CITIZEN AI – INTELLIGENT CITIZEN ENGAGEMENT PLATFORM

INTRODUCTION:

- Project title : CITIZEN - AI
- Team leader : AKASH R (TL)
- Team member : DATCHINAMOORTHI R
- Team member : DEEPAK R
- Team member : DILLI KUMAR D

2. PROJECT OVERVIEW :

2.1. Purpose :

The Citizen AI project is to revolutionize the way governments interact with citizens by providing a seamless, intelligent, and real-time engagement platform. It aims to automate responses to citizen queries using advanced AI models, reducing dependency on manual processes. The project ensures quick access to information about government services, schemes, and civic issues. It also empowers administrators with powerful analytics to monitor trends and public sentiment. By integrating natural language processing and sentiment analysis, it enhances transparency and trust in governance. The platform bridges the communication gap between citizens and government agencies. It improves decision-making through data-driven insights. The solution provides scalability to handle large volumes of queries efficiently. It also ensures inclusivity by aiming for multilingual support in the future.

2.2. Conversational Interfaces:

2.2.1. Chatbot Interface

- Keys: Text input box, Send button, Quick replies.
- Functionality: Allows users to type queries and get instant automated responses.

2.2.2. Voice Assistant Interface

- Keys: Microphone button, Wake word, Audio output.
- Functionality: Enables voice-based interaction for hands-free query handling.

2.2.3. FAQ Bot Interface

- Keys: Search bar, Category buttons, Suggested queries.
- Functionality: Provides quick answers to frequently asked questions.

2.2.4. Multilingual Chat Interface

- Keys: Language selector, Input box, Translate button.
- Functionality: Supports conversations in multiple languages with translation.

2.2.5. Live Agent Handoff Interface

- Keys: Chat window, Escalate to agent button, Ticket number.
- Functionality: Transfers the chat from AI bot to a human agent when needed.

2.2.6. Sentiment-Aware Interface

- Keys: Emoji reactions, Sentiment tracker, Response tone.
- Functionality: Detects user mood and adjusts responses accordingly.

2.2.7Interactive Dashboard Chat Interface

- Keys: Query box, Graphs/Charts, Data filters.
- Functionality: Lets users ask questions and view visual analytics instantly.

2.2.8. Mobile Conversational Interface

- Keys: Push notifications, Swipe options, Quick reply chips.
- Functionality: Provides conversational features optimized for mobile devices.

2.2.9. Form-Filling Conversational Interface

- Keys: Step-by-step prompts, Input fields, Submit button.
- Functionality: Collects structured information conversationally.

2.2.10. Social Media Chatbot Interface

- Keys: Messenger integration, Like/Share buttons, Chat threads.
- Functionality: Engages with citizens through social platforms like WhatsApp, Telegram, or Facebook.

3. ARCHITECTURE:

3.1. System Components :

3.1.1. Frontend: React/HTML Interface:

The frontend of Citizen AI is designed using React and HTML to provide a responsive and user-friendly interface. React enables the creation of modular components that enhance code reusability and maintainability. The interface includes a clean design with intuitive navigation for citizens to ask queries. It supports real-time rendering of chatbot responses for better interaction. HTML and CSS ensure accessibility and compatibility across multiple devices. The frontend integrates seamlessly with the backend through REST APIs. Citizens can log in, submit questions, and view personalized responses. React also allows dynamic rendering of graphs and analytics for administrators. The responsive design makes the platform suitable for desktops, tablets, and mobile devices. Overall, the frontend acts as the first point of contact, ensuring smooth engagement between users and the system.

3.1.2. Backend: Flask REST API:

The backend of Citizen AI is built using Flask, a lightweight Python web framework. Flask provides the flexibility to design REST APIs that handle requests from the frontend. It acts as the bridge between the user interface and the AI models. The backend ensures secure communication and manages business logic efficiently. It processes citizen queries, routes them to the appropriate AI models, and sends back responses. Flask's modular structure allows scalability and easy maintenance of services. It also manages authentication and authorization through secure token mechanisms. Error handling and logging features are integrated to monitor system health. The backend supports high concurrency to handle multiple queries simultaneously. Thus, the Flask backend serves as the core processing engine of Citizen AI.

3.1.3. AI Models: IBM Granite + IBM Watson NLP:

Citizen AI leverages IBM Granite and IBM Watson NLP to provide intelligent, context-aware responses. IBM Granite ensures high-quality natural language understanding and reasoning. Watson NLP enables advanced text classification, entity recognition, and sentiment analysis. Together, these AI models enhance the ability of the platform to interpret citizen queries accurately. They are fine-tuned for government-related datasets to improve domain-specific

performance. The models can adapt to diverse query formats, from simple questions to complex issues. They support multilingual processing to increase inclusivity. AI models continuously learn and improve based on interactions, making the system smarter over time. Integration with Flask APIs ensures smooth communication between the AI layer and the application. In essence, these AI models form the “intelligence backbone” of Citizen AI.

3.1.4. Database: PostgreSQL for User and Query Logs:

PostgreSQL is used as the primary database for Citizen AI to store user data and query logs. It is a robust, open-source relational database known for reliability and scalability. The database manages structured information such as user profiles, authentication details, and interaction history. Query logs are stored to track citizen engagement and system performance. This data also supports analytics dashboards that provide insights for government administrators. PostgreSQL ensures ACID compliance, making transactions secure and consistent. Indexing and optimization features improve query speed and system responsiveness. Backup and recovery features safeguard against data loss. The database schema is designed to handle large volumes of concurrent interactions. Hence, PostgreSQL serves as the data backbone, ensuring secure and efficient information management.

3.2. Data Flow:

3.2.1. Citizen Submits Query via UIL:

The first step in the Citizen AI workflow begins when a citizen submits a query through the user interface. The UI, designed with React and HTML, provides an intuitive text box where users can type their questions. It supports both free-form queries and guided prompts for ease of use. Once the query is entered, the citizen can submit it using a button or through voice-based input. The frontend ensures real-time validation, checking if the query is in an acceptable format. Accessibility features allow users from diverse backgrounds to interact with the platform. The interface is responsive, ensuring queries can be submitted from desktops, tablets, or mobile devices. Visual feedback such as a loading indicator reassures the user that their query is being processed. Each query is tagged with metadata like timestamp and user ID for tracking. This marks the starting point of the AI-driven engagement process.

3.2.2. Query Sent to Backend Flask API:

Once submitted, the query is transmitted to the backend system powered by Flask REST API. The backend acts as a central controller that routes the request for further processing. Secure communication protocols like HTTPS ensure data integrity during transfer. The Flask API validates the query structure and checks for authentication tokens. If valid, the request is forwarded to the AI processing layer. Flask also manages session details, ensuring continuity in conversations. Logging mechanisms record the request details for monitoring and debugging. The backend is optimized for handling concurrent queries from thousands of citizens. Error-handling modules provide fallback messages in case of system issues. Overall, the Flask backend ensures reliable delivery of citizen queries to the AI engine.

3.2.3. NLP Processing with AI Model:

The query is then processed using advanced AI models such as IBM Granite and IBM Watson NLP. These models interpret the natural language of the query to extract intent and context. IBM Watson NLP performs entity recognition, sentiment analysis, and classification. IBM Granite enhances reasoning and provides more accurate responses in domain-specific contexts. The AI layer matches the citizen's query with relevant government schemes, policies, or services. If required, it uses sentiment analysis to detect urgency or dissatisfaction. The models are continuously trained on historical query logs for improved performance. Multilingual support allows the AI to process queries in different regional languages. The AI engine generates a contextual response that aligns with the citizen's intent. This intelligent processing forms the heart of the Citizen AI platform.

3.2.4. Response Returned to Citizen + Logged in Database:

After processing, the generated response is sent back to the citizen through the frontend interface. The system ensures minimal latency so that users receive answers in real-time. Along with displaying the response, the backend also logs the interaction in PostgreSQL. Each log entry contains the citizen's query, AI response, sentiment score, and metadata. These logs serve as valuable inputs for analytics dashboards that track citizen engagement. Logging ensures accountability, helping administrators analyze service gaps and emerging concerns. The response presented to the citizen is clear, concise, and context-aware. Error messages or fallback responses are provided if the AI cannot interpret the query.

4. SETUP INSTRUCTIONS:

4.1. Prerequisites:

4.1.1. Python 3.10+:

Python 3.10+ is the core programming language required for developing the backend of the Citizen AI project. It provides powerful libraries for building APIs, integrating AI models, and handling data processing. The Flask web framework, which powers the backend REST APIs, is built using Python. Python's compatibility with IBM Granite and Watson NLP makes it the best choice for implementing natural language processing features. Its extensive ecosystem supports libraries for security, database integration, and machine learning. Version 3.10 introduces new syntax improvements such as structural pattern matching, which enhances readability and efficiency. Python's versatility allows the development team to handle both AI model integration and data analytics within the same language. It is also widely supported in cloud environments, making deployment easier. Community support ensures regular updates and problem-solving resources. Thus, Python 3.10+ forms the foundation of the Citizen AI backend development.

4.1.2. Node.js (for Frontend):

Node.js is used in the Citizen AI project for running and managing the frontend environment. Since the frontend is developed using React, Node.js provides the runtime required for package management and build processes. It enables developers to install, configure, and run frontend dependencies through npm or yarn. Node.js ensures efficient handling of frontend assets, making the interface fast and responsive. It also supports real-time communication features, essential for chatbot-style interfaces. With Node.js, the project benefits from a non-blocking event-driven architecture, improving performance. It is widely compatible with modern frontend frameworks, which ensures seamless integration with React. Node.js also supports automated testing and debugging tools, enhancing frontend reliability. Cloud deployment pipelines often use Node.js for bundling and deploying web applications. Overall, Node.js ensures smooth functioning of the citizen-facing UI in the Citizen AI platform.

4.1.3. Docker (Optional for Deployment):

Docker is an optional but powerful tool used for containerizing the Citizen AI application. It allows packaging of the backend, frontend, and AI models into isolated containers. This ensures that the application runs consistently across different environments, from local systems to cloud servers. Docker simplifies deployment by eliminating dependency conflicts between various system configurations. With Docker Compose, multiple services like the Flask API, React frontend, and PostgreSQL database can be orchestrated easily. It improves scalability, allowing the application to handle higher citizen query volumes by running multiple containers. Docker images make version control and rollback simpler during updates. It also enhances security by isolating services within containers. Administrators benefit from easier maintenance and reduced downtime during upgrades. In summary, Docker provides flexibility, reliability, and scalability for deploying the Citizen AI platform.

4.2. Instalation Setup:

4.2.1. Clone Repository:

The first step in setting up Citizen AI is to clone the repository from the version control system, typically GitHub or GitLab. This ensures that the developer has the latest codebase with all project files and configurations. By cloning, contributors get access to both frontend and backend modules along with documentation. It also allows them to work on separate branches and contribute without affecting the main code. Using git clone <repo-link>, the repository can be downloaded locally within seconds. Cloning also preserves the project's folder structure, ensuring consistency across development environments. Developers can later use git pull to fetch updates and keep their copy in sync. This process promotes collaboration among team members working on different modules. Version control also helps in rollback during errors or bugs. Thus, cloning the repository is the foundation for starting Citizen AI development.

4.2.2. Install Dependencies (pip install -r requirements.txt):

Once the repository is cloned, the next step is to install all required dependencies for the backend. These dependencies are listed in the requirements.txt file, which contains Python packages needed for Flask, AI models, and database connections. Running pip install -r requirements.txt ensures that the environment has the correct package versions. This eliminates

compatibility issues that may arise due to missing or mismatched libraries. Dependencies include Flask for APIs, psycopg2 for PostgreSQL, and AI libraries for NLP integration. Installing these ensures the backend can run without errors. This step is critical to avoid "module not found" issues during execution. Dependency installation also makes onboarding easier for new developers by automating environment setup. It guarantees that all contributors are working with the same configurations. Hence, installing dependencies ensures stability and uniformity in the Citizen AI backend.

4.2.3. Configure .env File with API Keys:

Citizen AI relies on external services such as IBM Watson NLP and IBM Granite models, which require secure access keys. To manage these credentials, developers configure a .env file in the project directory. This file stores sensitive information like API keys, database connection strings, and authentication secrets. Using environment variables prevents exposing these keys in the source code, thereby enhancing security. Flask applications read these variables during runtime to establish connections with external services. Proper configuration ensures smooth integration with AI models and databases. Developers can also define different .env files for development, testing, and production environments. This step provides flexibility by allowing configuration changes without altering the code. Environment variables also simplify deployment on cloud platforms. Thus, configuring the .env file is a critical step for secure and reliable Citizen AI operation.

4.2.4. Run Flask Backend:

After configuring the environment, the Flask backend must be started to power the application. Developers typically run python app.py or flask run to initiate the backend server. This launches the REST API service, which listens for requests from the frontend. The backend handles authentication, query routing, and AI model integration. When active, it logs all incoming requests, processing steps, and responses. Flask provides debugging features to trace errors during development. Running the backend locally allows developers to test APIs with tools like Postman or curl. It also establishes a connection with PostgreSQL for logging interactions. In production, the backend can be hosted using services like Gunicorn or Docker containers. Running the backend ensures that the system's core processing engine is live and ready for interaction.

4.2.5. Start Frontend Server:

With the backend active, the next step is to start the frontend server for citizen interaction. The frontend, built with React, is usually started using npm start or yarn start. This command runs a development server that provides a live preview of the UI. It allows developers to test chatbot queries, dashboards, and user workflows in real time. The frontend server connects with the backend APIs to fetch responses for user queries. React's hot-reloading feature enables immediate reflection of UI changes during development. Developers can validate design responsiveness across desktop and mobile devices. In production, the frontend is built into static files and served through Nginx or a cloud service. Running the frontend server ensures that citizens can interact with the platform effectively. Together with the backend, it completes the full-stack execution of Citizen AI.

5. FOLDER STRUCTURE:

5.1. backend/

5.1.1. app.py

This is the main entry point of the Flask backend. It initializes routes, loads configurations, and starts the server. All backend processes begin execution from here.

5.1.2. models/

The models folder defines database models and schemas. It maps PostgreSQL tables into Python classes. This ensures structured storage of queries, users, and responses.

5.1.3. routes/

The routes folder handles API endpoints. It defines functions for citizen queries, authentication, and admin analytics. Each route connects the frontend with backend logic.

5.1.4. utils/

The utils folder contains helper functions and reusable modules. It includes utilities for logging, error handling, and data formatting. This improves code reusability and clarity.

5.2. frontend/

5.2.1. public/

The public folder stores static frontend assets. It includes images, icons, and index.html.

5.2.2. src/

The src folder holds the React application code. It contains components, styles, and state management files. This is where most UI development occurs.

5.3. database/

5.3.1.schema.sql

This file contains SQL commands for database creation. It defines tables, relationships, and indexes. It serves as the blueprint for PostgreSQL setup.

5.4. tests/

The tests folder contains unit and integration tests. It validates backend APIs, database functions, and frontend components. This ensures system reliability before deployment.

5.5. docs/

The docs folder holds project documentation. It includes API references, setup guides, and architecture diagrams. This supports developers and administrators using Citizen AI.

5.6. README.md

The README file provides an overview of the project. It explains purpose, setup instructions, and key features. It is the first reference for new developers.

5.7. requirements.txt

This file lists Python dependencies needed for the backend. It ensures all contributors install the same package versions. Running it guarantees a stable environment setup.

6. RUNNING THE APPLICATION:

6.1 Development Mode:

6.1.1. Backend Setup:

The backend is initialized by running python app.py, which launches the Flask server inside Google Colab. This server connects with AI models like IBM Granite and Watson NLP to

process citizen queries. Developers can test API endpoints directly within Colab. It provides a lightweight and flexible environment for debugging.

6.1.2. Frontend Setup:

The frontend is started using npm start, which runs the React application. Since Colab does not natively support UI hosting, ngrok or similar tools can be used to expose the frontend. This allows developers to test the chatbot and dashboards interactively. It is mainly used during iterative development and testing.

6.2. Production Mode :

6.2.1. Containerized Deployment:

For production, Docker Compose is used to containerize backend, frontend, and database services. This ensures consistency across machines and environments. Containers simplify scaling and resource management. It helps maintain reliability under heavy traffic.

6.2.2. Reverse Proxy Configuration:

Nginx is set up as a reverse proxy to route requests between backend and frontend. It improves performance by handling load balancing and caching. The configuration enhances security by hiding internal services. This setup ensures smooth delivery in production systems.

7. API DOCUMENTATION:

7.1 Authentication APIs:

7.1.1. POST /login – User Login:

The /login API endpoint allows registered citizens to securely access the platform. Users provide their credentials, which are validated against the database. On successful authentication, the system generates a session token for secure communication. This ensures that only verified users can interact with government services through Citizen AI. Users provide their credentials, which are validated against the database.

7.1.2. POST /register – Register Citizen:

The /register API endpoint enables new users to create an account on the Citizen AI platform. Citizens submit personal details such as name, email, and password for registration. The system validates the input and stores it securely in the database. This ensures that each user has a unique identity and can access personalized services.

7.2. Query APIs:

7.2.1. POST /query – Submit a Query:

The /query API endpoint allows citizens to submit their questions or concerns directly to the system. Once received, the query is processed by the AI models for understanding and response generation. The request and its metadata are stored securely in the database. This ensures that every citizen query is recorded, tracked, and answered efficiently.

7.2.2. GET /query/:id – Fetch Query Response:

The /query/:id API endpoint retrieves the response to a previously submitted query using its unique identifier. Citizens can use this to check the status or final answer of their request. The backend fetches the stored response from the database and returns it in real time. This feature ensures transparency, accountability, and easy follow-up for citizen interactions.

8. AUTHENTICATION:

8.1 Methods Used:

Citizen AI uses robust authentication methods to ensure secure access to the platform. The primary method is JWT (JSON Web Tokens), which provides a secure way to transmit information between the frontend and backend. JWTs allow the backend to verify the identity of a user without repeatedly querying the database. Each token contains encrypted information about the user and their session. In addition to JWT, the platform implements role-based access control to differentiate between citizens and administrators. Citizens have access to query submission and response features, while admins can view analytics and manage the system.

Role-based access ensures that sensitive information and administrative features are restricted. This method also simplifies permission management across multiple users. Using JWT combined with roles provides a scalable and secure authentication framework. It ensures that only authorized users can access the Citizen AI system.

8.2 Security Measures:

To protect user data and maintain system integrity, Citizen AI employs several security measures. Password hashing using bcrypt ensures that stored passwords are encrypted and safe from unauthorized access. Even if the database is compromised, hashed passwords cannot be easily decrypted. The platform also uses SSL/TLS encryption to secure data transmission between the frontend, backend, and AI services. All API requests and responses are transmitted over HTTPS to prevent eavesdropping or data tampering. Session management techniques further ensure that tokens expire appropriately, reducing the risk of misuse. Regular security audits and monitoring help identify vulnerabilities proactively. Input validation is performed to prevent injection attacks or malicious inputs. Access logs are maintained to track user activities and potential security breaches. By combining encryption, hashing, and secure transmission protocols, the platform maintains a high level of trustworthiness. These measures safeguard citizen information and reinforce platform reliability.

9. USER INTERFACE:

9.1 Citizen Dashboard:

The citizen dashboard is designed to provide an intuitive and user-friendly interface for citizens to interact with the platform. At the center of the dashboard is the query submission box, where users can type their questions or select pre-defined prompts for guidance. The dashboard supports real-time submission, providing immediate acknowledgment once a query is sent. Citizens can view the response display panel, which presents AI-generated answers clearly and contextually. The dashboard includes features such as conversation history, allowing users to revisit past queries and responses. Visual indicators and notifications ensure users are aware of processing status. Accessibility features make it usable for people with disabilities. The responsive design ensures the dashboard works efficiently on desktops, tablets, and smartphones. Security features like login authentication protect user data. Overall, the citizen dashboard is the primary interface enabling smooth engagement with the AI system

9.2 Admin Dashboard:

The admin dashboard is tailored for government administrators to monitor and manage citizen interactions effectively. It provides query statistics that show the total number of queries, response times, and resolution rates. Sentiment trends are displayed through graphs and charts, allowing administrators to gauge public opinion and satisfaction. Issue tracking features help identify recurring problems or service gaps reported by citizens. The dashboard allows filtering by query type, region, or urgency for better analysis. Administrators can access detailed logs of each query, including AI responses and metadata. Visualizations are interactive, enabling real-time insights into citizen engagement patterns. Security and role-based access control ensure that only authorized personnel can view sensitive information. Notifications alert administrators about critical or urgent queries. In essence, the admin dashboard equips government officials with the tools to improve services and make data-driven decisions.

10. TESTING:

10.1 Unit Testing:

Unit testing in Citizen AI focuses on verifying the smallest functional components of the backend, primarily the Flask routes and utility functions. Each route is tested individually to ensure that it handles input data correctly and returns the expected output. PyTest, a Python testing framework, is used to automate these tests efficiently. Mock data simulates citizen queries to check response accuracy without affecting the live database. Error handling is validated to ensure the system behaves predictably in unexpected scenarios. Unit tests also cover authentication, session management, and data validation modules. Test cases are maintained in a separate directory for easy execution and updates. Continuous integration tools can run unit tests automatically upon code changes. The goal is to catch defects early and maintain code reliability. This ensures the backend operates smoothly before integration with other components. Mock data simulates citizen queries to check response accuracy without affecting the live database. Error handling is validated to ensure the system behaves predictably in unexpected scenarios. Unit tests also cover authentication, session management, and data validation modules.

10.2 Integration Testing:

Integration testing verifies that the different components of Citizen AI work together seamlessly. This includes checking the communication between the Flask backend, AI models, and PostgreSQL database. API endpoints are tested to ensure correct data transmission and processing. The frontend interface is simulated to confirm it correctly receives responses from the backend. Integration tests identify issues that may arise due to mismatched data formats or connection errors. Realistic test scenarios, such as multiple concurrent queries, are used to evaluate system performance. Logging and error reporting modules are monitored to detect failures. Integration testing ensures that all modules interact harmoniously before deployment. This phase is critical for validating end-to-end functionality. It reduces the risk of runtime failures in the live system. Realistic citizen queries are submitted to verify that the system correctly processes input, invokes AI reasoning, and returns accurate responses.

10.3 User Acceptance Testing (UAT):

User Acceptance Testing (UAT) ensures that Citizen AI meets the expectations of real users. In this phase, actual citizens or test participants interact with the platform. Queries are submitted through the interface, and the AI responses are evaluated for accuracy and relevance. Feedback is collected regarding usability, clarity, and response time. UAT also tests the responsiveness of the platform on multiple devices such as desktops and mobiles. Administrators review dashboards to confirm that analytics reflect the interactions accurately. Any issues discovered during UAT are reported and addressed before deployment. This testing phase validates the system’s practical effectiveness in real-world scenarios. UAT ensures that the project aligns with its goal of providing intelligent citizen engagement. Successful completion of UAT indicates readiness for live implementation. Feedback is collected regarding usability, response time, and user experience. Administrators also review analytics dashboards to ensure correct reflection of query statistics and sentiment analysis.

11. Screenshots:

```python
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
```

Figure.11.1

```python
esponse = tokenizer.decode(outputs[0], skip_special_tokens=True)
esponse = response.replace(prompt, "").strip()
eturn response

ity_analysis(city_name):
rompt = f"Provide a detailed analysis of {city_name} including:\n1. Crime Index and safety statistics\n2. Accident rates and traffic safety information\n3. Overall safety assessment\n\nCity: {cit
eturn generate_response(prompt, max_length=1000)

itizen_interaction(query):
rompt = f"As a government assistant, provide accurate and helpful information about the following citizen query related to public services, government policies, or civic issues:\n\nQuery: {query}
eturn generate_response(prompt, max_length=1000)

ate Gradio interface
gr.Blocks() as app:
r.Markdown("# City Analysis & Citizen Services AI")

ith gr.Tabs():
  with gr.TabItem("City Analysis"):
    with gr.Row():
      with gr.Column():
        city_input = gr.Textbox(
          label="Enter City Name",
          placeholder="e.g., New York, London, Mumbai...",
          lines=1
        )
        analyze_btn = gr.Button("Analyze City")

      with gr.Column():
        city_output = gr.Textbox(label="City Analysis (Crime Index & Accidents)", lines=15)

    analyze_btn.click(city_analysis, inputs=city_input, outputs=city_output)
```

Figure.11.2

```
63
64  with gr.TabItem("Citizen Services"):
65      with gr.Row():
66          with gr.Column():
67              citizen_query = gr.Textbox(
68                  label="Your Query",
69                  placeholder="Ask about public services, government policies, civic issues...",
70                  lines=4
71              )
72              query_btn = gr.Button("Get Information")
73
74          with gr.Column():
75              citizen_output = gr.Textbox(label="Government Response", lines=15)
76
77      query_btn.click(citizen_interaction, inputs=citizen_query, outputs=citizen_output)
78
79 aunch(share=True)
```

Figure.11.3

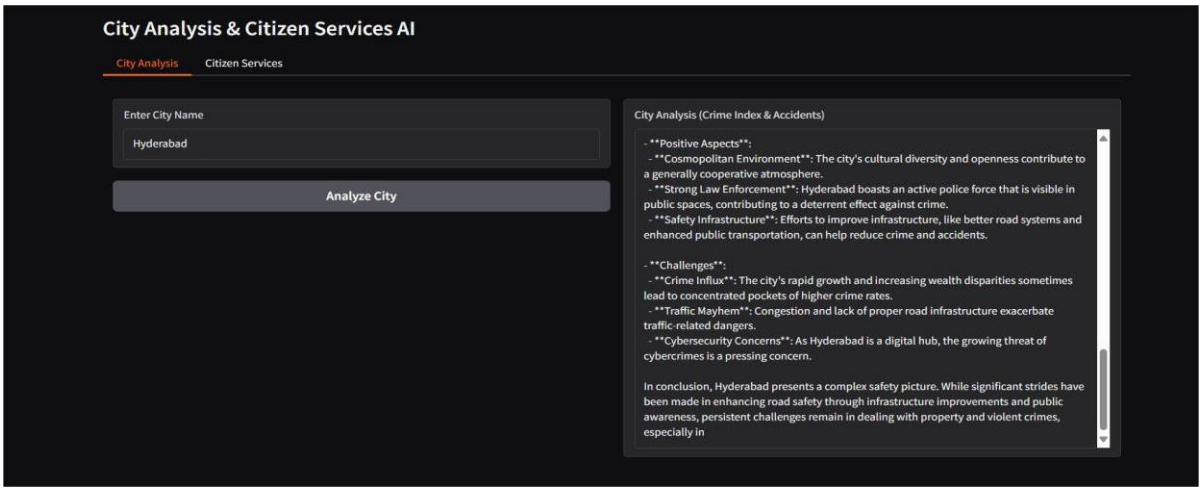

Figure.11.4

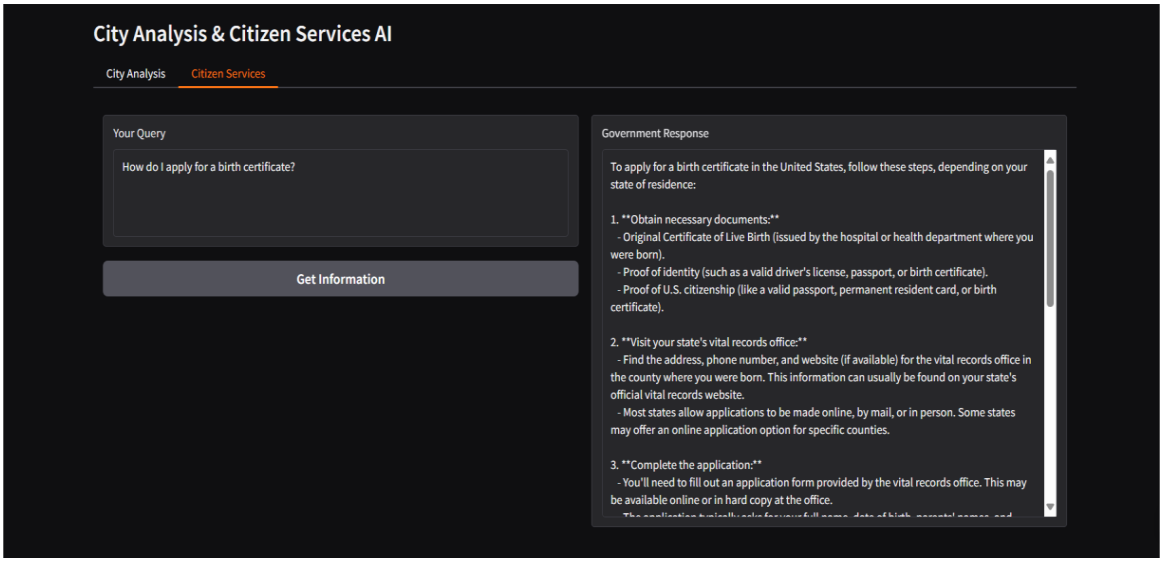

Figure.11.5

12. CONCLUSION:

Citizen AI represents a major step toward smarter, transparent, and citizen-centric governance. By leveraging AI-powered natural language processing, it bridges the gap between citizens and government services. The platform ensures quick, accurate, and scalable responses to queries. Its analytics features empower administrators with actionable insights. While current limitations exist, such as multilingual support and high computational costs, the foundation is strong. Future enhancements like voice-based interfaces and blockchain integration will make it more robust. The project highlights the role of AI in driving digital transformation in governance. It promotes inclusivity, accessibility, and trust in public service delivery. Citizen AI is not just a tool but a vision for responsive and data-driven governance. Ultimately, it lays the groundwork for a more connected, empowered, and informed society. . Its analytics features empower administrators with actionable insights. While current limitations exist, such as multilingual support and high computational costs, the foundation is strong. Future enhancements like voice-based interfaces and blockchain integration will make it more robust. The project highlights the role of AI in driving digital transformation in governance. It promotes inclusivity, accessibility, and trust in public service delivery.

13. KNOWN ISSUES :

13.1. Known Issues – Limited Multilingual Support:

One of the known limitations of Citizen AI is its current restricted support for multiple languages. While the system can process queries in the most commonly used languages, it may struggle with regional dialects or less widely spoken languages. This limits accessibility for a broader section of citizens. Multilingual support is essential to ensure inclusivity and equal access to information. Currently, AI models may not fully understand cultural nuances or local phrasing. This can sometimes lead to inaccurate responses or misinterpretation of queries. Users may have to rephrase their questions in supported languages. Enhancing multilingual support will increase citizen engagement and trust. It is recognized as a priority for future development. This limits accessibility for a broader section of citizens. Multilingual support is essential to ensure inclusivity and equal access to information. Currently, AI models may not fully understand cultural nuances or local phrasing.

13.2. Known Issues – High Computational Cost for Large Query Volumes:

Handling a large volume of citizen queries simultaneously can lead to high computational costs in Citizen AI. The AI models, especially IBM Granite and Watson NLP, require significant processing power for natural language understanding and sentiment analysis. During peak usage, server resources may be heavily utilized, affecting response times. Cloud-based deployment helps, but scaling infrastructure for high loads remains costly. The backend Flask APIs need to efficiently manage multiple concurrent requests. Optimizing AI inference times and caching frequent queries can reduce computational overhead. During peak usage, server resources may be heavily utilized, affecting response times. Cloud-based deployment helps, but scaling infrastructure for high loads remains costly. The backend Flask APIs need to efficiently manage multiple concurrent requests.

13.3. Known Issues – Requires Stable Internet Connectivity:

Citizen AI relies on stable internet connectivity for smooth operation. Since the platform interacts with cloud-hosted AI models and backend servers, any network disruption can interrupt service. Citizens may experience delays or failures in query responses during low connectivity periods. Real-time features, such as dynamic dashboards and instant responses, are particularly sensitive to network stability. Mobile users in remote areas may face challenges accessing the platform consistently. The dependency on internet speed also affects the quality of AI model inference. Administrators need to ensure redundancy and robust network architecture. Citizens may experience delays or failures in query responses during low connectivity periods. Real-time features, such as dynamic dashboards and instant responses, are particularly sensitive to network stability.