CITIZEN PLATFORM ENGAGEMENT
CITIZEN AI – INTELLIGENT

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.

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.

Architecture
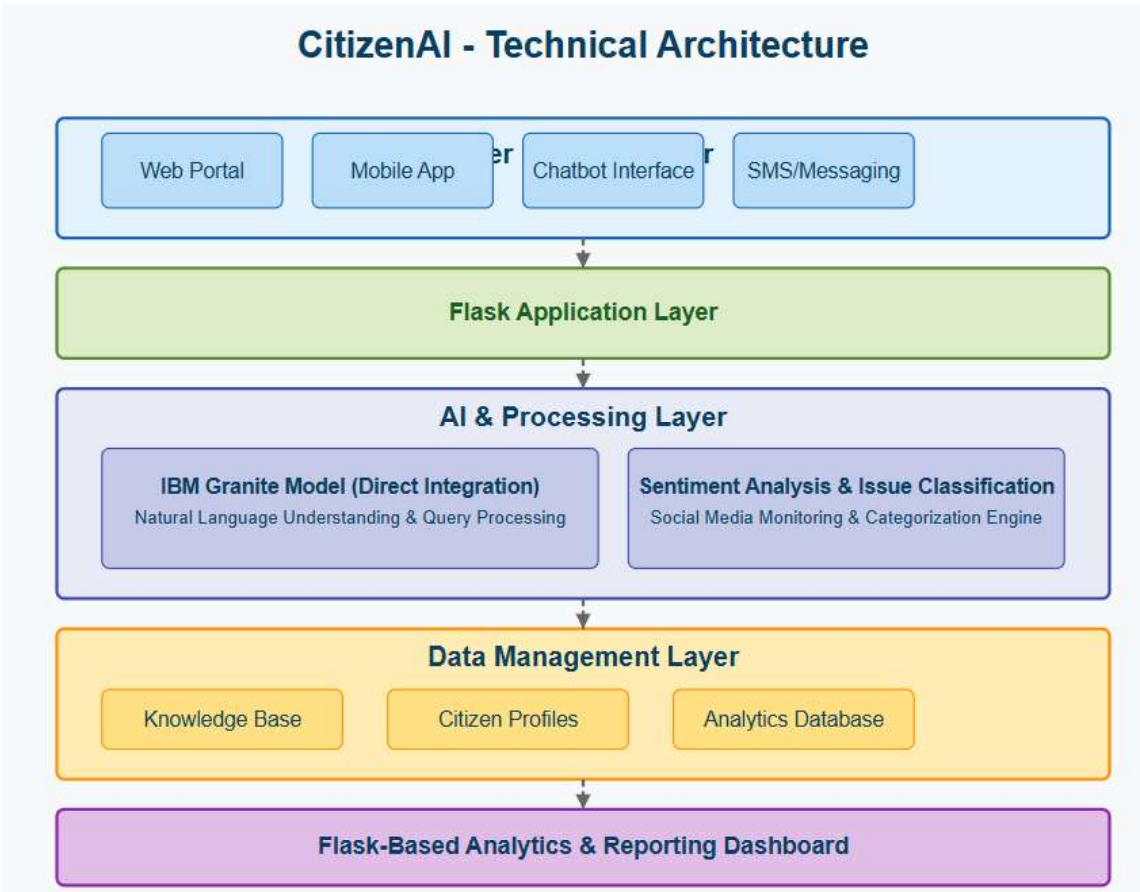
Pre-requisites

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 Workflow

Activity 1: 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.

Activity 2: 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.

Activity 3: 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).

Activity 4: 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).

Activity 5: 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.

Activity 6: 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.

Milestone 1: Project Setup and Architecture

In this milestone, we focus on confirming the core AI model and libraries, defining the overall system structure, and setting up the development environment.

Activity 1.1: Select and Confirm AI Model

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.

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.

Activity 1.2: 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.

Activity 1.3: Set Up the Development

- **Environment Install Necessary Tools:**

2. Ensure Python (3.7+) and pip are installed for managing project dependencies.

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).
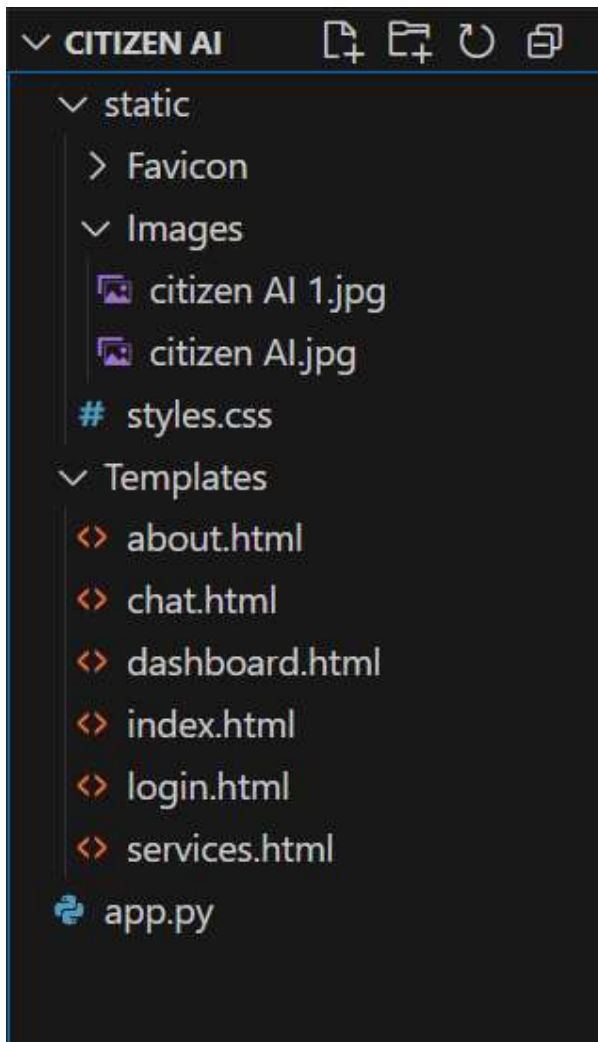
Milestone 2: Core Functionalities

Core Functionalities Development This milestone focuses on building the essential backend capabilities of the Citizen AI platform. It includes implementing Flask routes and establishing user authentication and session management. A core activity is integrating the IBM Granite AI model to handle citizen queries, alongside developing the logic for features like chat responses, sentiment analysis, and concern reporting.

Activity 2.1: Develop the Core Functionalities

Activity 2.1: Develop the Core Functionalities

```html
<!-- Chat Section -->
<section class="chat-section">
    <h2>Ask the Assistant</h2>
    <form action="/ask" method="POST">
        <label for="question">Ask a Question:</label>
        <input type="text" id="question" name="question" required />
        <button type="submit">Submit</button>
    </form>

    {% if question_response %}
    <div class="response">
        <h3>Response:</h3>
        <div class="response-text">
            {{ question_response | safe }}
        </div>
    </div>
    {% endif %}
</section>
```

This HTML code defines the Chat Section of the page. It includes a form titled "Ask the Assistant" where users can input a question using a text field and submit it with a button. The form sends the question to the /ask route on the backend. Below the form, there's a section designed to display the response. If a question_response variable is received from the server, its content is rendered here, showing the user the assistant's answer to their question.

Sentiment   **Analysis:**

```html
<!-- Sentiment Analysis Section -->
<section class="sentiment-section">
    <h2>Feedback Sentiment</h2>
    <form action="/feedback" method="POST">
        <label for="feedback">Enter Feedback for Sentiment:</label>
        <textarea id="feedback" name="feedback" rows="4"></textarea>
        <button type="submit">Submit</button>
    </form>

    {% if sentiment %}
    <div class="response">
        <h3>Sentiment:</h3>
        <p>{{ sentiment }}</p>
    </div>
    {% endif %}
</section>
```

This HTML code defines the Sentiment Analysis Section of your application. It provides a form titled "Feedback Sentiment" where users can enter their feedback into a text area and submit it. The submitted feedback is sent to the /feedback route for backend processing. The code checks if a sentiment result is available from the server. If a sentiment value (such as Positive, Negative, or Neutral) is present, it is displayed in a response area, allowing users to see the analyzed sentiment of their feedback.

```
<!-- Concern Submission Section -->
<section class="concern-section">
    <h2>Report a Concern</h2>
    <form action="/concern" method="POST">
        <label for="concern">Describe Your Concern (for Issue Identification):</label>
        <textarea id="concern" name="concern" rows="4"></textarea>
        <button type="submit">Submit</button>
    </form>

    {% if concern_submitted %}
    <div class="response">
        <h3>Concern Submitted:</h3>
        <p>Your concern has been recorded.</p>
    </div>
    {% endif %}
</section>
</main>
</body>
</html>
```

This HTML code snippet creates a Concern Submission Section in a webpage, allowing users to report issues:

1. The <section> block with class concern-section holds the entire form.

2. It includes a heading (<h2>Report a Concern</h2>) to label the section.

3. The <form> sends a POST request to the /concern route when submitted.

4. Inside the form, there's a <label> and a <textarea> where users can type their concern.

5. The <button type="submit"> allows users to submit their input.

6. The form field is named "concern", which is used by the server to retrieve the submitted text.

7. After submission, if concern_submitted is True, a confirmation message is shown.

8. This is done using Jinja templating: {% if concern_submitted %} and {% endif %}.

9.   If triggered, a <div class="response"> appears with a success message.

10.   This snippet is designed for integration with a Flask backend that handles form submissions and sets concern_submitted accordingly.

Activity 2.2: Implement the Flask Backend for Managing Routing and User Input Processing

This activity involves writing the Python code in app.py to define the web application's structure, handle different page requests, process data submitted by users, and integrate with the AI model logic.

- **Define Routes in Flask:**

1. Set up distinct routes in app.py using the @app.route() decorator for each page and key interaction point: / (Home), /about, /services, /chat, /dashboard, /login (GET and POST), /logout.
2. Define routes specifically for handling form submissions: /ask (for chat questions), /feedback (for sentiment analysis input), and /concern (for reporting issues), ensuring they accept POST requests.
3. Link each defined route to a corresponding Python function that will execute when that URL is accessed or form is submitted.
4. Ensure that within each route function, the appropriate HTML template is rendered using render_template(), passing any necessary data to the template.

- **Process User Input:**

1. Ensure that HTML forms (e.g., in chat.html, login.html) have the correct method="POST" and action="{{ url_for('route_name') }}" attributes to send data to the defined backend routes.
2. Within the Flask functions handling POST requests (e.g., ask_question(), submit_feedback(), login()), use Flask's request.form to safely retrieve data submitted by the user from the HTML forms (e.g., request.form.get('question'), request.form.get('username')).
3. Perform any necessary basic validation on the retrieved user input (e.g., checking if fields are empty).

- **Integrate AI Model Calls and Logic:**

1. Within the route functions that require AI processing (e.g., the /ask route), call the previously implemented AI helper functions (like granite_generate_response()).
2. Pass the processed user input (e.g., the user's question) as arguments to the AI helper functions.
3. In routes handling feedback or concerns (e.g., /feedback, /concern), call the relevant processing functions (like analyze_sentiment()) with the user-provided text.
4. Capture and process the results returned by the AI helper functions or data processing logic (e.g., the generated text response, the sentiment label).
5. Prepare the results to be sent back to the frontend by passing them as arguments to the render_template() function (e.g., render_template("chat.html", question_response=response)).

Milestone 3: Application Logic and Data Handling

This milestone is dedicated to implementing the specific functionalities of the CitizenAI platform. It involves developing the core logic for processing chat interactions and performing sentiment analysis on feedback. Setting up the application's data storage (initially in-memory history) and creating the logic to prepare data for the dashboard view are also key components of this milestone.

Activity 3.1: 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.

3. **HTML pages Routes:**

```python
# ----------------- Routes -----------------

@app.route('/')
def index():
    # Index page - does NOT require login
    return render_template("index.html")


@app.route('/about')
@login_required # Requires login to access
def about():
    return render_template("about.html")


@app.route('/services')
@login_required # Requires login to access
def services():
    return render_template("services.html")
```

Chat.html page Route:

```python
@app.route('/chat')
@login_required # Requires login to access
def chat():
    return render_template("chat.html")


@app.route('/ask', methods=['POST'])
@login_required # Requires login to access the chat functionality
def ask_question():
    question = request.form['question']
    response = granite_generate_response(question)
    # When rendering chat.html after a POST, pass the necessary data
    return render_template("chat.html", question_response=response)
```

Sentiment Analysis Route:

```python
@app.route('/feedback', methods=['POST'])
@login_required # Requires login to submit feedback
def submit_feedback():
    feedback = request.form['feedback']
    sentiment = analyze_sentiment(feedback)
    history.append({
        'date': datetime.now(),
        'sentiment': sentiment,
        'issue': '' # Issue will be handled in the concern route
    })
    # When rendering chat.html after a POST, pass the necessary data
    return render_template("chat.html", sentiment=sentiment)
```

Concern Submission Route:

```
@app.route('/concern', methods=['POST'])
@login_required # Requires login to submit a concern
def submit_concern():
    concern = request.form['concern']
    history.append({
        'date': datetime.now(),
        'sentiment': 'Neutral', # Sentiment might be added later or remain neutral if only concern is submitted
        'issue': concern
    })
    # When rendering chat.html after a POST, pass the necessary data
    return render_template("chat.html", concern_submitted=True)
```

Dashboard page Route:

```
@app.route('/dashboard')
@login_required # Requires login to access
def dashboard():
    # Dashboard page - requires login
    last_7_days = datetime.now() - timedelta(days=7)
    filtered = [h for h in history if h['date'] > last_7_days]

    sentiment_counts = {'Positive': 0, 'Neutral': 0, 'Negative': 0}
    issues = []

    for h in filtered:
        sentiment_counts[h['sentiment']] += 1
        if h['issue']:
            issues.append(h['issue'])

    return render_template("dashboard.html", data={
        'positive': sentiment_counts['Positive'],
        'neutral': sentiment_counts['Neutral'],
        'negative': sentiment_counts['Negative'],
        'issues': issues
    })
```

Login page Route:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Login page - does NOT require login
    if request.method == 'POST':
        username = request.form.get('username') # Use .get() for safer access
        password = request.form.get('password') # Use .get() for safer access

        # --- Basic Authentication (Replace with secure method) ---
        # In a real application, you would check username and password against a database
        # Updated credentials
        if username == "lasya@gmail.com" and password == "lasya":
            session['user'] = username # Store username in session
            # Redirect to the page the user was trying to access, or to the about page if no next URL
            next_url = request.args.get('next')
            # Redirect to the about page if no specific 'next' URL was provided
            return redirect(next_url or url_for('about'))
        else:
            # Handle invalid login (e.g., show an error message on the login page)
            return render_template("login.html", error="Invalid email or password")
    # For GET request, render the login form
    return render_template("login.html")
```

Logout page Route:

```
@app.route('/logout')
def logout():
    # Logout route - logs the user out and redirects to index
    session.pop('user', None) # Remove user from session
    return redirect(url_for('index')) # Redirect to home page after logout
```

Milestone 4: Frontend Development

This milestone focuses on creating the user interface for CitizenAI. It involves designing the layout of each page using HTML and applying styling with CSS to ensure a user-friendly and visually appealing experience. Building interactive components like forms for chat, login, feedback, and concerns, and ensuring they correctly display data from the backend, are key tasks.

Activity 4.1: 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.

Activity 4.2: 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.

Activity 4.2.1: Bind Backend Data to HTML Templates

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.

Milestone 5: Deployment

This milestone focuses on preparing and launching the CitizenAI application. It involves setting up the deployment environment, ensuring all necessary Python libraries (Flask, PyTorch, Hugging Face libraries) and dependencies are correctly installed. The primary goal is to launch the Flask application locally to verify that the app runs smoothly in a test environment, allowing for final testing and refinement before potential cloud deployment.

Activity 5.1 : 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.

5.2: Configure Environment Variables

```python
# ------------------ Model Setup ------------------
# Note: Running large models locally requires significant resources (GPU, RAM)
model_path = "ibm-granite/granite-3.3-8b-instruct"

# Determine the device to use (GPU if available, otherwise CPU)
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# Load the tokenizer first
tokenizer = AutoTokenizer.from_pretrained(model_path)
```

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).

Activity 5.2: Testing and Verifying Local Deployment

1. Start the Flask Application

Run the following command to launch the application locally:

Python
app.py

Milestone 6: functional testing and verify

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.
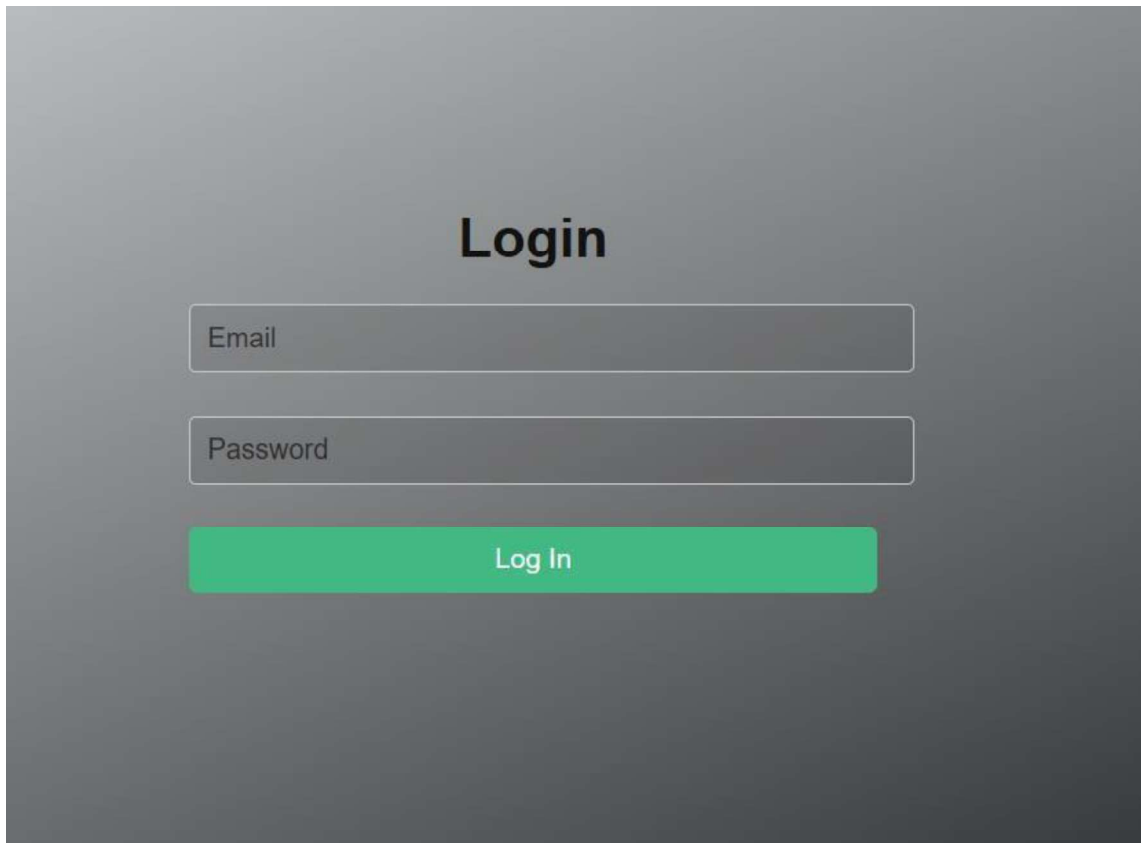
3. **Right Panel (Visual):**

   - 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.
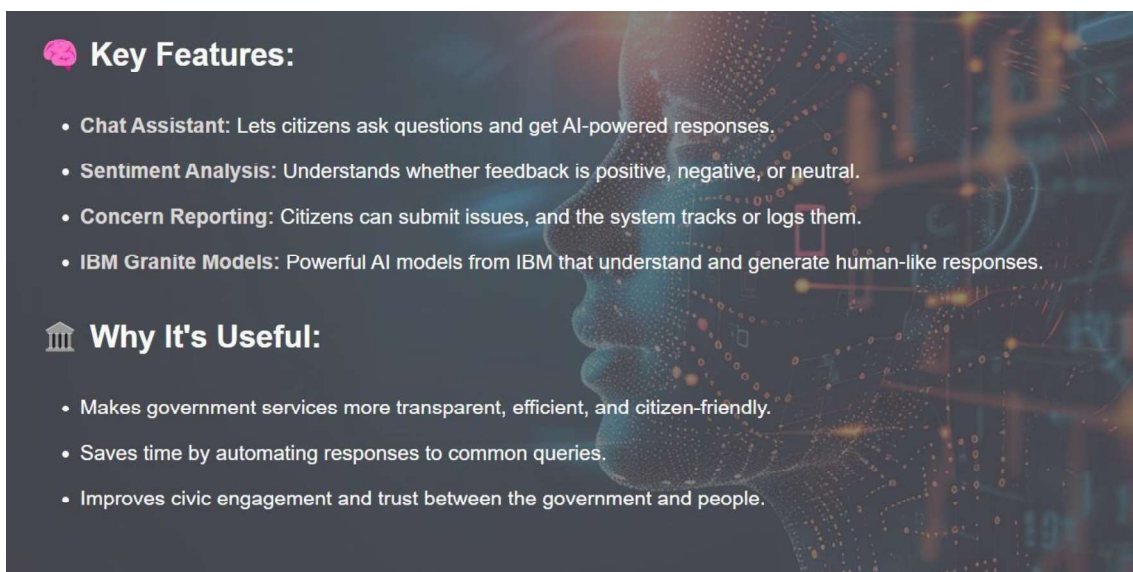
About   **Page:**

About CitizenAI

Back to Home

CitizenAI: CitizenAI is a platform designed to improve how governments and citizens interact, using artificial intelligence (AI) to make public services smarter, faster, and more accessible.

Our mission is to simplify civic interactions, encourage transparency, and foster trust by providing a smart, responsive, and accessible interface for every citizen. Whether you're submitting feedback, accessing government services, or exploring insights—CitizenAI is your bridge to a smarter society.

🚀 In Simple Terms:

Think of CitizenAI as a digital assistant for public service—just like a chatbot or an AI helpdesk, but for government-related tasks like:

- Asking questions about public services
- Reporting issues (like potholes, pollution, or delays)

The 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 and trust.

🧠 Key Features:

- **Chat Assistant:** Lets citizens ask questions and get AI-powered responses.
- **Sentiment Analysis:** Understands whether feedback is positive, negative, or neutral.
- **Concern Reporting:** Citizens can submit issues, and the system tracks or logs them.
- **IBM Granite Models:** Powerful AI models from IBM that understand and generate human-like responses.

🏛 Why It's Useful:

- Makes government services more transparent, efficient, and citizen-friendly.
- Saves time by automating responses to common queries.
- Improves civic engagement and trust between the government and people.

Chat Page:

Citizen Chat Assistant

Ask the Assistant

Ask a Question:

Submit

Please fill out this field.

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:

Ask the Assistant

Ask a Question:

how to apply for a group A exams

Submit

Response:

- To apply for Group A exams in India, follow these steps:
- 1. Check eligibility: Ensure you meet the educational qualifications and age limit requirements for the specific Group A exam you wish to appear for.
- 2. Visit the official website: Go to the official website of the respective exam conducting body, such as UPSC (Union Public Service Commission) for Civil Services Exam, SSC (Staff Selection Commission) for various exams, or RRB (Railway Recruitment Board) for Railway exams.
- 3. Register: Register on the official website by providing your personal details, contact information, and educational qualifications.
- 4. Pay application fee: Pay the required application fee through the available online payment options.
- 5. Fill the application form: Fill in the application form with accurate details and upload the necessary documents, such as photograph, signature, and educational certificates.
- 6. Submit the application: Submit the filled application form and pay the application fee, if not done earlier.
- 7. Download and print: Download and print the application form and fee receipt for future reference.
- 8. Keep a copy: Maintain a copy of the application form and fee receipt for any future correspondence or verification.
- 9. Prepare for the exam: Start your preparation for the exam by following a study plan and referring to relevant study materials.
- 1
- 0. Stay updated: Keep an eye on the official website for any updates, notifications, or changes regarding the exam date, syllabus, or other important information.

Response 2:

Ask the Assistant

Ask a Question:

how to renew my driving license

Submit

Response:

- in india To renew your driving license in India, follow these steps:
- 1. Check eligibility: Ensure your current DL is valid and you meet the age and other requirements.
- 2. Gather documents: Prepare necessary documents like application form, proof of address, passport size photo, and fee payment receipt.
- 3. Fill the application form: Download the form from the RTO website or get it from the RTO office. Fill it accurately.
- 4. Pay the fee: Pay the required fee at the designated counter or online, depending on your RTO.
- 5. Submit documents: Submit the filled form and required documents along with the fee at the RTO office.
- 6. Biometric registration: Some RTOs may require biometric registration. Follow the instructions given at the RTO.
- 7. Collect the renewed DL: Once the process is complete, collect your renewed driving license from the RTO office. Please visit your local RTO website for specific instructions and fees, as they may vary.

Sentiment Analysis:

Sentiment 1

Feedback Sentiment

Enter Feedback for Sentiment:

It's very good Response

Submit

Sentiment:

Positive

Sentiment 2:

Feedback Sentiment

Enter Feedback for Sentiment:

Its a bad response

Submit

Sentiment:

Negative

Sentiment 3:

Feedback Sentiment

Enter Feedback for Sentiment:

Its a neither good nor a bad response

Submit

Sentiment:

Neutral

Concern Reporting:

Report a Concern

Describe Your Concern (for Issue Identification):

there is street light problem in our area

Submit

Concern Submitted:

Your concern has been recorded.

Dashboard page:

Citizen Insights Dashboard

Weekly Sentiment Analysis

Positive	Neutral	Negative
3	2	1

Recent Citizen Issues

there is street light problem in our area

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.