# Introduction

**Projecttitle**: **Citizen AI**

Teammember: JAYARAJ.J

Teammember: ASHWIN.S

Teammember: AYYAPPAN.A

Teammember: SANTHOSH.D Citizen AI: Intelligent Citizen Engagement Platform

Generative AI with IBM

# Project overview

# 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.

1. **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.

1. **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.
2. **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:

1. 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:

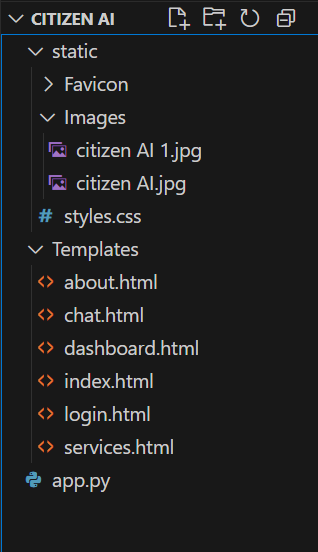
* Install the necessary libraries for AI model integration:

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 Functionaliton



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.

# https://lh7-rt.googleusercontent.com/docsz/AD_4nXeONCHyFFnM0vtbqc7coujT6EWX76vA0ooBq1LN09-jRIZX1MpnbAqhJVWqH1HiEN5YC8it-5zaFZRyY4Be4U9mrO3Qv8dZbXNP_Fn63EAsgyJqkZkn09Ql2PNjisNGwYqH_QWXGw?key=ih1a9HwgikqjODzsSadv1A

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

%}.

1. If triggered, a <div class="response"> appears with a success message.
2. 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.

# 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.

1. /feedback ? Accepts user feedback text and performs sentiment analysis on

it.

1. /concern ? Accepts a user's report of a concern or issue for logging.
2. /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.

# 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.

# 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).

### 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.

# 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.

1. 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.

# 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).

# 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

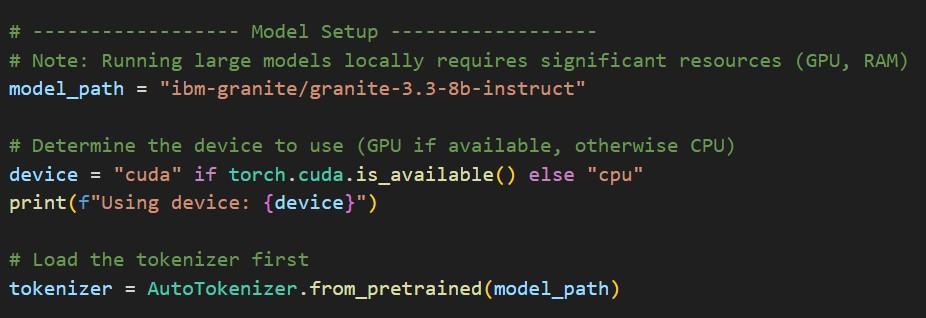
To ensure dependency isolation, create and activate a virtual environment before installing required packages.

“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

. 

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:

# Header Section:

* + Shows the main title: "Welcome to CitizenAI".
  + Contains navigation links: About, Services, Chat, Dashboard, and Login.

# 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.

# 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.

# 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**.

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.

# 