You're on the right track! Building a "Citizen AI" application with the ibm-granite/granite-3.3-2b-instruct model is an excellent idea, and choosing between FastAPI for a full web application or Gradio for a quick Colab demo covers your needs well.

Let's refine the plan and provide the code examples for both deployment scenarios, incorporating your desired functionalities.

Key Considerations Before We Start:

- **API Key Security is Paramount:** As you rightly mentioned, *never share your API key*. We'll use environment variables for this.
- **Model Size and Resources:** The ibm-granite/granite-3.3-2b-instruct model is a 2-billion parameter model.
    - **Local Download/Direct Loading (FastAPI in VS Code):** This requires significant RAM (usually >8GB, potentially 16GB or more for smooth operation) and storage (several GBs for the model weights). If you don't meet these specs, you'll need to rely on the Hugging Face Inference API.
    - **Google Colab (Gradio):** Using a T4 GPU in Colab is highly recommended for running this model directly, as it provides the necessary computational resources.
- **Graphing Library:** matplotlib is a standard and excellent choice in Python for generating static charts that can be easily embedded into web pages (as Base64 images). Plotly or Bokeh are alternatives for interactive charts but add more complexity. We'll stick with matplotlib for simplicity and direct embedding.
- **Image Generation:** For "Images for Citizen AI," if you want *dynamically generated* images (e.g., based on the AI's response), you'd integrate another generative AI model (like Stable Diffusion, DALL-E, etc.). For simplicity in these examples, we'll use a placeholder image or a simple text-based representation.

Scenario 1: Full Web Application with FastAPI (HTML & CSS) in VS Code

This approach provides maximum control over the UI/UX.

1. Project Setup and Directory Structure:

Create a new folder named CitizenAI_FastAPI and inside it:

```
CitizenAI_FastAPI/
├── main.py
```

```
├── requirements.txt
├── .env
├── templates/
│   └── index.html
└── static/
    ├── css/
    │   └── style.css
    └── img/
        └── citizen_ai_logo.png  (You'll need to create or download a placeholder image)
```

2. requirements.txt

```
fastapi
uvicorn[standard]
python-dotenv
transformers
torch # Or tensorflow-cpu/tensorflow, depending on your setup. For GPU, ensure CUDA and PyTorch/TF-GPU are installed separately.
matplotlib
Pillow # For basic image manipulation if needed
requests # For Hugging Face Inference API fallback
```

3. .env (Crucial for API Key Security)

Create this file in the CitizenAI_FastAPI directory and add your Hugging Face API token:

```
HUGGING_FACE_API_KEY="hf_YOUR_ACTUAL_HUGGING_FACE_API_KEY"
```

4. main.py (FastAPI Application Logic)

Python

```
from fastapi import FastAPI, Request, Form
```

```
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from dotenv import load_dotenv
import os
import matplotlib.pyplot as plt
import io
import base64
from transformers import pipeline
import requests # For Hugging Face Inference API fallback

# Load environment variables from .env file
load_dotenv()
HF_API_KEY = os.getenv("HUGGING_FACE_API_KEY")

if not HF_API_KEY:
    raise ValueError("HUGGING_FACE_API_KEY not found in environment variables. "
                     "Please set it in your .env file or as an environment variable.")

app = FastAPI(title="Citizen AI Application")

# Mount static files (CSS, images)
app.mount("/static", StaticFiles(directory="static"), name="static")

# Configure Jinja2Templates for HTML rendering
templates = Jinja2Templates(directory="templates")

# Initialize Hugging Face pipeline for the model
# IMPORTANT: Direct model loading requires significant RAM and potentially a GPU.
# If you face memory errors, comment this out and rely solely on the Inference API fallback.
llm_pipeline = None
try:
    # Try to load the model locally. Use device="cuda" if you have an NVIDIA GPU and CUDA installed.
    # Otherwise, it will default to CPU, which can be very slow for large models.
    llm_pipeline = pipeline(
        "text-generation",
        model="ibm-granite/granite-3.3-2b-instruct",
        token=HF_API_KEY, # Pass API key for private models or increased rate limits
        torch_dtype="auto", # Lets transformers decide optimal dtype (e.g., float16 if available)
        device="cuda" if os.getenv("USE_CUDA", "false").lower() == "true" and \
                         (os.getenv("CUDA_VISIBLE_DEVICES") or torch.cuda.is_available()) else "cpu"
    )
    print("Model loaded successfully for direct inference.")
```

```
except Exception as e:
    print(f"Could not load model directly ({e}). Falling back to Hugging Face Inference API.")


async def generate_text_with_granite(prompt: str) -> str:
    """Generates text using the IBM Granite model, preferring local pipeline or falling back to HF Inference API."""
    if llm_pipeline:
        try:
            # For direct pipeline, you might need to adjust generation parameters
            response = llm_pipeline(prompt, max_new_tokens=250, num_return_sequences=1,
                                    do_sample=True, temperature=0.7, top_p=0.9)
            return response[0]['generated_text']
        except Exception as e:
            print(f"Error during direct model inference: {e}. Attempting Inference API.")
            # Fallback to Inference API even if direct load failed during inference
            pass # Continue to the Inference API logic below

    # Fallback to Hugging Face Inference API
    HF_API_URL = "https://api-inference.huggingface.co/models/ibm-granite/granite-3.3-2b-instruct"
    headers = {"Authorization": f"Bearer {HF_API_KEY}", "Content-Type": "application/json"}
    payload = {"inputs": prompt, "parameters": {"max_new_tokens": 250, "temperature": 0.7, "top_p": 0.9}}

    try:
        response = requests.post(HF_API_URL, headers=headers, json=payload)
        response.raise_for_status() # Raises HTTPError for bad responses (4xx or 5xx)
        result = response.json()
        if isinstance(result, list) and result and 'generated_text' in result[0]:
            return result[0]['generated_text']
        else:
            return f"Error: Unexpected response format from Inference API: {result}"
    except requests.exceptions.RequestException as req_e:
        return f"Error communicating with Hugging Face Inference API: {req_e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"


def generate_bar_chart(data: dict, title: str, xlabel: str, ylabel: str) -> str:
    """Generates a bar chart and returns it as a Base64 encoded PNG string."""
    labels = list(data.keys())
    values = list(data.values())
```

```
    plt.figure(figsize=(8, 5))
    plt.bar(labels, values, color=['#007bff', '#28a745', '#ffc107', '#dc3545']) # Bootstrap-like colors
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()

    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    plt.close() # Close the plot to free memory
    return base64.b64encode(buf.getvalue()).decode('utf-8')

def generate_pie_chart(data: dict, title: str) -> str:
    """Generates a pie chart and returns it as a Base64 encoded PNG string."""
    labels = list(data.keys())
    values = list(data.values())
    colors = ['#fd7e14', '#6610f2', '#20c997'] # Another set of colors for variety

    plt.figure(figsize=(7, 7))
    plt.pie(values, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90,
explode=[0.05]*len(labels))
    plt.title(title)
    plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
    plt.tight_layout()

    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    plt.close() # Close the plot to free memory
    return base64.b64encode(buf.getvalue()).decode('utf-8')

@app.get("/", response_class=HTMLResponse)
async def read_root(request: Request):
    """Renders the initial Citizen AI application page."""
    # Placeholder for Citizen AI image
    citizen_ai_image_path = "/static/img/citizen_ai_logo.png"
    return templates.TemplateResponse("index.html", {
        "request": request,
        "app_name": "Citizen AI",
        "citizen_ai_image": citizen_ai_image_path,
        "ai_response": None, # Initially no AI response
        "graph_age": None,
        "graph_gender": None
```

```
    })

@app.post("/citizen_ai", response_class=HTMLResponse)
async def citizen_ai_response(request: Request, user_prompt: str = Form(...)):
    """Processes the user prompt, generates AI response and graphs, then renders the page."""
    generated_text = await generate_text_with_granite(user_prompt)

    # 1. Graphs for citizens as per their ages (Example Data)
    # In a real app, this data would come from a database or a data source
    age_data = {
        "0-18 years": 15000,
        "19-35 years": 25000,
        "36-60 years": 30000,
        "61+ years": 10000
    }
    graph_age_base64 = generate_bar_chart(age_data, "Citizen Distribution by Age Group", "Age Group", "Number of Citizens")

    # 2. Women's and men's and children's (Example Data)
    gender_data = {
        "Women": 40000,
        "Men": 35000,
        "Children": 20000
    }
    graph_gender_base64 = generate_pie_chart(gender_data, "Population Breakdown (Estimated)", "Population Segment")

    # 3. Images for Citizen AI (Using the placeholder logo for now)
    citizen_ai_image_path = "/static/img/citizen_ai_logo.png"
    # If you had a dynamic image generation model, you'd call it here
    # E.g., `generated_image_base64 = await generate_image_with_ai(user_prompt)`

    return templates.TemplateResponse("index.html", {
        "request": request,
        "app_name": "Citizen AI",
        "user_prompt": user_prompt,
        "ai_response": generated_text,
        "graph_age": graph_age_base64,
        "graph_gender": graph_gender_base64,
        "citizen_ai_image": citizen_ai_image_path # This is your static logo
        # "dynamic_ai_image": generated_image_base64 # If you implement dynamic image gen
    })
```

```
# Run the application using Uvicorn
if __name__ == "__main__":
    import uvicorn
    # Set USE_CUDA environment variable to "true" if you want to force GPU usage
    # (assuming PyTorch and CUDA are correctly installed and your GPU is compatible)
    # Example: os.environ["USE_CUDA"] = "true"
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Explanation of main.py:

- **API Key Handling:** python-dotenv loads HUGGING_FACE_API_KEY from .env.
- **Model Loading:** Attempts to load ibm-granite/granite-3.3-2b-instruct using transformers.pipeline. It includes a fallback to the Hugging Face Inference API if local loading fails.
    - **device="cuda":** This is critical for GPU usage. If torch.cuda.is_available() is True and USE_CUDA is "true", it will try to use the GPU. Otherwise, it defaults to "cpu".
- **generate_text_with_granite:** Asynchronously handles text generation, either via the loaded pipeline or by making a requests call to the Hugging Face Inference API. It includes basic error handling.
- **generate_bar_chart and generate_pie_chart:** These functions use matplotlib to create the desired graphs. They save the plots to an in-memory buffer (io.BytesIO) and then encode them as Base64 strings. This allows you to directly embed the images into your HTML without saving physical files.
- **FastAPI Endpoints:**
    - / (GET): Displays the initial form.
    - /citizen_ai (POST): Processes the user's prompt, calls the LLM, generates the example graph data, and then renders the index.html template with all the generated content.
- **uvicorn.run:** Starts the FastAPI development server.

5. templates/index.html

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ app_name }}</title>
    <link rel="stylesheet" href="/static/css/style.css">
```

```
</head>
<body>
    <header>
        <div class="header-content">
            <img src="{{ citizen_ai_image }}" alt="Citizen AI Logo" class="app-logo">
            <h1>{{ app_name }}</h1>
        </div>
    </header>

    <main>
        <section class="prompt-section">
            <h2>Engage with Citizen AI</h2>
            <form action="/citizen_ai" method="post">
                <textarea name="user_prompt" placeholder="Ask Citizen AI a question or provide a
scenario..." rows="8" required>{{ user_prompt if user_prompt else '' }}</textarea>
                <button type="submit">Get Citizen AI Insights</button>
            </form>
        </section>

        {% if ai_response %}
        <section class="ai-response-section">
            <h2>AI Response:</h2>
            <div class="ai-output">
                <p>{{ ai_response }}</p>
            </div>
        </section>

        <section class="data-insights">
            <h2>Citizen Data Insights</h2>
            <div class="graph-container">
                {% if graph_age %}
                <div class="graph-card">
                    <h3>Population by Age Group</h3>
                    <img src="{{ graph_age }}" alt="Citizens by Age Graph">
                </div>
                {% endif %}

                {% if graph_gender %}
                <div class="graph-card">
                    <h3>Population Breakdown</h3>
                    <img src="{{ graph_gender }}" alt="Gender and Children Graph">
                </div>
                {% endif %}
```

```
        </div>
        </section>
    {% endif %}
  </main>

  <footer>
    <p>© 2025 Citizen AI. All rights reserved.</p>
  </footer>
</body>
</html>
```

Explanation of index.html:

- **Jinja2 Templating:** {{ variable }} syntax is used to dynamically inject data from the FastAPI backend.
- **Form:** A simple textarea for user input and a submit button.
- **Conditional Display:** {% if ai_response %} ensures that the AI response and graphs only appear after a submission.
- **Base64 Images:** <img> tags with src="{{ graph_data }}" directly embed the generated charts into the HTML. This is a common and efficient way to display dynamically created images without saving them to disk.
- **Placeholder Image:** The citizen_ai_image variable points to your static logo.

6. static/css/style.css (Basic Styling)

CSS

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  margin: 0;
  padding: 0;
  background-color: #e9eff5;
  color: #333;
  line-height: 1.6;
}

header {
  background-color: #0056b3; /* Darker blue */
  color: white;
  padding: 1.5rem 0;
  text-align: center;
```

```
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.2);
}

.header-content {
    display: flex;
    align-items: center;
    gap: 15px;
}

.app-logo {
    width: 60px;
    height: 60px;
    border-radius: 50%;
    object-fit: cover;
    border: 3px solid #ffffff;
}

h1 {
    margin: 0;
    font-size: 2.5rem;
    letter-spacing: 1px;
}

main {
    max-width: 1000px;
    margin: 30px auto;
    padding: 25px;
    background-color: white;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
    border-radius: 10px;
}

section {
    margin-bottom: 40px;
    padding: 25px;
    border: 1px solid #dcdcdc;
    border-radius: 8px;
    background-color: #fcfcfc;
}
```

```css
h2 {
    color: #0056b3;
    text-align: center;
    margin-bottom: 25px;
    border-bottom: 2px solid #e0e0e0;
    padding-bottom: 10px;
}

textarea {
    width: calc(100% - 20px); /* Adjust for padding */
    padding: 10px;
    margin-bottom: 20px;
    border: 1px solid #ccc;
    border-radius: 5px;
    box-sizing: border-box;
    font-size: 1rem;
    resize: vertical;
    min-height: 120px;
}

button {
    display: block;
    width: 100%;
    padding: 12px 20px;
    background-color: #28a745;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 1.1rem;
    transition: background-color 0.3s ease;
}

button:hover {
    background-color: #218838;
}

.ai-output {
    background-color: #e2f0fb; /* Light blue */
    padding: 20px;
    border-left: 5px solid #007bff;
    border-radius: 5px;
```

```
    white-space: pre-wrap; /* Preserves whitespace and line breaks from model output */
    font-size: 1rem;
    line-height: 1.8;
}

.data-insights h2 {
    color: #0056b3;
}

.graph-container {
    display: flex;
    flex-wrap: wrap;
    justify-content: space-around;
    gap: 30px;
    margin-top: 30px;
}

.graph-card {
    flex: 1 1 calc(50% - 30px); /* Two columns with gap */
    min-width: 380px; /* Ensures graphs don't become too small */
    text-align: center;
    background-color: #f8f8f8;
    border: 1px solid #e0e0e0;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 2px 8px rgba(0, 0, 0, 0.08);
}

.graph-card h3 {
    color: #555;
    margin-top: 0;
    margin-bottom: 15px;
}

.graph-card img {
    max-width: 100%;
    height: auto;
    border-radius: 5px;
    border: 1px solid #ccc;
}

.dynamic-image-section {
    text-align: center;
```

```
    margin-top: 40px;
    padding: 20px;
    background-color: #f0f8ff;
    border: 1px dashed #a0d0ff;
    border-radius: 8px;
}

.dynamic-ai-image {
    max-width: 100%;
    height: auto;
    border-radius: 8px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.15);
}

footer {
    text-align: center;
    padding: 1.5rem 0;
    background-color: #343a40; /* Dark gray */
    color: white;
    margin-top: 40px;
    box-shadow: 0 -2px 5px rgba(0, 0, 0, 0.2);
}
```

7. static/img/citizen_ai_logo.png (Placeholder Image)

Create a simple image file (e.g., a circle with "CAI" or a symbolic icon) and save it as citizen_ai_logo.png in the static/img/ directory.

How to Run (VS Code / Local Machine):

1. **Navigate to Project Directory:** Open your terminal or VS Code and navigate to the CitizenAI_FastAPI folder.
2. **Install Dependencies:**
   Bash
   `pip install -r requirements.txt`
   Self-correction: If torch or tensorflow installation is problematic, install torch (for GPU if you have one, else CPU version) or tensorflow directly:
   - For CPU: pip install torch --index-url https://download.pytorch.org/whl/cpu
   - For GPU (NVIDIA CUDA required): pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118 (for CUDA 11.8) - adjust CUDA version as needed.
3. **Set API Key:** Ensure your .env file is correctly populated with your Hugging Face API key.
4. **Run FastAPI Application:**
   Bash

uvicorn main:app --reload
The --reload flag is great for development as it automatically restarts the server when you save changes to your code.

5. **Access in Browser:** Open your web browser and go to http://127.0.0.1:8000.

Scenario 2: Quick Demo with Gradio in Google Colab

This is perfect for rapid prototyping and sharing interactive applications, especially when dealing with larger models that benefit from Colab's T4 GPU.

Steps in Google Colab:

1. **Open a New Colab Notebook:** Go to Google Colab and create a new notebook.
2. **Change Runtime Type:** Click on Runtime -> Change runtime type. Select T4 GPU as the hardware accelerator and click Save. This is crucial for performance.
3. **Install Libraries:** Run the following in a Colab cell:
   Python
   ```
   !pip install transformers accelerate gradio matplotlib Pillow requests
   !pip install torch # Colab usually has torch pre-installed, but good to ensure
   !pip install google-colab-userdata # For secure API key handling
   ```

4. **Secure API Key (Using Colab Secrets):**
   - On the left sidebar of your Colab notebook, click the **key icon** (Secrets).
   - Click Add a new secret.
   - For "Name", type HUGGING_FACE_API_KEY.
   - For "Value", paste your actual Hugging Face API token (hf_YOUR_ACTUAL_HUGGING_FACE_API_KEY).
   - Make sure "Notebook access" is toggled ON for your current notebook.
   - Now you can access it in your code securely: from google.colab import userdata; HF_API_KEY = userdata.get('HUGGING_FACE_API_KEY')
5. **Gradio Application Code:** Run the following code in a new Colab cell:
   Python
   ```
   import gradio as gr
   import matplotlib.pyplot as plt
   import io
   import base64
   from transformers import pipeline
   import os
   from google.colab import userdata # For accessing Colab secrets
   import requests # For Hugging Face Inference API fallback

   # Load API Key from Colab Secrets
   HF_API_KEY = userdata.get('HUGGING_FACE_API_KEY')
   ```

```
if not HF_API_KEY:
    raise ValueError("HUGGING_FACE_API_KEY not found in Colab Secrets. "
                     "Please add it via the 'Secrets' tab on the left sidebar.")

# Initialize Hugging Face pipeline for the model
# Colab's T4 GPU is typically at device=0
llm_pipeline = None
try:
    llm_pipeline = pipeline(
        "text-generation",
        model="ibm-granite/granite-3.3-2b-instruct",
        token=HF_API_KEY,
        torch_dtype="auto",
        device=0 # Use GPU (typically device 0 in Colab)
    )
    print("Model loaded successfully for direct inference on GPU.")
except Exception as e:
    print(f"Could not load model directly on GPU ({e}). Falling back to Hugging Face Inference API.")
    # If direct loading failed, llm_pipeline remains None, triggering Inference API fallback.

def generate_text_with_granite(prompt: str) -> str:
    """Generates text using the IBM Granite model, preferring local pipeline or falling back to HF Inference API."""
    if llm_pipeline:
        try:
            response = llm_pipeline(prompt, max_new_tokens=250, num_return_sequences=1,
                                    do_sample=True, temperature=0.7, top_p=0.9)
            return response[0]['generated_text']
        except Exception as e:
            print(f"Error during direct model inference: {e}. Attempting Inference API.")
            pass # Fallback to Inference API even if direct load failed during inference

    # Fallback to Hugging Face Inference API
    HF_API_URL = "https://api-inference.huggingface.co/models/ibm-granite/granite-3.3-2b-instruct"
    headers = {"Authorization": f"Bearer {HF_API_KEY}", "Content-Type": "application/json"}
    payload = {"inputs": prompt, "parameters": {"max_new_tokens": 250, "temperature": 0.7, "top_p": 0.9}}

    try:
        response = requests.post(HF_API_URL, headers=headers, json=payload)
        response.raise_for_status() # Raises HTTPError for bad responses (4xx or 5xx)
```

```
        result = response.json()
        if isinstance(result, list) and result and 'generated_text' in result[0]:
            return result[0]['generated_text']
        else:
            return f"Error: Unexpected response format from Inference API: {result}"
    except requests.exceptions.RequestException as req_e:
        return f"Error communicating with Hugging Face Inference API: {req_e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"


def generate_bar_chart(data: dict, title: str, xlabel: str, ylabel: str) -> str:
    """Generates a bar chart and returns it as a Base64 encoded PNG string."""
    labels = list(data.keys())
    values = list(data.values())

    plt.figure(figsize=(8, 5))
    plt.bar(labels, values, color=['#007bff', '#28a745', '#ffc107', '#dc3545'])
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()

    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    plt.close()
    return base64.b64encode(buf.getvalue()).decode('utf-8')

def generate_pie_chart(data: dict, title: str) -> str:
    """Generates a pie chart and returns it as a Base64 encoded PNG string."""
    labels = list(data.keys())
    values = list(data.values())
    colors = ['#fd7e14', '#6610f2', '#20c997']

    plt.figure(figsize=(7, 7))
    plt.pie(values, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90,
explode=[0.05]*len(labels))
    plt.title(title)
    plt.axis('equal')
    plt.tight_layout()

    buf = io.BytesIO()
```

```
    plt.savefig(buf, format='png')
    plt.close()
    return base64.b64encode(buf.getvalue()).decode('utf-8')

def citizen_ai_interface(user_prompt: str):
    """Main function for Gradio interface that combines AI response and charts."""
    # Generate text response
    ai_response = generate_text_with_granite(user_prompt)

    # 1. Graphs for citizens as per their ages (Example Data)
    age_data = {
        "0-18 years": 15000,
        "19-35 years": 25000,
        "36-60 years": 30000,
        "61+ years": 10000
    }
    graph_age_base64 = generate_bar_chart(age_data, "Citizen Distribution by Age Group",
"Age Group", "Number of Citizens")

    # 2. Women's and men's and children's (Example Data)
    gender_data = {
        "Women": 40000,
        "Men": 35000,
        "Children": 20000
    }
    graph_gender_base64 = generate_pie_chart(gender_data, "Population Breakdown
(Estimated)", "Population Segment")

    # 3. Images for Citizen AI (Placeholder image from an external URL for Gradio)
    citizen_ai_image_html = '<img
src="https://via.placeholder.com/100x100?text=Citizen+AI+Logo" alt="Citizen AI Logo"
style="border-radius: 50%; display: block; margin: 10px auto; border: 3px solid #0056b3;">'

    # Construct the HTML output for Gradio
    html_output = f"""
    <div style="background-color: #f7f9fc; padding: 25px; border-radius: 12px; box-shadow: 0 4px
15px rgba(0,0,0,0.1);">
        <h2 style="color: #0056b3; text-align: center; margin-bottom: 20px;">Citizen AI Insights</h2>
        <div style="text-align: center; margin-bottom: 25px;">
            {citizen_ai_image_html}
            <p style="font-size: 1.1em; color: #555;">Your intelligent assistant for civic data.</p>
        </div>

        <div style="margin-bottom: 30px; padding: 20px; background-color: #e2f0fb; border-left:
```

```
5px solid #007bff; border-radius: 8px;">
            <h3 style="color: #333; margin-top: 0;">AI Response:</h3>
            <p style="white-space: pre-wrap; font-size: 1rem; line-height: 1.7;">{ai_response}</p>
        </div>

        <div style="margin-top: 30px; padding: 20px; background-color: #fcfcfc; border: 1px solid
#dcdcdc; border-radius: 8px;">
            <h3 style="color: #0056b3; text-align: center; margin-bottom: 25px;">Citizen Data
Visualizations</h3>
            <div style="display: flex; flex-wrap: wrap; justify-content: space-around; gap: 25px;">
                <div style="flex: 1 1 calc(50% - 25px); min-width: 350px; text-align: center;
background-color: #ffffff; border: 1px solid #e0e0e0; padding: 18px; border-radius: 10px;
box-shadow: 0 2px 10px rgba(0,0,0,0.08);">
                    <h4>Population Distribution by Age</h4>
                    <img src="{graph_age_base64}" alt="Citizens by Age Graph"
style="max-width: 100%; height: auto; border-radius: 5px; border: 1px solid #ccc;">
                </div>
                <div style="flex: 1 1 calc(50% - 25px); min-width: 350px; text-align: center;
background-color: #ffffff; border: 1px solid #e0e0e0; padding: 18px; border-radius: 10px;
box-shadow: 0 2px 10px rgba(0,0,0,0.08);">
                    <h4>Population Breakdown</h4>
                    <img src="{graph_gender_base64}" alt="Gender and Children
Graph" style="max-width: 100%; height: auto; border-radius: 5px; border: 1px solid #ccc;">
                </div>
            </div>
        </div>
    </div>
    """
    return html_output

# Gradio Interface Setup
iface = gr.Interface(
    fn=citizen_ai_interface,
    inputs=gr.Textbox(lines=5, label="Enter your prompt for Citizen AI:", placeholder="E.g.,
'What are the key demographic trends in urban areas?'"),
    outputs=gr.HTML(label="Citizen AI Output"), # Gradio will render the HTML string
    title="Citizen AI Interactive Demo",
    description="Interact with the IBM Granite model to get insights and visualize citizen data.",
    live=False, # Set to True for continuous updates as you type, but can be resource-intensive
    allow_flagging="auto", # Allows users to flag inputs for review
)

iface.launch(share=True) # share=True generates a public link for easy sharing
```

Explanation of Gradio Code:

- **Colab-Specific Imports:** google.colab.userdata for secure API keys.
- **Model Initialization:** Similar pipeline setup as FastAPI, but device=0 explicitly targets the first GPU in Colab.
- **Graph Functions:** Identical matplotlib functions to the FastAPI example, generating Base64 images.
- **citizen_ai_interface:** This function is the core of the Gradio app. It takes the user_prompt, calls the LLM, generates graphs, and then constructs a single, well-formatted HTML string containing all the results. This HTML string is what Gradio displays in its output.
- **gr.Interface:** Creates the Gradio user interface.
  - inputs: Defines the input component (a gr.Textbox).
  - outputs: Uses gr.HTML to render the complex HTML string returned by citizen_ai_interface.
- **iface.launch(share=True):** Starts the Gradio server. share=True generates a public, temporary URL that you can share with others, allowing them to interact with your demo directly in their browser.

How to Run (Google Colab):

1. After opening your notebook and setting the T4 GPU runtime, paste the "Install Libraries" code into a cell and run it.
2. Set up your HUGGING_FACE_API_KEY in Colab Secrets as described.
3. Paste the "Gradio Application Code" into a new cell and run it.
4. Gradio will print out two URLs: a local URL and a public ngrok URL (e.g., https://xxxx.gradio.live). Click on the public URL to open your "Citizen AI" demo in a new tab.

Summary of Deployment Ways:

As you correctly identified:

- **Application (Interactive Demos): Streamlit or Gradio**
  - **Pros:** Quick setup, excellent for interactive UIs, especially for ML/AI models, easy to share (Gradio's share=True in Colab is a prime example).
  - **Cons:** Less control over highly custom front-end design, not typically built for complex multi-page web applications with full user management or deep database integration without significant custom work.
  - **When to use:** Rapid prototyping, quick demonstrations, internal tools, educational purposes.
- **Web Application (Production-Ready Apps): FastAPI or Flask (HTML & CSS)**
  - **Pros:** Full control over front-end (HTML, CSS, JavaScript), scalable, robust for production environments, flexible for integrating databases, authentication, and

other backend services. FastAPI is known for its speed and modern asynchronous capabilities.
    - **Cons:** Requires more front-end development knowledge and more setup for deployment (e.g., Docker, cloud hosting).
    - **When to use:** Building a fully-featured public-facing web application, e-commerce sites, social platforms, or any application requiring a custom, polished user experience.

These examples provide a strong starting point for your "Citizen AI" project, covering both quick interactive demos and more robust web application deployments. Remember to iteratively improve and expand upon these foundational elements!