Sustainable Smart City Assistant Using IBM Granite LLM

1.Introduction

- Project Title: Sustainable Smart City Assistant Using IBM Granitee LLM
- Team Members: 1) Vadranam Bindu
  Role: Team Lead & Full Stack Integrator
  Key Area: Overall design, APIs, integration, deployment.
  2) Siddineni Jogendra Venkata Sai Kumar
  Role: Backend & API Developer
  Key Area: FastAPI routers, Granite LLM, Pinecone.
  3) Surepalli Sampath Kumar
  Role: ML & Data Module Lead
  Key Area: KPI forecasting, anomaly detection, data processing
  4) Sai Sri Chaitra Karavadi
  Role: UI/UX & Report Specialist
  Key Area: Streamlit UI, AI reports, frontend polish

2. Project Overview

- Purpose: The purpose of the *Sustainable Smart City Assistant* is to provide an AI-powered platform that enhances urban governance, environmental awareness, and citizen engagement. By leveraging IBM Watsonx Granite LLM, real-time data analytics, and machine learning, the system enables city administrators and residents to make informed, sustainable decisions. It offers tools for KPI forecasting, anomaly detection, policy summarization, semantic search, and eco-advice, all through an interactive dashboard interface.

- Features:

1. **AI Chat Assistant** – Users can ask sustainability-related questions and receive intelligent responses from IBM Watsonx Granite LLM.
2. **Policy Summarization** – Automatically converts complex policy documents into easy-to-understand summaries.
3. **Citizen Feedback System** – Allows residents to report local issues with category tagging and backend storage.
4. **KPI Forecasting** – Uses machine learning to predict future trends in water, energy, and air quality metrics.
5. **Anomaly Detection** – Flags unusual spikes in uploaded KPI datasets to alert city officials.
6. **Eco Tips Generator** – Generates actionable, AI-powered tips on living sustainably based on user input topics.
7. **Semantic Policy Search** – Uses Pinecone to perform intelligent search across embedded policy documents.

8. **Sustainability Report Generator** – Produces detailed AI-generated reports based on city KPI data.

3.Architecture

- Frontend: The frontend is built using **Streamlit**, a Python-based web framework that enables rapid development of interactive dashboards. It includes modular components like smart_dashboard.py, chat_assistant.py, and feedback_form.py, which communicate with the FastAPI backend via REST API calls. Streamlit-option-menu is used for sidebar navigation, and real-time responses are rendered dynamically using JSON data from the backend.
- Backend: The backend is built using **FastAPI** a high-performance Python web framework that enables modular, RESTful API development. Each feature (chat, feedback, KPI forecasting etc.) is handled by a dedicated router inside the app/api/ directory. Core services like IBM Watsonx Granite LLM integration, Pinecone vector search, and ML models are encapsulated in the services/module. Environmental configurations are managed using python-dotenv and Pydantic. The backend handles data ingestion (CSV, text), ML processing, semantic embedding, and AI inference, and returns JSON responses to the frontend.
- Database: This project uses **Pinecone** as a **vector database** to store and search semantically embedded policy documents. Each document is broken into chunks and converted into 384-dimensional vectors using the sentence-transformers model. These vectors, along with associated metadata (e.g., document name, chunk ID), are stored in the smartcity-policies index.
- Additionally, **citizen feedback data** (name, category, message, timestamp) is temporarily stored in structured JSON format and can be extended to a traditional database (e.g., PostgreSQL or MongoDB) for persistent storage.

4.Setup Instructions

- Prerequisites: This project requires Python 3.8+, with FastAPI for the backend and Streamlit for the frontend. Uvicorn is used to run the API server, and python-dotenv handles environment variables. It integrates IBM Watsonx Granite LLM for AI responses, Pinecone for semantic search, and sentence-transformers for embeddings. Machine learning features use scikit-learn and pandas, while matplotlib handles visualizations. Optional tools include markdown and fpdf for report export.
- Installation: Clone the Project Repository git clone https://github.com/your-username/smart-city-assistant.git

cd smart-city-assistant

- **Create a Virtual Environment (Optional but Recommended)**
- python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

- **Install Required Dependencies**
- pip install -r requirements.txt

- **Set Up Environment Variables**

- Create a .env file in the root directory.
- Copy the following template into it and fill in your actual keys:
- WATSONX_API_KEY=your_ibm_api_key

WATSONX_PROJECT_ID=your_project_id
WATSONX_URL=https://your-region.ml.cloud.ibm.com
WATSONX_MODEL_ID=ibm/granite-13b-instruct-v2
PINECONE_API_KEY=your_pinecone_key
PINECONE_ENV=your_pinecone_environment
INDEX_NAME=smartcity-policies

- **Run the Backend (FastAPI)**
- uvicorn app.main:app --reload

- **Run the Frontend (Streamlit Dashboard)**
- streamlit run frontend/smart_dashboard.py

5.Folder Structure:

- Client: The frontend of the project is developed using Streamlit and is structured in a modular way. The main file, smart_dashboard.py, manages the overall layout and sidebar navigation, dynamically loading different sections like chat, feedback, eco tips, and KPI analysis. Each feature is implemented in its own component file, such as chat_assistant.py, feedback_form.py, and eco_tips.py, ensuring a clean and scalable design. These components interact with the FastAPI backend using HTTP requests and render real-time responses in the UI. Visual enhancements like gradient backgrounds, icon-rich sidebars, and rounded cards are added using streamlit-option-menu and custom styling to deliver a user-friendly dashboard experience.
- Server: The backend is built using FastAPI and follows a modular structure for scalability and clarity. All API routes are organized under the app/api/ directory, with separate files for chat, feedback, policy summarization, eco tips, KPI uploads, and anomaly detection. Core services like IBM Watsonx LLM integration, Pinecone search, and ML forecasting are implemented in the app/services/ folder. Environment variables are securely managed using .env and Pydantic in config.py. The main entry point, app/main.py, initializes the FastAPI app and includes all routers for seamless backend interaction.

6.Running the Application

- Commands to start the frontend and backend servers locally.
- Frontend: **Navigate to your project directory (if not already):**
- cd smart-city-assistant
- **Run the Streamlit dashboard:**
- streamlit run frontend/smart_dashboard.py
- Backend: **Navigate to your project directory (if not already):**
- cd smart-city-assistant
- **Run the FastAPI server using Uvicorn:**
- uvicorn app.main:app --reload

7. API Documentation

- **Backend API Endpoints Documentation (Summary)**
- POST /chat/ask

Accepts a prompt and returns an AI-generated response using IBM Watsonx Granite LLM.

- POST /policy/summarize

Accepts a policy document (text) and returns a summarized version.

- GET /get-eco-tips?topic=solar

Returns AI-generated eco tips based on the provided topic keyword.

- POST /submit-feedback

Receives user feedback with name, category, and message, then stores it.

- POST /upload-kpi

Uploads a .csv file containing city KPIs (e.g., water, energy) for forecasting.

- POST /detect-anomalies

Accepts KPI data and returns flagged anomalies in the dataset.

- POST /generate-report

Generates a sustainability report from given KPI inputs using Granite LLM.

- POST /upload-doc

Uploads and embeds a .txt document into Pinecone for semantic search.

- GET /search-docs?query=urban planning

Searches the Pinecone vector index for documents matching the query.

- **Backend API Endpoints Overview**
- **1. /chat/ask**

Method: POST

Body: { "prompt": "How can cities reduce carbon emissions?" }

- **Response:**
- { "response": "Cities can reduce emissions by adopting green rooftops, promoting EVs..." }
- **2. /policy/summarize**

Method: POST

Body: { "text": "Long policy document text here..." }

- **Response:**
- { "summary": "This policy focuses on urban mobility and clean energy..." }
- **3. /get-eco-tips?topic=water**

Method: GET

Response:

- { "tip": "Fix leaking taps to conserve thousands of liters annually." }
- **4. /submit-feedback**

Method: POST

Body:

```
{
  "name": "XYZ",
  "category": "Water",
  "message": "Burst pipe near street 9."
}
```

- **Response:** { "status": "Feedback submitted successfully." }

- **5. /upload-kpi**

Method: POST
FormData: File upload (.csv)
Response: { "forecast": { "2025": 1832.4, "2026": 1920.5 } }

- **6. /detect-anomalies**

Method: POST
FormData: File upload (.csv)
Response:

- { "anomalies": ["March 2024: spike in water usage"] }
- **7. /generate-report**

Method: POST
Body: { "city": "Xcity", "kpi_data": { "water": [...], "energy": [...] } }

- **Response:** { "report": "Xcity has shown significant progress in water conservation..." }
- **8. /upload-doc**

Method: POST
FormData: File upload (.txt)
Response:

- { "status": "Document embedded successfully." }
- **9. /search-docs?query=urban planning**

Method: GET
Response:

- { "results": ["Policy 1: Promote public transport...", "Policy 2: Increase green zones..."] }

8. Authentication

- **Authentication and Authorization in the Project**
- At present, the project focuses on open access for development and demonstration purposes—meaning endpoints can be accessed without login or role restrictions.
- To **secure the system**, especially for admin-only features like uploading KPIs or generating reports, you can implement:
- **Authentication (Verifying Identity):**
- Use **API key-based authentication** or **JWT (JSON Web Token)** to validate users:
- **API Key Approach** (for internal/admin use):
    - Require a secret key in headers for protected endpoints.
    - Example:
- Authorization: Bearer YOUR_SECRET_KEY
- **JWT Token Approach** (for multi-user systems):
    - User logs in → backend issues a signed JWT token.
    - Token must be passed with each request to access protected routes.
    - Use FastAPI's built-in OAuth2/JWT support.
- **Authorization (Controlling Access):**
- Based on roles (e.g., **admin**, **citizen**), you can control access:
- Citizens: allowed to submit feedback, get eco tips, ask chat questions.
- Admins: allowed to upload KPI data, detect anomalies, generate reports.
- You'd define role checks in your FastAPI endpoints using dependencies.
- **Authentication & Authorization**

- This project currently uses **open access** without user login, tokens, or sessions. No authentication tokens (like JWT) or session-based mechanisms are implemented. However, for future security, **JWT (JSON Web Tokens)** can be added to protect sensitive endpoints like /upload-kpi or /generate-report, and roles like **admin** or **citizen** can be used to manage access. FastAPI provides built-in support to handle these securely.

9.User Interface

- Screenshots Showcasing different UI features.
- https://github.com/user-attachments/assets/300cd126-6392-4299-9418-da0826aed03f
- https://github.com/user-attachments/assets/2f3931a1-eb44-4d9a-8834-f06fcbeb491a
- https://github.com/user-attachments/assets/c78f7b80-e7bb-4b8d-8620-ac8f26c9c5ee
- https://github.com/user-attachments/assets/7c057e8a-eb0f-4e65-a348-33ff5cd1cbe2

10.Testing

- **Testing Strategy and Tools**
- The project uses a **manual and modular testing strategy** to validate individual components during development. Each FastAPI route is tested using the **built-in Swagger UI** (http://localhost:8000/docs) to ensure proper request-response behavior, correct payload formats, and error handling.
- For frontend testing, the **Streamlit dashboard** is run locally, and each UI module (chat, eco tips, KPI upload, etc.) is tested by simulating user inputs and verifying real-time responses from the backend.
- CSV uploads for forecasting and anomaly detection are tested with sample datasets, while document embedding and search features are validated using dummy .txt files and Pinecone logs.
- Though automated testing frameworks like **Pytest** are not integrated, the modular code structure supports easy extension to unit and integration testing in the future.

11.Screenshots or demo

- https://github.com/user-attachments/assets/a25ada7b-0b4d-4dbf-b20f-f1b76e7ec2f1
- https://github.com/user-attachments/assets/f152ddd5-db11-4070-bbf0-cced273063c6
- https://github.com/user-attachments/assets/eccbfd21-2479-4881-b4a7-c3ad09b802ff
- https://github.com/user-attachments/assets/f49197e9-5441-4251-a754-51e4cc40efce

12. Known Issues

- **Known Bugs / Issues**
- **No authentication**: Currently, all endpoints are public. Sensitive actions like uploading KPIs or generating reports are not restricted.
- **File format sensitivity**: Uploading malformed or non-standard CSV files may cause forecasting or anomaly modules to fail.
- **No persistent database**: Feedback and logs are not stored in a permanent database, so data may be lost after restart.
- **Limited error handling**: Some API routes return generic errors instead of descriptive messages, which may affect debugging.
- **No multi-user support**: The system doesn't support sessions or role-based access yet.

13. Future Enhancements

- **Future Improvements**

- Add user authentication with role-based access, integrate a persistent database for storing feedback and files, and enhance the dashboard with analytics and mobile responsiveness. Future updates could include multilingual AI responses, real-time IoT data sync, secure JWT-based API protection, and advanced export options like Excel or CSV reports.