Citizen AI – Intelligent Citizen Engagement Platform

Project Documentation

1.Introduction:

Project Title: **Citizen AI – Intelligent Citizen Engagement Platform**

Team Members:

Team Member	Role
Palivela Abhiram (Lead)	Project Lead & Backend Developer (Flask)
Kalla Yamuna Maha Lakshmi	Full Stack Developer
Immandi Indu Priya	API Integration Specialist
Karri Shruthi Haasini	Quality Assurance & Testing

2.Project Overview Purpose:

To enable real-time, AI-powered citizen engagement with government services through intelligent chat, sentiment analysis, and feedback analytics.

Features:

- AI chatbot (IBM Watson/Granite)
- Sentiment analysis engine
- Dynamic analytics dashboard
- Feedback form and history tracker

3. Architecture:

- **Frontend:**

HTML/CSS (using templates like **`chat.html`**, **`dashboard.html`**, **`feedback.html`**)

- **Backend:**

Python Flask (**`app.py`**) handles API routing, sentiment analysis, and database interactions.

- **Database:**

SQLite (**`citizen_ai.db`**) stores feedback and sentiment data. Structured tables are used for user feedback, responses, and analytics summaries.

4. Setup Instructions:

Prerequisites:

 Python 3.8+ – The programming language used to build and run the backend logic of the application.

 Flask – A lightweight Python web framework used to develop the server and handle routes and APIs**.**

 IBM Watson SDK – Provides access to Watson services like AI-based sentiment analysis and language processing.

 SQLite – A lightweight, serverless database used to store user feedback, sentiment results, and analytics data**.**

Installation:

- **Clone the Repository**

```
git clone <repo_url>
```

→ Downloads the Citizen AI project files from the remote repository to your local system.

- **Navigate to the Project Directory**

```
cd CITIZEN_AI( WITH API KEY )
```

→ Moves into the extracted project folder where the application files are located.

- **Install Required Dependencies**

```
pip install -r requirements.txt
```

→ Installs all Python packages listed in the requirements file needed to run the project.

- **Start the Flask Application**

```
python app.py
```

→ Launches the backend server locally, making the app accessible via browser (typically at **`http://127.0.0.1:5000`**).

5. Folder Structure Client:

Client (Frontend):

The frontend is built using **HTML/CSS** templates located in the `templates/` folder and styled via files in the `static/` directory. Pages like **`chat.html`, `feedback.html`,** and **`dashboard.html`** provide a user-friendly interface for interacting with the chatbot, submitting feedback, and viewing sentiment analytics.

Server (Backend):

The backend is developed using **Python Flask**, defined in app.py. It handles routing, processes user input, performs sentiment analysis using **IBM Watson**, interacts with the SQLite database, and renders the frontend templates dynamically through Jinja2 templating.

- **.env** – Contains sensitive information like API keys used securely by the application.
- **app.py** – Main Flask script that handles routing, logic, and backend processes.
- **templates/** – Folder storing HTML files that render the user interface in the browser.
- **static/** – Contains static assets such as CSS stylesheets for UI design.
- **citizen_ai.db** – SQLite database file that stores user feedback, sentiment data, and history.

6. Running the Application:

- **Frontend:** Rendered via Flask routes (e.g**.,** **`/chat`, `/feedback`, `/dashboard`**)
- **Backend:**

python app.py

To run the backend, simply execute python app.py, which starts the Flask server locally.

The frontend is automatically served through Flask routes like /chat, /feedback, and /dashboard in the browser**.**

7. API Documentation:

 POST /analyze – Sends user input to the server to analyze sentiment using IBM Watson.

- Params: message
- Response: { "sentiment": "Positive" }

 GET /history – Retrieves all past user feedback entries stored in the database.

- Response: List of entries

 POST /feedback – Submits new feedback data along with a timestamp for storage and analysis.

- Params: message, timestamp

User Query:

In the API documentation, a User Query refers to the text input submitted by a citizen through the chatbot or feedback form.

This query is processed by the backend and passed to the sentiment analysis model.

The response helps determine the sentiment (Positive/Neutral/Negative) and store relevant data for analytics.

8.Authentication:

- Basic session management (Flask sessions)
- No external OAuth or JWT is used in the current implementation

9.User Interface:

- **chat.html** – AI Chat interface
- **feedback.html** – Feedback submission form
- **dashboard.html** – Sentiment & feedback overview
- **admin.html** – Admin panel (for monitoring)

10. Testing:

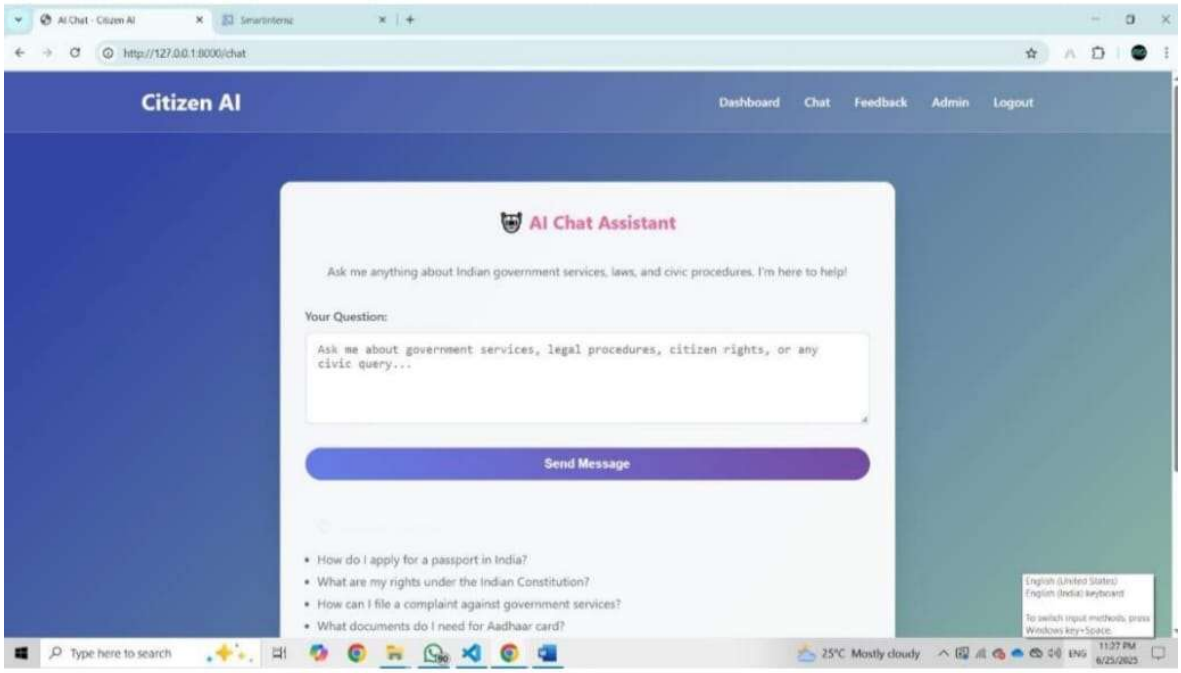
- Manual testing performed via Flask routes and UI validation
- Test Cases:

- **Input Validation:**
  Ensures that user inputs (text fields, forms) accept valid data and display appropriate error messages for invalid entries.
  This helps maintain data quality and enhances the user experience by preventing incorrect submissions.

- **Sentiment Accuracy:**
  Verifies that the sentiment analysis API correctly interprets and classifies the user's emotional tone.
  This ensures reliable insights for the analytics dashboard and accurate feedback representation.

- **Database Write/Read Checks:**
  Confirms that user feedback is successfully stored in the SQLite database and retrieved correctly for display.
  This guarantees the integrity and accessibility of data for both users and administrators.

11. Screenshots:



12. Issues:

No Authentication Layer for Admin Features

The application currently lacks user authentication or access control for admin-level pages like the dashboard.

This poses a security risk, as anyone with the URL can view sensitive data without restriction.

Error Handling is Basic (e.g., Watson API Timeouts)

The system has minimal error management for scenarios like API delays or failures from IBM Watson services.

This may lead to a poor user experience or app crashes if external services are temporarily unavailable.

13. Future Enhancements:

 Add OAuth (Google/Gov ID) Login – Implement secure user authentication to verify citizen identity via trusted providers.

 Deploy on Cloud (e.g., AWS/GCP) – Host the platform on a scalable cloud service to ensure availability and reliability.

 Improve Sentiment Accuracy Using IBM Granite Fine-Tuning – Enhance AI precision by customizing Granite models with domain-specific training data.

 Enable Multi-Language Support – Allow citizens to interact in regional languages for broader accessibility and inclusivity.