Citizen AI – Intelligent Citizen Engagement Platform

Team ID : LTVIP2025TMID59973

Team Size : 4

Team Leader : Jagili Mounika

Team member : Kamisetty Harsha Vardhan

Team member : Machiraju Chaitanya Kumar

Team member : Kotha Manohar

Project Description:

Citizen AI is an intelligent citizen engagement platform designed to revolutionize how governments interact with the public. Leveraging Flask, IBM Granite models, and IBM Watson, Citizen AI provides real-time, AI-driven responses to citizen inquiries regarding government services, policies, and civic issues. The platform integrates natural language processing (NLP) and sentiment analysis to assess public sentiment, track emerging issues, and generate actionable insights for government agencies.

A dynamic analytics dashboard offers real-time visualizations of citizen feedback, helping policymakers enhance service delivery and transparency. By automating routine interactions and enabling data-driven governance, Citizen AI improves citizen satisfaction, government efficiency, and public trust in digital governance.

Scenarios:

Scenario 1: Real-Time Conversational AI Assistant:

The Real-Time Conversational AI Assistant in Citizen AI serves as the primary interface for citizen interaction. It allows users to engage with public services naturally by typing questions or requests. The system captures user input in real- time and immediately sends it to a powerful underlying AI model, such as IBM Granite. This model processes the query and generates a relevant, human-like response on the fly. The assistant then displays this response back to the user almost instantly, facilitating quick access to information, support, and the ability to perform tasks like reporting issues, 24/7. It aims to provide a seamless and efficient conversational experience for civic engagement.

Scenario 2: Citizen Sentiment Analysis:

Citizen Sentiment Analysis in Citizen AI is a core feature designed to understand the public's feelings about government services and related topics.

It works by analysing text input, whether from direct citizen feedback submitted through the platform or potentially from other digital interactions (though the current implementation focuses on submitted text).

Using AI (like the simple analyse_sentiment function in app.py), the system classifies the sentiment of the text as Positive, Neutral, or Negative.

This process helps the government quickly identify areas of public satisfaction or concern. By aggregating sentiment data, the platform provides valuable insights into overall citizen mood and highlights specific issues that may need attention, ultimately aiming to improve service delivery and citizen satisfaction. The results are presented on the dashboard for easy monitoring.

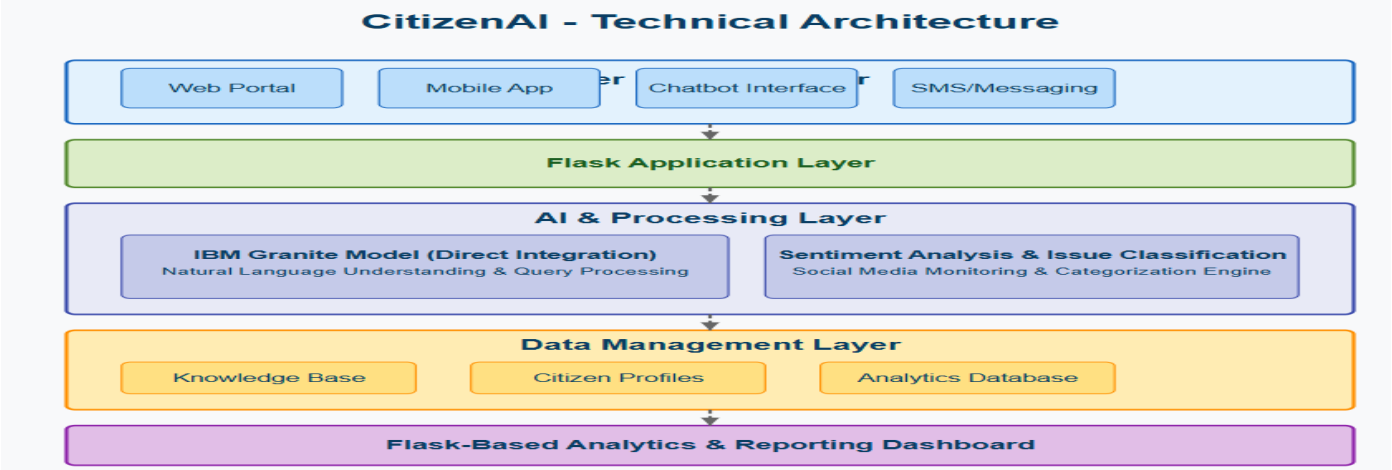
Scenario 3: Dynamic Dashboard:

The Dynamic Dashboard in Citizen AI serves as a central hub for government officials to gain real-time insights into citizen feedback and interactions. It visualizes key data points, including the overall citizen sentiment (positive, neutral, negative) derived from submitted feedback. The dashboard also tracks interaction trends over time, showing peak periods of activity. Furthermore, it can display aggregated government service ratings or issues reported by citizens. By presenting this information dynamically through charts and clear metrics, the dashboard empowers government departments to quickly understand public perception, identify areas needing improvement, and make data-driven decisions to enhance public services and citizen satisfaction. It transforms raw interaction data into actionable intelligence for a more responsive government.

Scenario 4: Personalized & Contextual Response System: Citizen AI features a Personalized & Contextual Response System, powered by IBM Granite models. This system acts as your intelligent chat assistant. Utilizing Granite's advanced natural language understanding (NLU), the platform can accurately interpret citizen queries, understanding the nuance and context of their questions. This capability allows Citizen AI to provide relevant and tailored

responses related to public services and information. The aim is to move beyond generic answers, offering smarter, faster, and more accessible interactions by understanding the specific needs behind each citizen's query and providing information accordingly.

Technical Architecture:



Pre-requisites

Pre-requisites:

1. **Python:** You need a working Python 3.7+ environment installed on your system.
2. **Flask:** The Flask web framework is required to run the web application.
3. **PyTorch:** As you are using a deep learning model, you need PyTorch installed. If you plan to use your GPU for faster inference, ensure you install the version of PyTorch with CUDA support that matches your GPU and CUDA toolkit version.
4. **Hugging Face Libraries:** The transformers, accelerate, and bitsandbytes libraries are essential for loading and utilizing the IBM Granite model, especially with quantization.
5. **Sufficient Hardware:** Running a large language model like IBM Granite 3.3B requires significant resources. You will need:

- **RAM:** A substantial amount of RAM (typically 16GB or more is recommended, even with quantization).
- **GPU (Recommended):** A compatible NVIDIA GPU with sufficient VRAM (8GB or more is highly recommended, especially for the 8B model, even with 4-bit quantization) and correctly installed CUDA drivers for reasonable inference speed. Running solely on a CPU will be very slow.

6. **Internet Connection:** The first time you run the application, the IBM Granite model files will be downloaded from the Hugging Face Hub. You need an active internet connection for this.

7. **Project Structure:** The project files should be organized correctly with app.py, a templates folder containing your HTML files (index.html, about.html, services.html, chat.html, dashboard.html, login.html), and a static folder containing your CSS (styles.css) and image/favicon subfolders (e.g., static/Images, static/Favicon).

Project Setup and Architecture:

- **Activity 1.1:** Select and confirm the generative AI model (IBM Granite) and necessary libraries (Transformers, Accelerate, BitsAndBytes, PyTorch).
- **Activity 1.2:** Define the system architecture: Flask backend, HTML/CSS frontend, AI model integration, and data handling (in-memory history, planning for database persistence).
- **Activity 1.3:** Set up the development environment, installing Python, Flask, and all required AI/ML libraries and dependencies.

Backend Core Functionalities

- **Activity 2.1:** Implement core Flask routes (/, /about, /services, /chat, /dashboard, /login, /logout).
- **Activity 2.2:** Develop user authentication logic for login/logout and session management.
- **Activity 2.3:** Integrate the IBM Granite model loading and text generation functionality.
- **Activity 2.4:** Implement helper functions for AI response generation, sentiment analysis, and data formatting.

Data Handling and Logic

- **Activity 3.1:** Set up in-memory storage for chat history, sentiment, and concerns (plan for database integration for persistence).

- **Activity 3.2:** Implement logic for processing user input (questions, feedback, concerns) and updating the data storage.
- **Activity 3.3:** Develop logic for fetching and aggregating data for the dashboard view (e.g., sentiment counts, recent issues).

Frontend Development

- **Activity 4.1:** Design and develop HTML templates for all project pages (index.html, about.html, services.html, chat.html, dashboard.html, login.html).
- **Activity 4.2:** Implement styling using styles.css and inline CSS for page layout and appearance.
- **Activity 4.3:** Create forms for user input (chat, feedback, concern, login) and ensure correct data submission.
- **Activity 4.4:** Display dynamic content from the backend in the HTML templates (AI responses, dashboard data, error messages).

Integration and Testing

- **Activity 5.1**: Integrate the frontend templates with the Flask backend routes.
- **Activity 5.2:** Test all user flows, including login, logout, page navigation, chat interaction, feedback/concern submission, and dashboard viewing.
- **Activity 5.3:** Debug any errors encountered in the backend or frontend.

Refinement and Deployment

- **Activity 6.1**: Refine UI/UX based on testing and feedback. Optimize code for performance, especially AI inference.
- **Activity 6.2**: Prepare for deployment (configure server environment, set up a persistent database if planned).
- **Activity 6.3:** Deploy the application to a hosting platform.
- **Activity 6.4:** Provide documentation and user guides.

Select and Confirm AI Model & Libraries Understand Project Requirements

1. Review the key functionalities of CitizenAI, including chat responses, sentiment analysis, concern reporting, and dashboard insights.
2. Identify the type of AI capabilities needed: natural language understanding (NLU), text generation, and basic text analysis.

Confirm AI Model & Libraries:

1. Confirm the selection of the IBM Granite model for core AI capabilities.

Define the Architecture of the Application Draft an Architectural Overview:

1. Design a structured architecture including the Flask backend, HTML/CSS frontend, integration points for the IBM Granite AI model, and the approach for data handling (initially in-memory history, planning for future persistent storage).
2. Define how user requests and data will flow through the system components.

Define Data Flow:

1. Map the flow of user input (chat messages, feedback, concerns) to the backend, AI processing, data storage, and back to the frontend for display.
2. Specify the necessary Python libraries: Flask for the web framework, PyTorch for the AI model backend, and Hugging Face libraries (transformers, accelerate, bitsandbytes) for model handling.

Explore Library Documentation:

1. Review documentation for selected libraries to understand model loading, inference, quantization, and device handling.
2. Examine Flask documentation for routing, templating, and session management.

Set Up the Development Environment Install Necessary Tools

1. Ensure Python (3.7+) and pip are installed for managing project dependencies.
2. Install Flask and Dependencies:

- Use pip to install Flask and any other required backend libraries:

  >> pip install Flask

Install AI/ML Libraries:

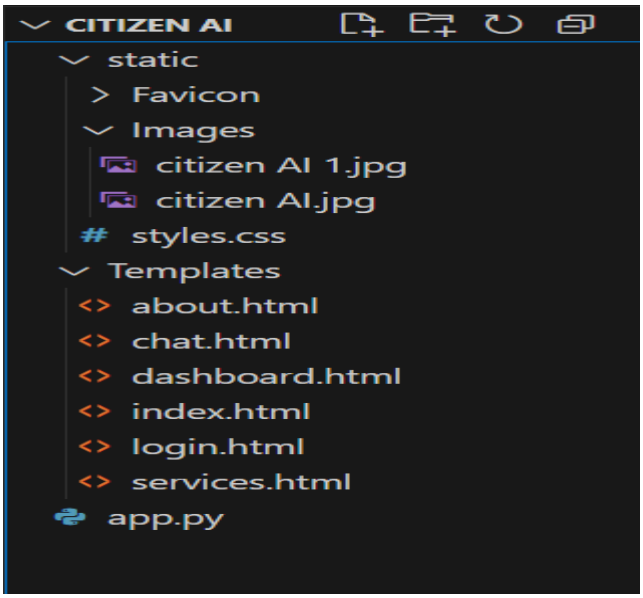
- Install the necessary libraries for AI model integration:

  >> pip install torch transformers accelerate bitsandbytes

(Ensure you install the correct PyTorch version for your CUDA setup if using a GPU).

Set Up Application Structure:

- Create the basic project directory structure: app.py, templates/ (for HTML files), and static/ (with css/, Images/, Favicon/ subfolders).



Writing the Main Application Logic in app.py

This activity involves implementing the Python functions within app.py that define how the application responds to specific user actions and integrates the core AI and data processing functionalities.

1. **Define the Core Routes in app.py**

Set up separate Flask routes using the @app.route() decorator in app.py for the specific actions that involve processing user input and triggering application logic:

1. /ask ? Accepts a user's question from the chat interface and generates a response using the IBM Granite model.
2. /feedback ? Accepts user feedback text and performs sentiment analysis on it.
3. /concern ? Accepts a user's report of a concern or issue for logging.
4. /login (with methods=['POST']) ? Handles the submission of username and password for user authentication.

Each of these routes will serve as the entry point in your backend to process specific user requests and initiate the corresponding application logic.

2. Set Up Route Handlers for Each Feature

For each of the routes defined above, implement the corresponding Python function in app.py. These functions act as the handlers for incoming requests to these routes:

1. Capture user inputs from the HTML forms associated with these routes using request.form.get('input_name'), ensuring safe retrieval of data like the question text, feedback content, concern details, username, and password.
2. (Optional but recommended) Include basic validation steps to check if submitted data is present and in the expected format before proceeding with processing.
3. Prepare the data to be passed to the next steps in the workflow (e.g., to AI functions or data storage).

4. Determine the appropriate response to send back to the user, which typically involves rendering an HTML template (render_template()) or redirecting to another page (redirect()).

   For example:

1. The /ask handler will retrieve the question text submitted via the chat form.
2. The /feedback handler will retrieve the feedback text entered by the user.
3. The /login (POST) handler will retrieve the entered username and password.

3. Integrate AI Model Calls and Logic in Each Function

Within the relevant route handler functions defined in step 2, integrate the calls to your AI model and other application logic:

1. In the /ask route handler function (e.g., ask_question()), implement the call to your IBM Granite inference function (e.g., granite_generate_response()), passing the user's question as input.
2. In the /feedback route handler function (e.g., submit_feedback()), implement the call to your sentiment analysis function (e.g., analyze_sentiment()), passing the user's feedback text.
3. Process the results returned by these functions (e.g., store the sentiment result, get the generated text).
4. Format the AI-generated output or processing results as needed for display on the frontend.
5. Pass the final results to the render_template() function to display them clearly on the appropriate HTML page (e.g., passing the generated response to chat.html, passing the sentiment result to chat.html).
   For example:

1. The ask_question() function will call granite_generate_response() with the user's question and then render chat.html, including the original question and the AI's response.

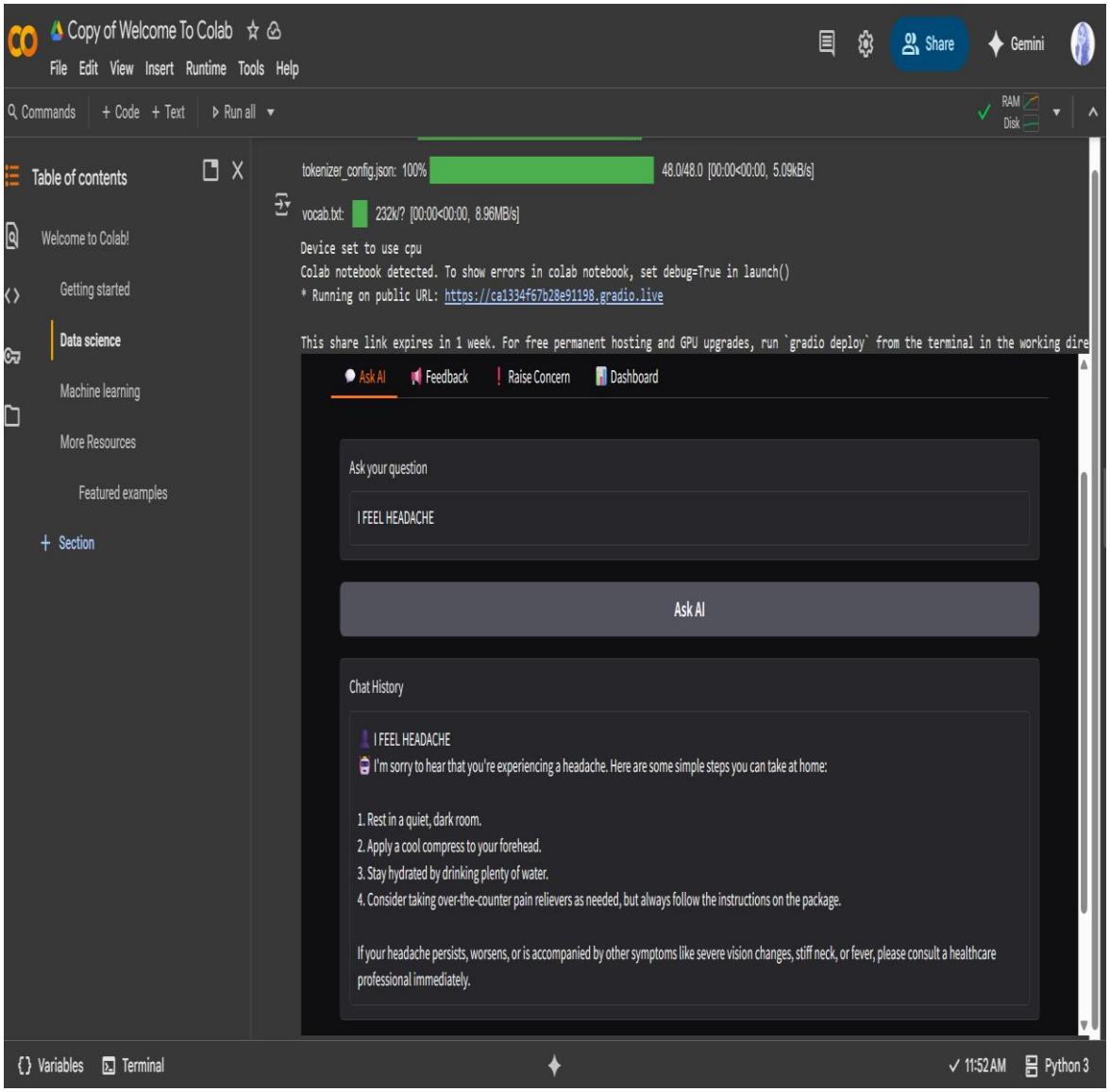
2. The submit_feedback() function will call analyze_sentiment() with the feedback and then render chat.html, indicating the sentiment result.

```
•
•  !pip install gradio transformers torch --quiet
•  import gradio as gr
•  import torch
•  from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
•
•  # 🔄 Load Granite model
•  model_name = "ibm-granite/granite-3.3-2b-instruct"
•  tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
•  model = AutoModelForCausalLM.from_pretrained(
•      model_name,
•      torch_dtype=torch.float16,
•      device_map="auto",
•      trust_remote_code=True
•  )
•
•  # ☑️ Optimized pipeline (faster)
•  chat_pipeline = pipeline(
•      "text-generation",
•      model=model,
•      tokenizer=tokenizer,
•      max_length=512,  # 🔽 Reduced for speed
•      temperature=0.7,
•      do_sample=True,
•      pad_token_id=tokenizer.eos_token_id
•  )
•
•  # Sentiment model (fast + small)
•  sentiment_model = pipeline("sentiment-analysis")
•
•  # Global memory
•  chat_history = []
•  feedback_list = []
•  concerns = []
•
•  # 💬 Smart Chat
•  def handle_chat(user_input):
•      prompt = f"You are a helpful AI assistant for Indian public services.
   Answer clearly:\n\n{user_input}\n\nAnswer:"
•      response = chat_pipeline(prompt)
•      answer = response[0]['generated_text'].split("Answer:")[-1].strip()
•      chat_history.append((user_input, answer))
•      trimmed = chat_history[-5:]  # Limit to last 5 Q&A
•      return "\n\n".join([f"👤 {q}\n🤖 {a}" for q, a in trimmed])
•
```

```
# 📢 Feedback
def handle_feedback(text):
    sentiment = sentiment_model(text)[0]["label"]
    feedback_list.append((text, sentiment))
    return f"☑ Feedback received. Sentiment: {sentiment}"

# ❗ Concern
def handle_concern(text):
    concerns.append(text)
    return "📬 Concern submitted successfully."

# 📊 Dashboard
def dashboard_summary():
    pos = sum(1 for _, s in feedback_list if s == "POSITIVE")
    neg = sum(1 for _, s in feedback_list if s == "NEGATIVE")
    dash = f"🟢 Positive Feedbacks: {pos}\n🔴 Negative Feedbacks: {neg}\n\n📌 Concerns:"
    dash += "\n" + "\n".join([f"• {c}" for c in concerns]) if concerns else "\nNo concerns submitted yet."
    return dash

# 🎛 Gradio UI
with gr.Blocks(title="Citizen AI - Fast Granite Edition") as app:
    gr.HTML("<h2 style='text-align:center;'>IN Citizen AI</h2><p style='text-align:center;'>Granite-powered Assistant (Fast Mode)</p>")

    with gr.Tabs():
        with gr.Tab("💬 Ask AI"):
            user_input = gr.Textbox(label="Ask your question")
            ask_btn = gr.Button("Ask AI")
            chat_output = gr.Textbox(label="Chat History", lines=10)
            ask_btn.click(fn=handle_chat, inputs=user_input, outputs=chat_output)

        with gr.Tab("📢 Feedback"):
            feedback_input = gr.Textbox(label="Write your feedback")
            feedback_btn = gr.Button("Submit Feedback")
            feedback_output = gr.Textbox(label="Status")
            feedback_btn.click(fn=handle_feedback, inputs=feedback_input, outputs=feedback_output)

        with gr.Tab("❗ Raise Concern"):
            concern_input = gr.Textbox(label="Describe your concern")
            concern_btn = gr.Button("Submit Concern")
            concern_output = gr.Textbox(label="Status")
            concern_btn.click(fn=handle_concern, inputs=concern_input, outputs=concern_output)

```

```
        with gr.Tab("📊 Dashboard"):
            dash_btn = gr.Button("Refresh Dashboard")
            dash_output = gr.Textbox(label="Summary", lines=12)
            dash_btn.click(fn=dashboard_summary, outputs=dash_output)

    gr.HTML("<p style='text-align:center;'>⚡ Optimized for faster response | Powered by IBM Granite + Hugging Face</p>")

# 🚀 Launch the app
app.launch(share=True)
```

output:



Designing and Developing the User Interface

This activity involves creating the structure and visual appearance of all the web pages in your CitizenAI application using HTML and CSS.

1. Set Up the Base HTML Structure

Develop the necessary HTML files that represent each page of your application.

1. Create the main HTML template files: index.html, about.html, services.html, chat.html, dashboard.html, and login.html.
2. Include the standard HTML5 boilerplate (<!DOCTYPE html>, <html>, <head>, <body>) in each file.

3. Incorporate a consistent header and navigation menu in pages where appropriate (e.g., on protected pages after login) to allow users to easily access different sections like:

- About
- Services
- Chat
- Dashboard
- Login/Logout (conditional display based on session)

Use semantic HTML elements such as <header>, <nav>, <main>, <section>, <footer>, <h1>, <p>, <ul>, <li>, <form>, <input>, <button>, <textarea> to create a clean, organized, and accessible page structure.

2. Design the Layout and Styling Using CSS

Create and apply CSS rules to control the visual presentation and layout of your web pages.

1. Create or update the main CSS stylesheet (static/css/styles.css) to define the overall look and feel, including font styles, colors, and spacing for common elements.
2. Implement layout techniques within your CSS (e.g., using Flexbox for centering elements like the login box, or adjusting margins and padding for content areas) to arrange elements on the page effectively.
3. (Optional but recommended) Consider implementing media queries in your CSS to ensure the layout and styling are responsive and look good on different screen sizes (desktops, tablets, mobile phones).
4. Apply specific visual styles (backgrounds, borders, text colors) to individual elements or sections, potentially using inline <style> blocks in specific HTML files for page-unique styles (like the background image on index.html or about.html).

3. Create Separate Pages for Each Core Functionality

Develop the specific content and interactive elements for each distinct page of the application.

1. index.html: Design the landing page with an introduction to CitizenAI and a clear call to action (e.g., the "Get Started" button linking to login).
2. login.html: Create the page with the login form, including input fields for username/email and password, and a submit button. Include a placeholder for displaying login error messages.

3. about.html: Develop the page containing information about the project's mission, features, and impact.
4. services.html: Create a page detailing the services offered by CitizenAI.
5. chat.html: Design the interface for the AI chat assistant, including a form for user input (question) and a dedicated area to display the AI's generated response. This page might also include forms for submitting feedback and concerns.
6. dashboard.html: Build the page to display aggregated data, such as sentiment counts and a list of reported issues.

Each of these pages will contain the necessary user input forms and designated areas where dynamic content from the backend (like AI responses or dashboard data) will be displayed.

Creating Dynamic Templates with Flask's render_template

This activity focuses on using Flask's built-in templating engine to populate the HTML structures created in Activity 4.1 with dynamic data generated by the backend.

1. Integrate Flask’s render_template for Dynamic Content Rendering:

Within each Flask route function in app.py that serves an HTML page:

1. Utilize the render_template('filename.html', ...) function. This function takes the name of the HTML file located in the templates folder as its primary argument.
2. Pass Python variables containing dynamic data as keyword arguments to the render_template() function (e.g., render_template('chat.html', question_response=ai_response, sentiment=feedback_sentiment)). This makes these variables accessible within the specified HTML template.
3. In the HTML templates, use Jinja2 template syntax ({{ variable_name }} to display variable values, {% if condition %} for conditional rendering, {% for item in list %} for loops, etc.) to access and display the data passed from the Flask backend. This allows the frontend to dynamically render AI-generated responses, sentiment results, dashboard statistics, error messages, and other variable content based on the backend's processing.

Binding backend data to HTML templates is crucial for dynamic web pages in Flask. Your app.py processes user input and generates results, such as AI responses or sentiment analysis. Flask's render_template() function then sends these results as variables to your HTML files. Within the HTML, Jinja2 templating syntax, like {{ variable_name }}, is used to display the value of these variables. This makes the content shown to the user update dynamically based on the backend's processing.

Set Up a Virtual Environment

To ensure dependency isolation, create and activate a virtual environment before installing required packages.

“python -m venv env

source env/bin/activate (Linux/Mac) env\Scripts\activate

(Windows) pip install -r requirements.txt”

This ensures that Flask, Gemini API libraries, and other dependencies are installed and available.

Configure Environment Variables

This Python snippet sets up an IBM Granite AI model for use in a project:

1. model_path is defined as "ibm-granite/granite-3.3-8b-instruct", specifying the path to the pretrained IBM Granite model.

2. A comment notes that large models require considerable hardware resources (RAM, GPU).

3. The code determines whether a GPU (cuda) is available using torch.cuda.is_available().

4. If a GPU is available, it sets device = "cuda"; otherwise, it defaults to "cpu".

5. It prints the chosen device for transparency: Using device: cuda or cpu.

6. AutoTokenizer.from_pretrained(model_path) loads the tokenizer from the specified model path.

7. The tokenizer is essential for converting user input into token IDs the model can understand.

8. This setup is typically part of a larger pipeline for generating AI responses or predictions.

9. It leverages Hugging Face's transformers library functionality.

10. The model is likely used for text generation, classification, or interaction (e.g., chat assistant).

Testing and Verifying Local Deployment

1. **Start the Flask Application**

Run the following command to launch the application locally:

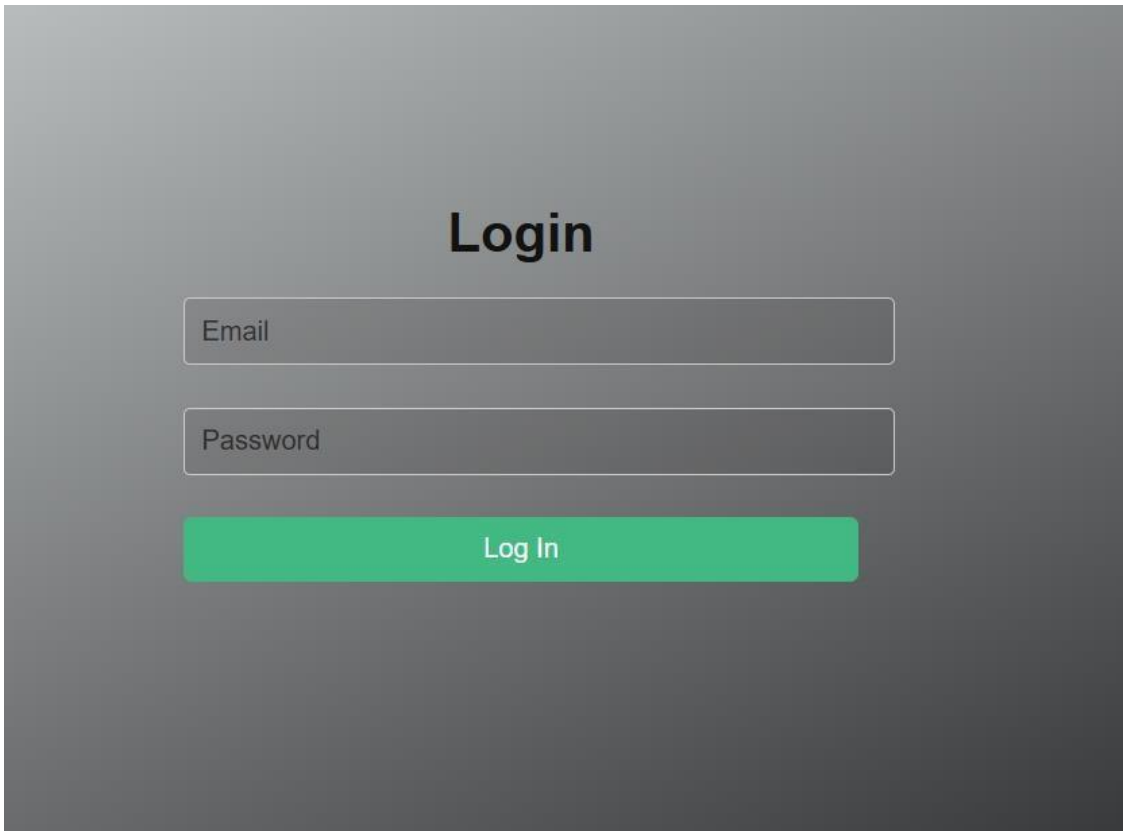
python app.py **Exploring**

website’s Web Pages: Index page:

This is the landing page of the CitizenAI web application, designed for civic engagement through AI. Here's a breakdown:

1. Header Section:
    - Shows the main title: "Welcome to CitizenAI".
    - Contains navigation links: About, Services, Chat, Dashboard, and Login.
2. Left Panel (Intro Section):
    - Headline: "Empowering Citizens Through AI".
    - Describes CitizenAI as an intelligent assistant helping citizens engage with government services, provide feedback, and communicate more effectively.
    - Includes a prominent "Get Started" button, likely redirecting to user interaction or signup.
3. Right Panel (Visual):
    - Features a digital, futuristic background with a human head silhouette and glowing circuit lines.
    - Overlaid with code and AI-related text to emphasize the tech-driven nature of the platform.
4. Purpose:
    - The page promotes an AI-powered civic platform aimed at building a smarter, responsive government-citizen relationship.
    - It encourages users to begin interacting with the system by clicking **Get Started**.

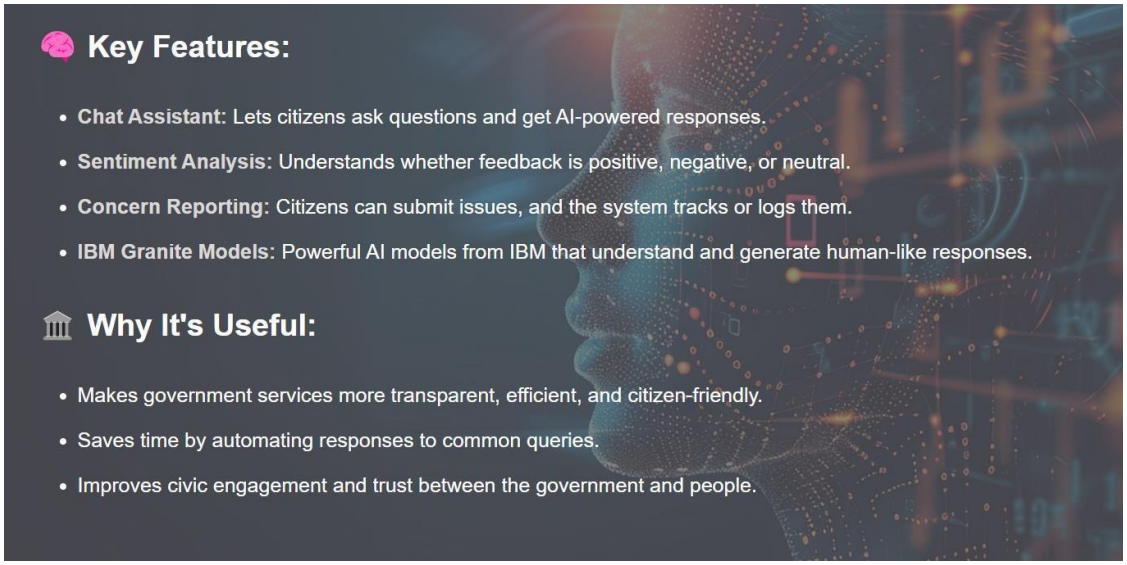
Login page:



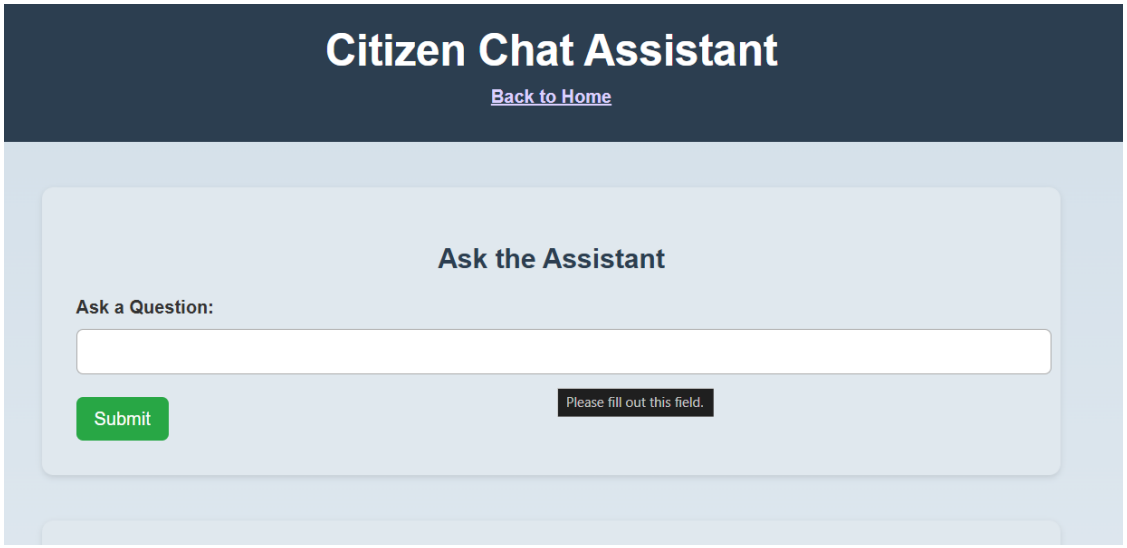
This page is the Login page for the application. Its primary function is to allow users to securely access the platform. You can see input fields for entering your Email (or username) and Password, along with a Log In button to submit your credentials. This page is where users authenticate themselves before gaining access to the protected features and content of the application.

This page is the "About Citizen AI" page. It serves to introduce users to the project, explaining what Citizen AI is and outlining its mission to improve interactions between governments and citizens using AI. The page provides a brief description and then breaks down the concept "In Simple Terms," likely highlighting key functions like asking questions and reporting issues. It also includes a "Back to Home" link for easy navigation.

This section of the page highlights the core aspects of the CitizenAI platform. It details the Key Features offered, such as the Chat Assistant for AI-powered responses, Sentiment Analysis for feedback, and Concern Reporting for issues, mentioning the use of IBM Granite Models. It also explains Why It's Useful, outlining benefits like making government services more transparent and efficient, and improving civic engagement

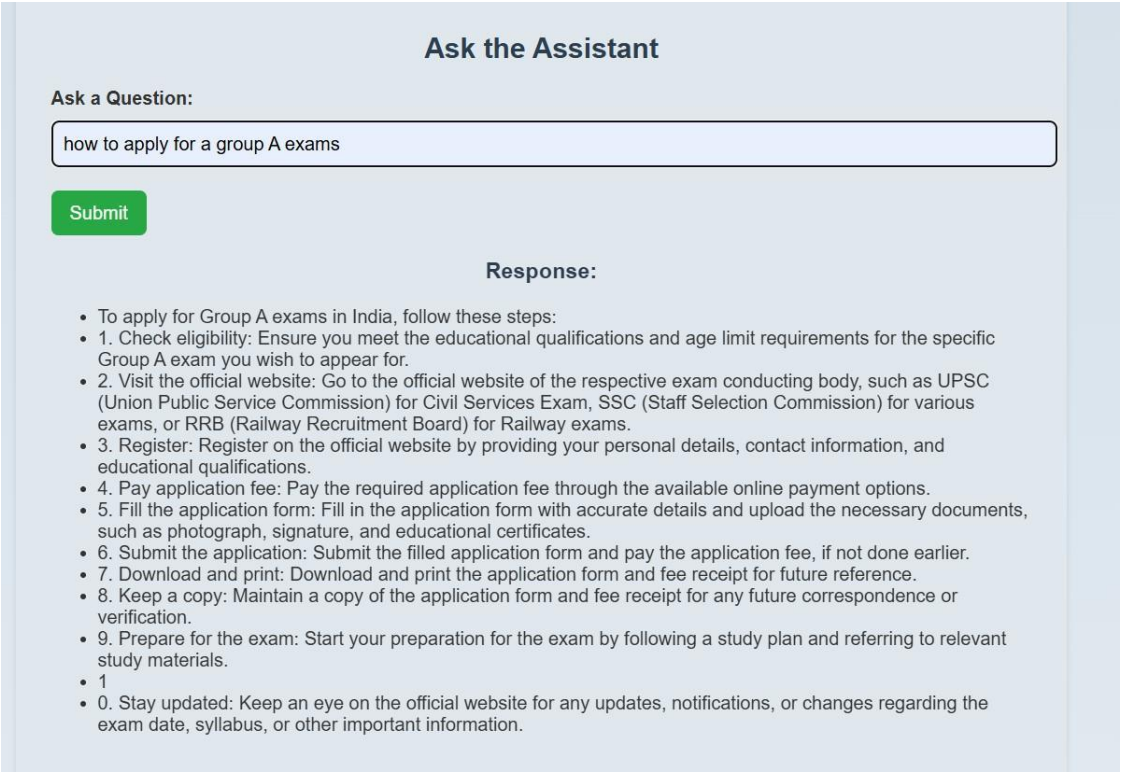


Chat Page:

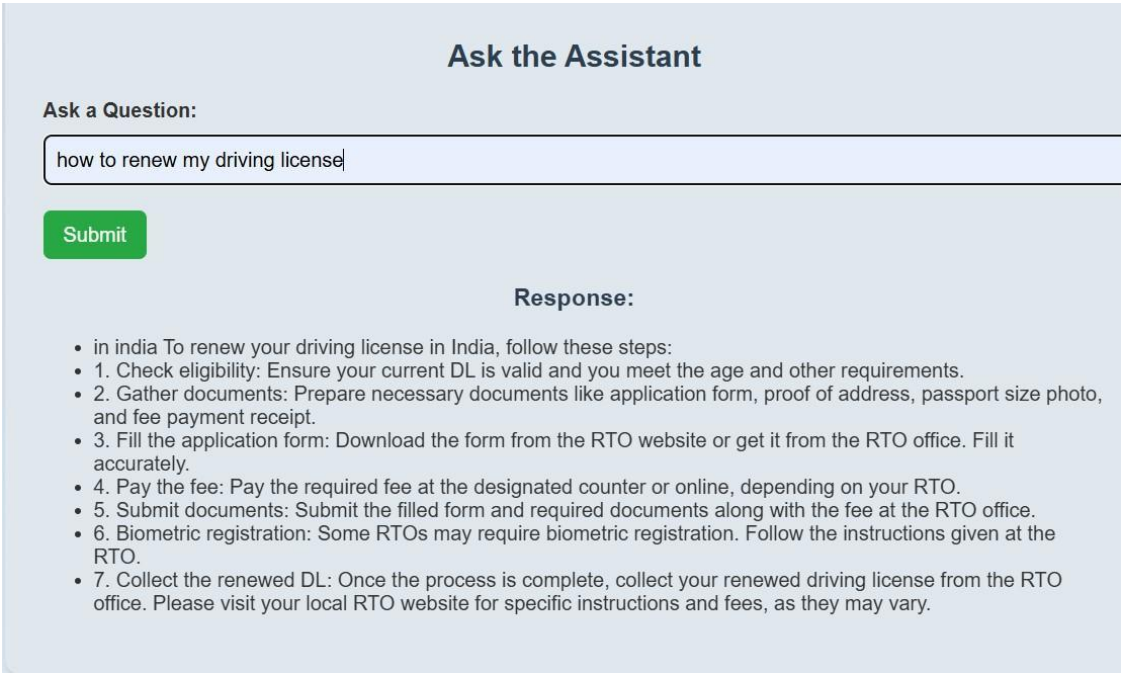


This page is the **"Citizen Chat Assistant"**. It's where you can directly interact with the AI assistant. You'll find a section titled "Ask the Assistant" with a text field to **type your question** and a **Submit button** to send it. The "Back to Home" link allows you to return to the main page. The message "Please fill out this field" indicates that you need to enter text in the question box before submitting.

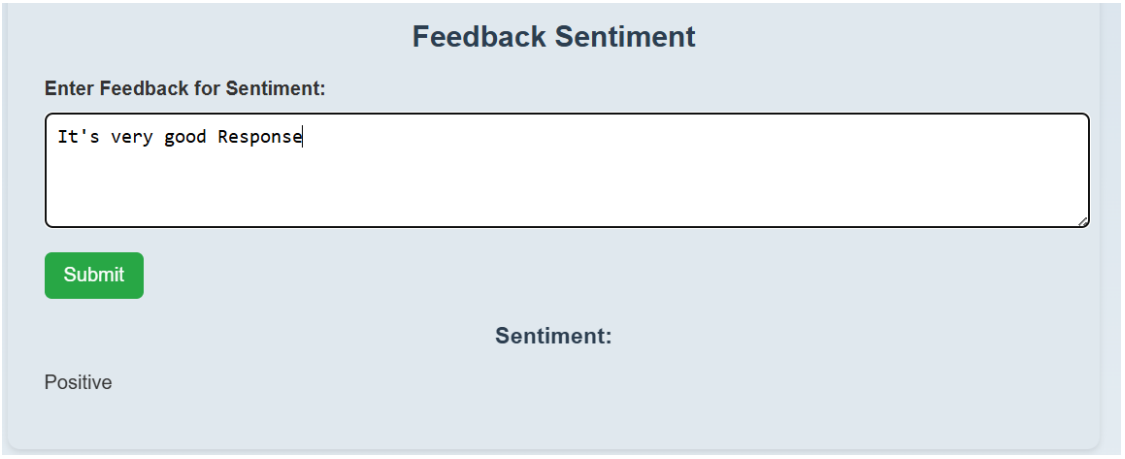
Response 1:



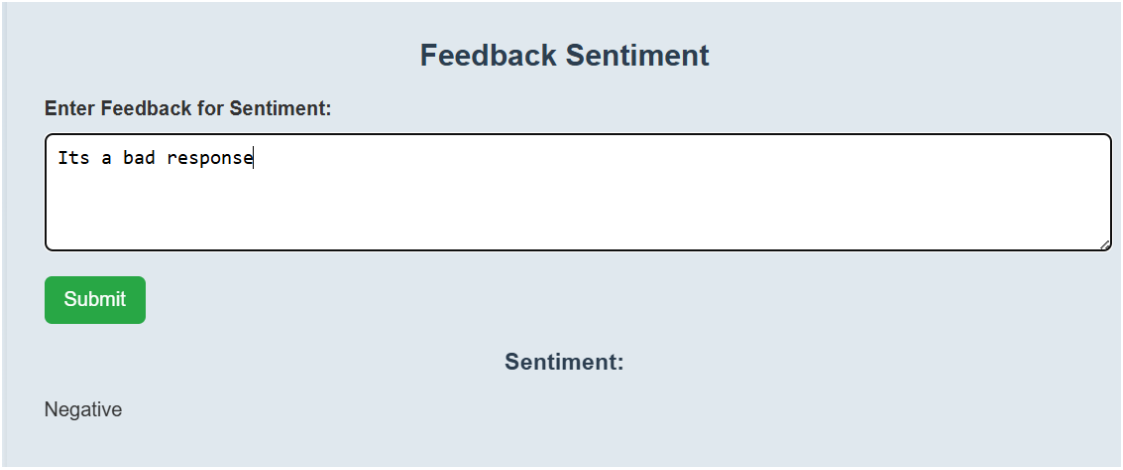
Response 2:



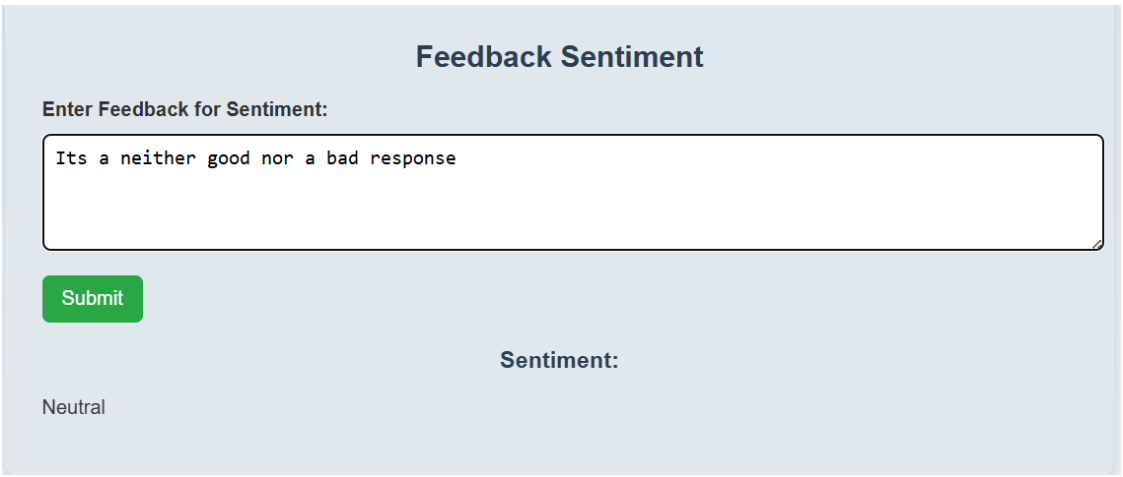
Sentiment Analysis: Sentiment 1



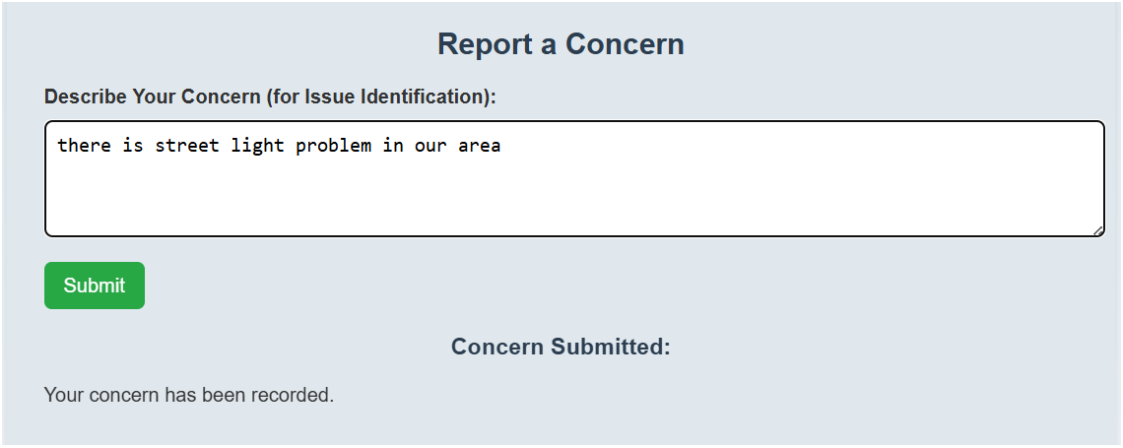
Sentiment 2:



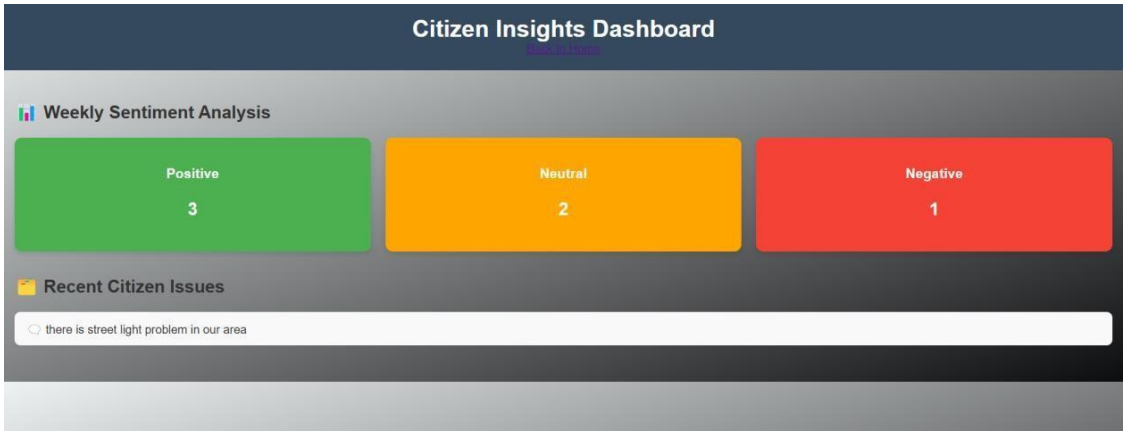
Sentiment 3:



Concern Reporting:



Dashboard page:



This page is the "Citizen Insights Dashboard". It provides an overview of the feedback and issues reported by citizens. You can see a summary of the Weekly Sentiment Analysis, showing counts for Positive, Neutral, and Negative feedback. Below that, there's a section for Recent Citizen Issues, listing the concerns that have been reported. This dashboard helps to quickly visualize citizen sentiment and identify common issues.

Conclusion:

Your AI-powered CitizenAI platform is designed to enhance interaction, accessibility, and transparency between citizens and government services. By integrating an AI chat assistant, sentiment analysis, concern reporting, and dashboard insights, the platform empowers users to easily access information, provide feedback, and report issues. With a Flask backend and an interactive HTML/CSS frontend, powered by the IBM Granite AI model, your project ensures a user-friendly experience while providing smart and responsive civic engagement tools. This innovative solution simplifies communication and fosters trust, making civic participation more convenient and efficient for all.