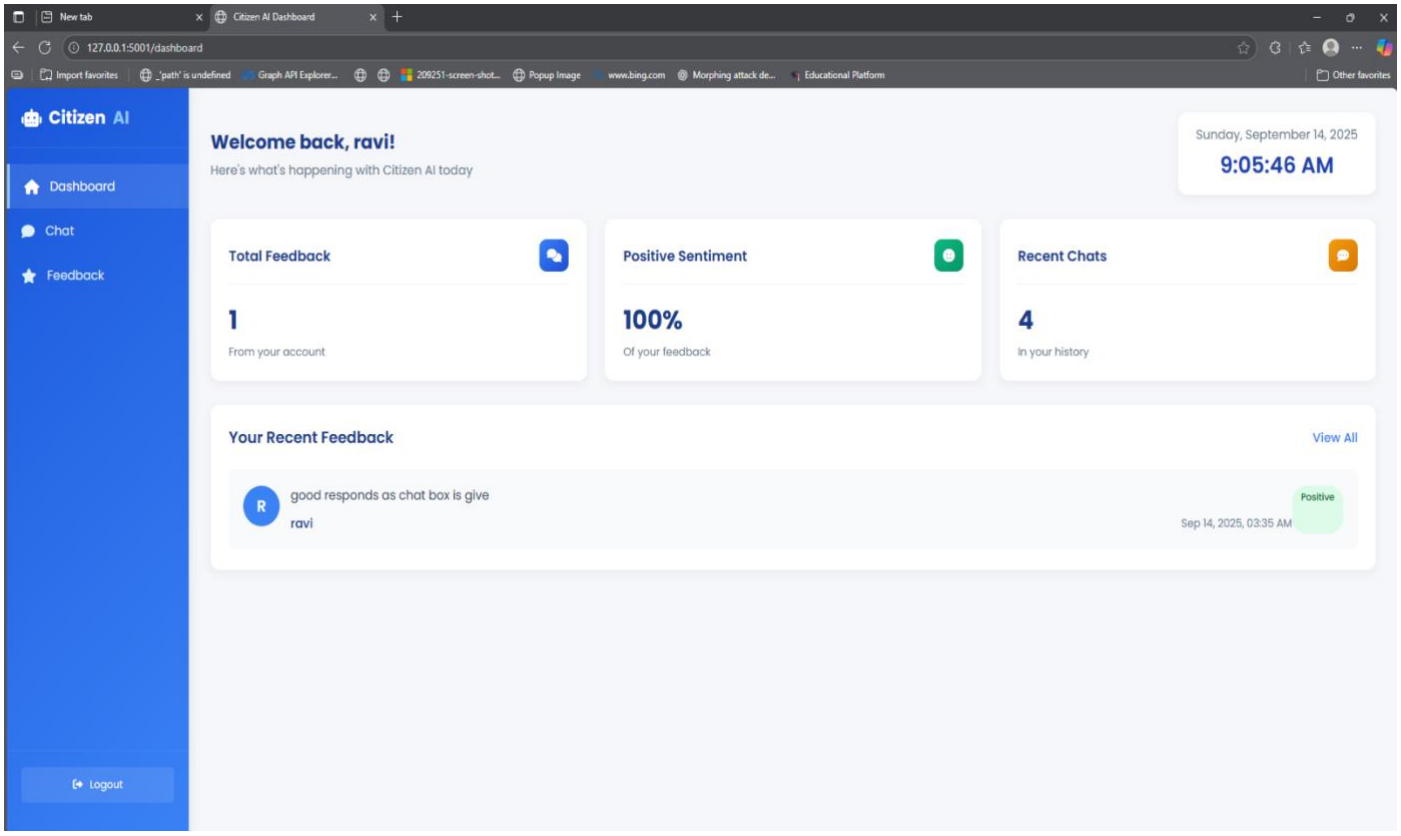
AI-Chat page:



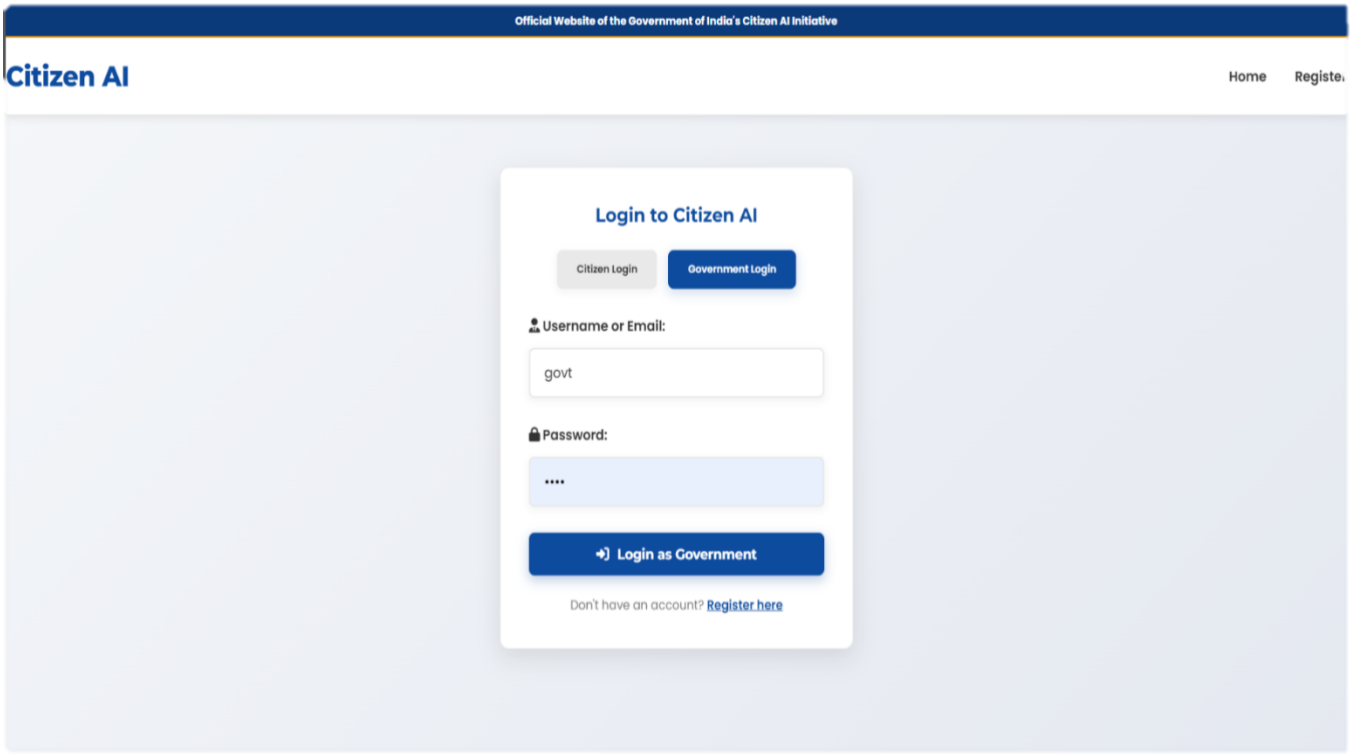
Citizen-Feedback page :



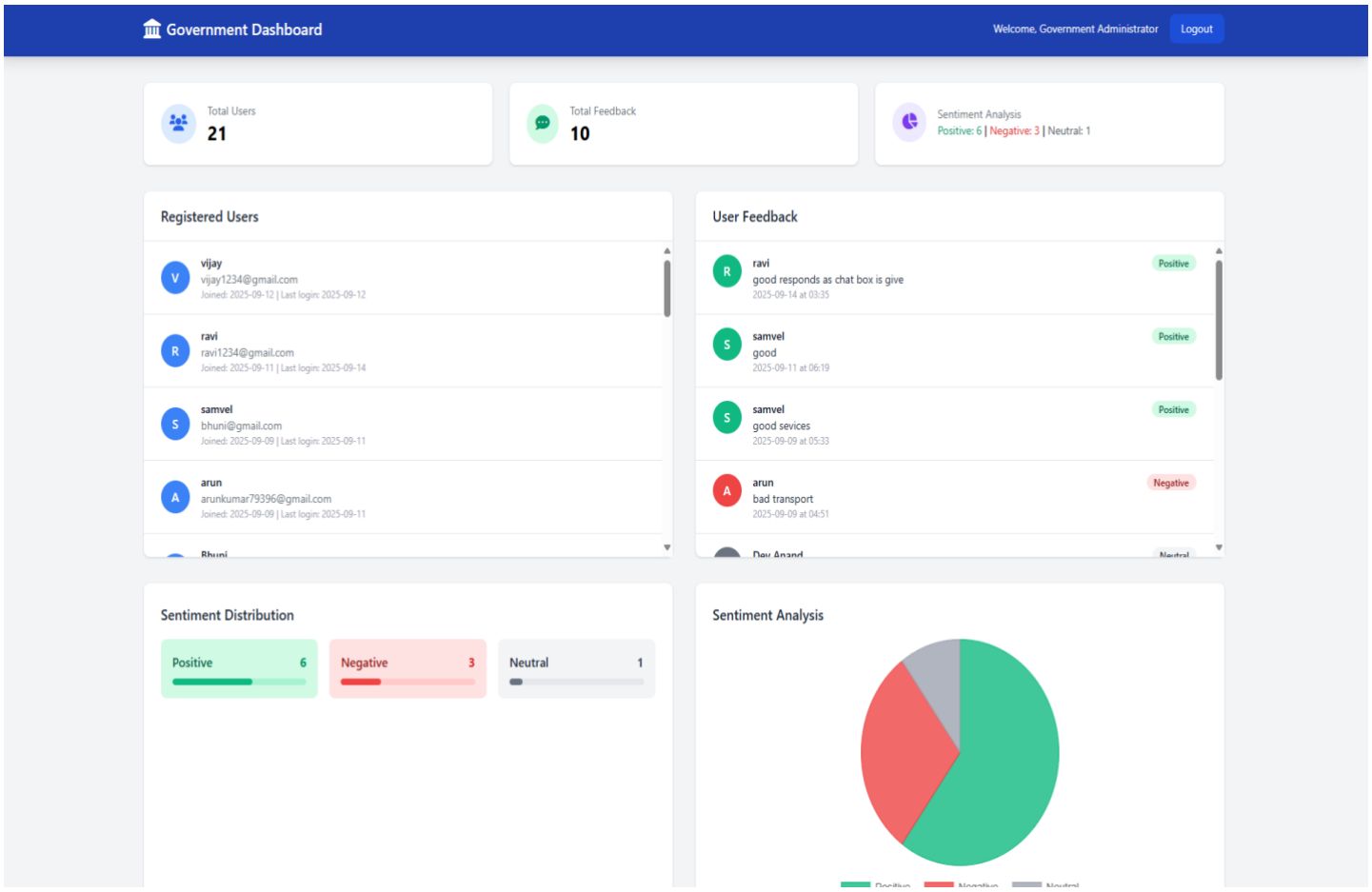
Citizen-Dashboard page:



Government-login page:



Government-Dashboard page:



12. Known Issues :

1.Model Dependency: The application cannot function without a valid IBM Watsonx setup or a working internet connection for the initial model download, despite the fallback mechanism.
2.Database: Uses SQLite, which is not suitable for production-scale deployment. Should be migrated to PostgreSQL or MySQL.

13. Future enhancement:

- SMS/Email notifications for users.
- File upload support for document analysis.
- Multi-language support for broader accessibility across India.