**Medical AI Assistant – A Conversational Health Support Tool**

**Project Documentation**

**1. Introduction**

**Team Members:**

* Team Member 1: **Mythili.M**
* Team Member 2: **Mythri.P**
* Team Member 3: **Preethi.G**
* Team Member 4: **Nivetha.R**

**Background and Motivation**

Healthcare is one of the most vital aspects of human life, yet access to medical expertise is often limited due to geographical, financial, or time constraints. In today’s fast-paced world, patients frequently turn to the internet to self-diagnose or search for treatments—an approach that can be inaccurate or misleading. At the same time, healthcare professionals face overwhelming workloads and resource constraints, making it harder to provide personalized attention to every patient.

To address these gaps, our project introduces a **Medical AI Assistant**, built using **IBM Granite AI Models**, **Transformers**, **PyTorch**, and **Gradio**. The system is designed to act as a conversational companion for individuals who want quick insights about their symptoms, possible medical conditions, and generalized treatment suggestions. While it does not replace certified medical professionals, it serves as a **first-level informational tool** that empowers users to make informed decisions and encourages them to seek timely medical consultation.

**Objective of the Project**

The **primary objective** of the Medical AI Assistant is to:

1. Provide a **symptom-based analysis** to help users understand possible underlying medical conditions.
2. Suggest **generalized treatment guidelines**, including home remedies, lifestyle adjustments, and common medication categories (without prescribing specific drugs).
3. Offer an **accessible, conversational interface** through which patients can interact naturally, reducing the barrier to accessing medical information.
4. Ensure **ethical usage** by embedding disclaimers, highlighting the importance of consulting doctors, and preventing misuse of the system as a substitute for professional healthcare.
5. Demonstrate the **practical integration** of modern LLMs (Large Language Models) with user-friendly frontends for health-related applications.

**Importance of the Project**

The Medical AI Assistant represents a significant step towards the democratization of healthcare information:

* **For patients:** It provides a trusted, interactive medium to understand symptoms, reducing anxiety and promoting awareness.
* **For doctors and hospitals:** It helps filter and prepare patients before consultations, saving valuable time.
* **For researchers and students:** It acts as a case study of how AI can be applied responsibly in the medical domain.

By combining AI’s predictive power with an easy-to-use interface, the project shows how technology can augment—not replace—human healthcare professionals, bridging the gap between **medical knowledge** and **patient accessibility**.

**Scope of the Project**

The system focuses on **two primary functionalities** at this stage:

1. **Disease Prediction** – Based on user-entered symptoms, the AI generates possible conditions and offers informational guidance.
2. **Treatment Plan Generation** – Based on user-entered condition, age, gender, and medical history, the AI suggests general lifestyle modifications, home remedies, and potential treatment directions.

Future expansions could include integration with wearable devices, real-time monitoring, multilingual support, or advanced clinical decision support for healthcare professionals.

**2. Project Overview**

**Purpose**

The purpose of the **Medical AI Assistant** is to leverage artificial intelligence and natural language processing (NLP) technologies to provide **informational healthcare support**. This project aims to empower individuals with **quick, personalized, and conversational access** to medical insights while emphasizing the importance of consulting certified healthcare professionals for accurate diagnosis and treatment.

Unlike traditional search engines, where patients sift through thousands of unverified results, the Medical AI Assistant uses **IBM Granite LLMs**, a curated prompt structure, and a **Gradio-powered conversational interface** to return clear, contextual, and human-like responses.

This assistant addresses key challenges in healthcare access:

* **Accessibility** – Reaching patients in underserved or remote areas.
* **Awareness** – Helping users understand common conditions and preventive care.
* **Efficiency** – Allowing doctors and healthcare systems to handle patient queries more effectively by filtering preliminary questions.
* **Trust & Safety** – Providing disclaimers and reminders to avoid misuse while ensuring the assistant remains an informational guide, not a replacement for a doctor.

The broader vision is to **bridge the knowledge gap** between healthcare systems and patients, making medical information more **reliable, conversational, and accessible.**

**Features**

The Medical AI Assistant provides multiple functionalities that make it both useful and unique. Below are the **core features** explained in detail:

**1. Conversational Interface**

* **Key Point:** Natural language interaction.
* **Functionality:**  
  The system allows users to type their symptoms or conditions in plain language and receive structured, AI-generated insights. This makes it easy for people who may not know medical terminology to still communicate effectively.
* **Example:** A user typing *“I have a fever and body aches”* will receive a response outlining possible conditions like flu, dengue, or viral infection, along with suggestions for general care.

**2. Disease Prediction**

* **Key Point:** Symptom-based condition analysis.
* **Functionality:**  
  By analyzing input symptoms, the assistant generates a list of possible medical conditions and general recommendations. It avoids exact diagnoses but helps narrow down possibilities.
* **Example:** For *“persistent cough and shortness of breath”*, the system may suggest conditions like bronchitis, asthma, or respiratory infection.

**3. Treatment Plan Generator**

* **Key Point:** Personalized health guidance.
* **Functionality:**  
  Users provide additional details such as **age, gender, and medical history**, enabling the AI to generate tailored treatment suggestions. This includes **home remedies, lifestyle modifications, and general medication categories**.
* **Example:** A diabetic patient asking about “hypertension treatment” will get guidance factoring in their chronic condition, with emphasis on safe lifestyle practices.

**4. Safety-First Design**

* **Key Point:** Ethical AI usage.
* **Functionality:**  
  Every response includes disclaimers to remind users that the assistant is not a doctor. It emphasizes consulting healthcare professionals, ensuring that users don’t treat AI advice as a substitute for medical consultation.

**5. User-Friendly Gradio UI**

* **Key Point:** Intuitive and accessible interface.
* **Functionality:**  
  The project uses **Gradio Blocks** to provide an interactive and visually appealing dashboard with tabs for “Disease Prediction” and “Treatment Plans.”
  + Input boxes allow users to type symptoms or conditions.
  + Buttons trigger AI-powered analysis.
  + Results are displayed in large, scrollable text fields for clarity.
* **Example:** A patient can switch between predicting conditions and generating treatment plans with just one click.

**6. Expandability (Future Potential)**

* **Key Point:** Scalable design.
* **Functionality:**  
  Though the project currently supports **text-based analysis**, it can be extended to support:
  + **Voice input** for accessibility.
  + **Multilingual support** for diverse populations.
  + **Integration with wearable devices** (e.g., smartwatches, health bands).
  + **Cloud-based patient history tracking** with secure authentication.

**Benefits of the Medical AI Assistant**

1. **For Citizens / Patients:**
   * Quick answers to common health questions.
   * General advice without needing immediate doctor visits.
   * Increased awareness about preventive care.
2. **For Doctors / Healthcare Systems:**
   * Helps pre-screen patient concerns.
   * Saves time during consultations by filtering basic queries.
   * Reduces misinformation spread by giving AI-curated responses.
3. **For Researchers / Students:**
   * Demonstrates how AI can be ethically applied in healthcare.
   * Provides a live working system that bridges theory and application.

**Alignment with AI for Social Good**

The project aligns with the **AI for Social Good** initiative by using advanced AI technology to **improve health literacy and accessibility**. It emphasizes **sustainability in healthcare** by reducing unnecessary strain on hospitals while empowering individuals with the knowledge to make informed decisions.

**3. Architecture**

1. **Frontend (Gradio UI):**
   * Provides the user-facing interface where patients can enter symptoms or conditions.
   * Displays AI-generated results in a clean, conversational format.
   * Implements tab-based navigation for “Disease Prediction” and “Treatment Plans.”
2. **Backend (Python + Transformers):**
   * Responsible for handling requests from the frontend.
   * Encodes user inputs into tokens that can be processed by the language model.
   * Manages response generation, applying rules like maximum length and temperature for natural responses.
3. **LLM Integration (IBM Granite LLMs):**
   * Core engine of the assistant, built on top of Hugging Face’s transformers library.
   * The model processes user inputs and generates contextually accurate and conversational outputs.
   * Specialized prompts guide the model to produce medically aligned and safety-aware responses.
4. **Device & Resource Handling:**
   * The system detects whether a GPU (CUDA-enabled) is available.
   * If available, the model runs on GPU with torch.float16 for efficiency.
   * If not, it gracefully falls back to CPU with torch.float32.
5. **Output Layer:**
   * Decodes model-generated tokens into human-readable text.
   * Cleans up unnecessary repetitions or special tokens.
   * Ensures final responses are concise, clear, and accompanied by disclaimers.

**Frontend (Gradio Blocks)**

The frontend of the application is built using **Gradio Blocks**, a modern framework that allows rapid prototyping of interactive machine learning interfaces.

**Key Features of the Frontend:**

* **Tabbed Layout:**
  + **Disease Prediction Tab**: Users input symptoms and get AI analysis.
  + **Treatment Plan Tab**: Users input condition, age, gender, and medical history.
* **Widgets Used:**
  + **Textbox**: For inputting symptoms, medical condition, and medical history.
  + **Dropdown**: For selecting gender.
  + **Number Box**: For entering age.
  + **Buttons**: For triggering predictions or treatment generation.
  + **Output Areas**: Scrollable textboxes for results.
* **User Experience Focus:**
  + Minimalist design to avoid overwhelming patients.
  + Clear labels and placeholder text for guidance.
  + Immediate, real-time feedback on button clicks.

**Backend (Model & Logic)**

The backend is where all the processing happens. It is implemented in **Python**, leveraging **PyTorch** and the **Transformers library**.

**Core Functions:**

1. **generate\_response(prompt, max\_length=1024)**
   * Encodes the prompt into tokens.
   * Sends it to the model for text generation.
   * Applies temperature (0.7) for balanced creativity and accuracy.
   * Decodes results into human-readable responses.
2. **disease\_prediction(symptoms)**
   * Constructs a medical-specific prompt around symptoms.
   * Emphasizes disclaimers and professional consultation.
   * Returns structured possible conditions and general advice.
3. **treatment\_plan(condition, age, gender, medical\_history)**
   * Takes user demographic and health details.
   * Generates a more personalized treatment suggestion.
   * Includes lifestyle tips, home remedies, and medication categories.

**LLM Integration (Granite + Transformers)**

The system uses **IBM Granite LLMs** integrated via Hugging Face.

* **Tokenizer:** Splits user text into tokens the model can process.
* **Model:** Pre-trained AutoModelForCausalLM designed for conversation and instruction following.
* **Safety Considerations:**
  + Prompts explicitly instruct the model to avoid providing definitive diagnoses or prescribing drugs.
  + Every response ends with a safety disclaimer.

**Textual Diagram Representation**

Since I can’t embed an actual diagram here, below is a **step-by-step representation of the architecture**:

[ User Interface (Gradio UI) ]

|

v

[ Input: Symptoms / Condition / History ]

|

v

[ Backend (Python Functions) ]

|

|--> [ Prompt Engineering ]

|--> [ Tokenization (Tokenizer) ]

v

[ Granite AI Model (LLM) via Transformers ]

|

v

[ Output Decoding + Cleanup ]

|

v

[ User Output Display (Gradio Textbox) ]

**4. Setup Instructions**

**4.1 Prerequisites**

Before running the project, make sure the following prerequisites are met:

**Hardware Requirements:**

* **CPU:** Intel i5 or higher (for CPU-only execution).
* **GPU (optional but recommended):** NVIDIA GPU with CUDA support (for faster inference).
* **RAM:** At least 8 GB (16 GB recommended if running locally without GPU).
* **Disk Space:** 5–10 GB free for model downloads, libraries, and cache.

**Software Requirements:**

* **Operating System:** Windows 10/11, Linux (Ubuntu 20.04+), or macOS.
* **Python Version:** Python 3.9 or later.
* **Pip:** Python package manager installed and updated.
* **Virtual Environment Tool:** venv or conda for dependency isolation.
* **Internet Access:** Required to download the Granite model, tokenizer, and dependencies.

**4.2 Installation Process**

Follow these steps to set up the project:

**Step 1: Clone the Repository (if hosted on GitHub)**

git clone https://github.com/your-username/medical-ai-assistant.git

cd medical-ai-assistant

If the project is not hosted on GitHub, simply copy the code files into a new local directory.

**Step 2: Create and Activate Virtual Environment**

**For Windows:**

python -m venv venv

venv\Scripts\activate

**For Linux/Mac:**

python3 -m venv venv

source venv/bin/activate

**Step 3: Install Dependencies**

All dependencies should be listed in a requirements.txt file. Example:

gradio==4.0.2

torch==2.1.0

transformers==4.35.0

accelerate==0.24.1

To install them:

pip install -r requirements.txt

If a requirements.txt file does not exist, manually install:

pip install gradio torch transformers accelerate

**Step 4: Download and Configure the Model**

The assistant uses **IBM Granite LLM** available via Hugging Face. The model is automatically downloaded the first time you run the script.

Code snippet in main.py (already included in your project):

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

)

⚠️ **Note:** Model download may take time depending on internet speed (several GBs).

**Step 5: Configure Environment Variables (Optional)**

If you plan to deploy or extend the project, create a .env file to store:

* API keys (if integrating external services).
* Deployment configurations (cloud hosting, authentication).

Example .env:

MODEL\_NAME=ibm-granite/granite-3.2-2b-instruct

APP\_PORT=7860

You can use python-dotenv to load these variables.

**Step 6: Launch the Application**

Run the application:

python main.py

After successful launch, you’ll see output like:

Running on local URL: http://127.0.0.1:7860

Running on public URL: https://xxxx.gradio.live

* Open the given link in a browser.
* Use the Gradio interface with tabs for **Disease Prediction** and **Treatment Plans**.

**Step 7: Testing the Setup**

To ensure everything works:

1. Go to the **Disease Prediction** tab.
2. Enter:
3. fever, headache, fatigue

and click **Analyze Symptoms**.

1. The model should return possible conditions like viral fever, flu, or dengue, along with a disclaimer.

Then, test the **Treatment Plan tab** with:

Condition: Hypertension

Age: 45

Gender: Male

History: Diabetes

Expected output: A personalized treatment plan with home remedies, lifestyle suggestions, and disclaimers.

**5. Folder Structure**

A well-organized folder structure is essential for the maintainability and scalability of any software project. The **Medical AI Assistant** follows a **modular directory structure**, separating frontend, backend, and AI-related logic. This ensures that new contributors can quickly understand the workflow, while developers can extend features without breaking existing code.

Below is the recommended **folder hierarchy**:

medical-ai-assistant/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── api/

│ │ ├── \_\_init\_\_.py

│ │ ├── disease.py

│ │ ├── treatment.py

│ │ └── utils.py

│ │

│ ├── core/

│ │ ├── \_\_init\_\_.py

│ │ ├── model\_loader.py

│ │ ├── prompt\_builder.py

│ │ └── response\_cleaner.py

│ │

│ └── config/

│ └── settings.py

│

├── ui/

│ ├── \_\_init\_\_.py

│ ├── main\_ui.py

│ └── components/

│ ├── disease\_tab.py

│ ├── treatment\_tab.py

│ └── shared.py

│

├── tests/

│ ├── test\_api.py

│ ├── test\_prompts.py

│ └── test\_ui.py

│

├── assets/

│ ├── screenshots/

│ └── styles/

│

├── requirements.txt

├── main.py

├── README.md

└── .env.example

**5.Folder structure**

**5.1 Folder and File Descriptions**

**1. app/ – Backend Logic**

This is the **heart of the project**. It contains all business logic, API functions, and AI integration modules.

* **\_\_init\_\_.py**: Marks the folder as a Python package.
* **api/**: Contains API routes and handlers.
  + disease.py: Defines the disease\_prediction() function, which takes symptoms and generates possible conditions.
  + treatment.py: Implements treatment\_plan() logic for personalized treatment suggestions.
  + utils.py: Common helper functions such as input validation and formatting.
* **core/**: Houses core logic for interacting with the AI model.
  + model\_loader.py: Loads the Granite LLM model and tokenizer. Handles GPU/CPU device mapping.
  + prompt\_builder.py: Creates structured prompts for disease prediction and treatment generation.
  + response\_cleaner.py: Cleans raw model outputs, removes repetitions, and appends disclaimers.
* **config/**: Centralized settings for model name, max token length, temperature, etc.

**2. ui/ – Frontend Components**

This folder handles the **user interface layer**, built using Gradio.

* **\_\_init\_\_.py**: Package initializer.
* **main\_ui.py**: Entry point for the frontend. Defines Gradio Blocks layout with tabs.
* **components/**: Holds modular UI files.
  + disease\_tab.py: Layout and logic for the Disease Prediction tab.
  + treatment\_tab.py: Layout and logic for the Treatment Plan tab.
  + shared.py: Common UI elements (disclaimer banners, styled buttons).

This modular approach makes it easier to extend the UI later—for example, adding a “Mental Health” tab.

**3. tests/ – Testing Suite**

Testing ensures reliability and correctness of the system.

* **test\_api.py**: Unit tests for API endpoints (symptom analysis, treatment plan generation).
* **test\_prompts.py**: Validates correctness of prompts and ensures disclaimers are always included.
* **test\_ui.py**: Tests UI components to confirm they render correctly in Gradio.

**4. assets/ – Media & Styling**

This folder stores static assets such as screenshots, logos, and custom CSS.

* **screenshots/**: Used in documentation (UI snapshots for disease prediction and treatment outputs).
* **styles/**: Custom CSS for styling the Gradio interface if required.

**5. Root-Level Files**

* **requirements.txt**: Lists Python dependencies (gradio, torch, transformers, etc.).
* **main.py**: The main script that ties together frontend and backend. This is the file you run with python main.py.
* **README.md**: Project documentation (short overview, setup instructions, usage).
* **.env.example**: Sample environment configuration file for storing credentials or app settings.

**5.2 How the Files Work Together**

Here’s the step-by-step **execution workflow** when a user runs the app:

1. **Execution Start:**
   * User runs python main.py.
   * main.py imports main\_ui.py from the ui/ folder.
2. **UI Initialization:**
   * main\_ui.py loads disease and treatment components from ui/components/.
   * Displays the Gradio interface.
3. **User Interaction:**
   * User enters symptoms in the Disease Prediction tab.
   * disease\_tab.py calls the disease\_prediction() function from app/api/disease.py.
4. **Backend Processing:**
   * disease.py → calls prompt\_builder.py to create structured prompt.
   * model\_loader.py loads Granite LLM.
   * response\_cleaner.py sanitizes the output.
5. **Output Returned:**
   * Backend sends processed text back to the UI.
   * Results are displayed in the output textbox along with disclaimers.

**6.Running the Application**

**6.1 Local Execution**

**Step 1: Start the Application**

Navigate to the root project directory (medical-ai-assistant/) and run:

python main.py

* If the setup was successful, you should see output similar to:
* Running on local URL: http://127.0.0.1:7860
* Running on public URL: https://xxxx.gradio.live
* The **local URL** is accessible only on your machine.
* The **public Gradio link** can be shared with others for temporary remote access.

**Step 2: Accessing the Interface**

* Open your browser and visit the local or public URL.
* You will see a **Gradio interface** with:
  + A **Disease Prediction Tab**
  + A **Treatment Plan Tab**
  + Disclaimer banners for medical safety

**Step 3: Using the Application**

**1. Disease Prediction Tab**

* Input:
* fever, headache, fatigue
* Backend:
  + Converts symptoms into a structured prompt.
  + Sends prompt to Granite LLM.
  + Cleans raw response and adds disclaimers.
* Output:
* Possible Conditions:
* - Viral Fever
* - Dengue
* - Influenza
* (Note: This is an AI suggestion and not a medical diagnosis.)

**2. Treatment Plan Tab**

* Input:
* Condition: Hypertension
* Age: 45
* Gender: Male
* History: Diabetes
* Backend:
  + Builds a personalized treatment prompt.
  + Returns a structured care plan.
* Output:
* Suggested Treatment Plan:
* - Lifestyle: Reduce salt intake, regular walking
* - Monitoring: Check BP twice daily
* - Medications: Consult doctor for antihypertensives
* - Home Remedies: Deep breathing, hydration
* Disclaimer: Always consult a certified physician before starting treatment.

**6.2 Running in Development Mode**

If you want to modify code and test changes:

app.launch(debug=True, share=True)

* **debug=True** → Logs detailed errors for debugging.
* **share=True** → Provides a public Gradio link.

This mode is ideal for developers making iterative improvements.

**6.3 Running in Production Mode**

For production deployments, consider the following options:

**Option 1: Docker Deployment**

Create a Dockerfile:

FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "main.py"]

Build and run:

docker build -t medical-ai .

docker run -p 7860:7860 medical-ai

Now the app will be available at http://localhost:7860.

**Option 2: Cloud Deployment**

Deploy the app to cloud platforms:

* **AWS EC2**: Launch an Ubuntu instance, install dependencies, and run the app.
* **Google Cloud Run / IBM Cloud**: Use Docker image for containerized deployment.
* **Heroku / Render**: Deploy using Procfile.

**Option 3: Gradio Sharing**

For demonstrations without deploying on servers:

app.launch(share=True)

* Generates a **temporary link** hosted by Gradio.
* Suitable for showcasing the project to peers or clients.
  + to GPU/cloud execution.

**7. Testing**

**7.1 Unit Testing**

Unit testing focuses on verifying **individual components or functions** of the application to ensure they behave as expected.

**Core Areas Tested:**

1. **Prompt Generation (prompt\_builder.py)**
   * Test that symptom inputs produce correctly structured prompts for Granite LLM.
   * Ensure disclaimers and safety instructions are always included.
   * Validate that prompts do not exceed the model’s maximum token length.
2. **Model Loader (model\_loader.py)**
   * Verify that the model loads correctly on both CPU and GPU.
   * Check tokenizer initialization and pad token assignment.
   * Ensure device mapping works properly (device\_map="auto" for GPU, None for CPU).
3. **Response Cleaner (response\_cleaner.py)**
   * Remove any duplicate sentences or extraneous special tokens.
   * Append disclaimers to every response.
   * Ensure trimming and formatting are consistent for display in Gradio UI.
4. **API Functions (api/disease.py & api/treatment.py)**
   * Validate that input data types are handled correctly (strings, integers).
   * Confirm output contains structured text with predictions or treatment suggestions.

**Example Unit Test using pytest:**

from app.api.disease import disease\_prediction

def test\_disease\_prediction\_output():

symptoms = "fever, headache, fatigue"

response = disease\_prediction(symptoms)

assert "Possible Conditions" in response

assert "Disclaimer" in response

**7.2 Integration Testing**

Integration tests ensure that **different modules work together correctly**, focusing on end-to-end functionality.

**Key Integration Scenarios:**

1. **Frontend → Backend → LLM → Output**
   * Input symptoms from Gradio textbox.
   * Validate that backend functions receive the input and return processed results.
   * Confirm final output appears in the UI with proper formatting.
2. **Treatment Plan Generation Flow**
   * Input condition, age, gender, and medical history.
   * Ensure AI generates a relevant treatment plan.
   * Verify all necessary disclaimers and home remedy suggestions are included.
3. **Device Compatibility**
   * Test integration on machines with GPU and CPU.
   * Validate model performance and execution time.

**7.3 API Testing**

If the backend is exposed via API endpoints (for example, using FastAPI in future upgrades), testing focuses on **request-response validation**.

**Sample Endpoints:**

1. **POST /disease-prediction**
   * Input: JSON with symptoms field
   * Output: JSON with conditions and disclaimer fields
2. **POST /treatment-plan**
   * Input: JSON with condition, age, gender, history
   * Output: JSON with treatment\_plan and disclaimer

**Testing Tools:**

* **Postman:** Send sample requests and validate responses.
* **Swagger UI:** Test endpoints interactively.
* **Automated Scripts:** Python scripts to simulate multiple inputs and verify output structure.

**8. API Documentation**

**8.1 API Endpoints Overview**

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /api/disease-prediction | POST | Accepts user symptoms and returns possible medical conditions along with recommendations. |
| /api/treatment-plan | POST | Accepts patient condition, age, gender, and medical history; returns personalized treatment suggestions. |
| /api/health-tips | GET | Returns general health and wellness tips based on common conditions (optional). |
| /api/feedback | POST | Stores user feedback for system improvement and analysis. |

**8.2 POST /api/disease-prediction**

**Purpose:** Generate potential medical conditions based on user-provided symptoms.

**Request Format:**

{

"symptoms": "fever, headache, fatigue"

}

**Response Format:**

{

"conditions": [

"Viral Fever",

"Influenza",

"Dengue"

],

"recommendations": [

"Stay hydrated",

"Rest",

"Monitor temperature regularly"

],

"disclaimer": "This is for informational purposes only. Always consult a healthcare professional."

}

**Example Usage:**

curl -X POST http://localhost:8000/api/disease-prediction \

-H "Content-Type: application/json" \

-d '{"symptoms": "fever, headache, fatigue"}'

**Notes:**

* Symptoms should be a comma-separated string.
* Response always includes a disclaimer to maintain medical safety.

**8.3 POST /api/treatment-plan**

**Purpose:** Generate a personalized treatment plan using patient-specific details.

**Request Format:**

{

"condition": "Hypertension",

"age": 45,

"gender": "Male",

"medical\_history": "Diabetes"

}

**Response Format:**

{

"treatment\_plan": [

"Lifestyle: Reduce salt intake, walk 30 minutes daily",

"Monitoring: Check blood pressure twice daily",

"Medications: Consult doctor for antihypertensives",

"Home Remedies: Deep breathing, hydration"

],

"disclaimer": "This plan is for informational purposes only. Always consult a certified physician before starting treatment."

}

**Example Usage:**

curl -X POST http://localhost:8000/api/treatment-plan \

-H "Content-Type: application/json" \

-d '{"condition": "Hypertension", "age": 45, "gender": "Male", "medical\_history": "Diabetes"}'

**Notes:**

* Age must be a positive integer.
* Gender can be Male, Female, or Other.
* Medical history is optional; if left empty, the AI assumes no prior conditions.

**8.4 GET /api/health-tips (Optional)**

**Purpose:** Provide general wellness suggestions and eco-friendly health tips.

**Request:**

GET /api/health-tips?category=nutrition

**Response Format:**

{

"category": "nutrition",

"tips": [

"Include more fruits and vegetables in your diet",

"Stay hydrated",

"Avoid processed foods"

]

}

**Notes:**

* Categories can include nutrition, exercise, sleep, or mental-health.
* Useful for additional features like “daily tips” in the app.

**8.5 POST /api/feedback**

**Purpose:** Collect user feedback to improve system performance and recommendations.

**Request Format:**

{

"user\_id": "12345",

"feedback": "The disease prediction was helpful but could include more symptoms."

}

**Response Format:**

{

"status": "success",

"message": "Thank you for your feedback!"

}

**Notes:**

* Feedback is stored in a database or a CSV for later analysis.
* Helps train future AI models and refine prompt engineering.

**9. Authentication**

**9.1 Importance of Authentication**

1. **Protects User Data:**
   * Ensures that sensitive information like medical history, symptoms, and treatment plans is accessible only to authorized users.
2. **Prevents Unauthorized Access:**
   * Protects backend services from misuse or malicious attempts to exploit the AI system.
3. **Maintains Accountability:**
   * Logs and tracks user activity, making it easier to audit system usage and monitor abnormal behavior.
4. **Supports Role-Based Functionality:**
   * Different users may have different access levels, e.g., **patients, clinicians, or administrators**.

**9.2 Authentication Methods**

**1. Token-Based Authentication (JWT)**

* **Description:** JSON Web Tokens (JWT) provide a compact and secure way to transmit information between the client and server.
* **Flow:**
  1. User logs in with credentials.
  2. Server validates credentials and issues a JWT token.
  3. Client includes the token in request headers for API access.
  4. Server validates the token before processing the request.

**2. OAuth2 with IBM Cloud Credentials**

* **Description:** OAuth2 provides delegated access, allowing users to log in using existing accounts like IBM Cloud or corporate credentials.
* **Flow:**
  1. User clicks “Login via IBM Cloud.”
  2. Redirect to IBM OAuth2 authorization page.
  3. Upon approval, the app receives an access token.
  4. Token used for authenticated API requests.

**3. Role-Based Access Control (RBAC)**

* **Purpose:** Define access rights for different types of users:
  + **Admin:** Full access to all functionalities, system settings, and analytics.
  + **Clinician/Researcher:** Access to patient input, treatment suggestions, and analytics.
  + **Patient/User:** Access only to their own queries, predictions, and plans.
* **Implementation:**
  + Roles stored in JWT payloads or user database.
  + Backend checks role before processing requests or displaying sensitive information.

**10. User Interface**

**10.1 Layout Overview**

The interface is structured into **two main tabs**, each catering to a specific functionality:

1. **Disease Prediction Tab**
2. **Treatment Plan Tab**

**Sidebar and Header**

* **Header:** Displays the project title (Medical AI Assistant) and disclaimer.
* **Sidebar:** Can be extended in future versions for navigation, including links to **Health Tips**, **Feedback**, or **Settings**.
* **Purpose:** Provides immediate context, user guidance, and ensures ethical compliance through disclaimers.

**10.2 Disease Prediction Tab**

**Components:**

1. **Symptoms Input Textbox:**
   * Placeholder: "e.g., fever, headache, cough, fatigue..."
   * Accepts multiple symptoms as a comma-separated list.
   * Multi-line textbox for readability.
2. **Analyze Symptoms Button:**
   * Triggers the disease\_prediction() function in the backend.
3. **Output Textbox:**
   * Displays possible conditions and general recommendations.
   * Multi-line textbox with scrollbar to handle longer outputs.
   * Includes **disclaimer** to ensure users understand that predictions are for informational purposes only.

**10.3 Treatment Plan Tab**

**Components:**

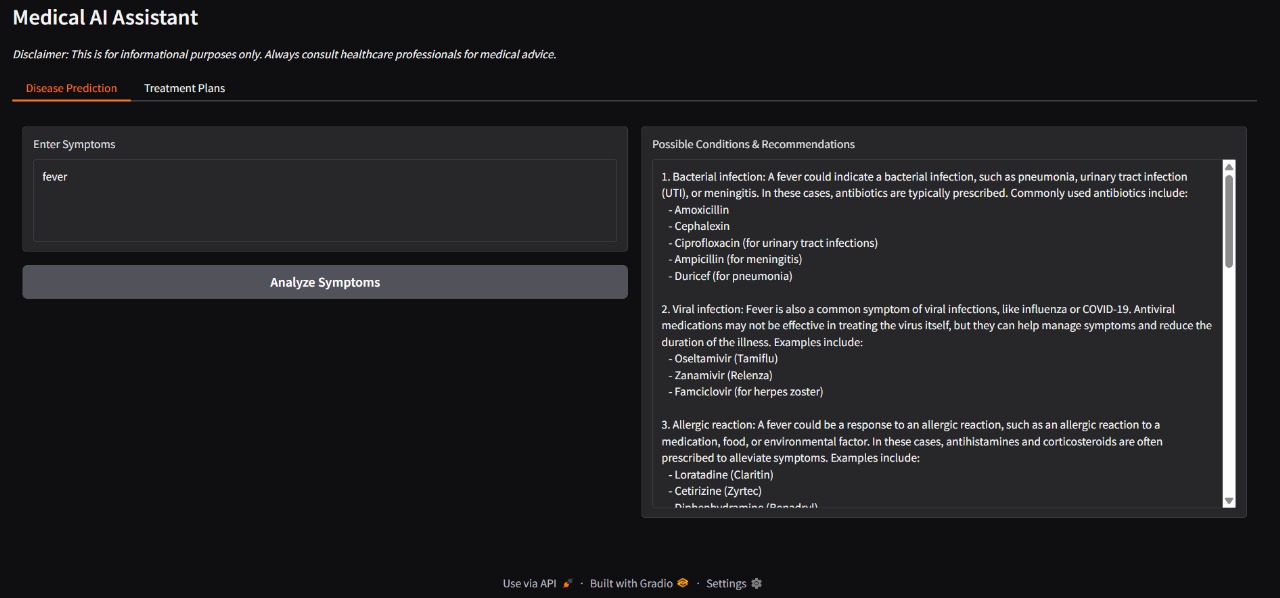
1. **Medical Condition Input:**
   * Textbox for specifying the diagnosed or suspected condition.
   * Placeholder: "e.g., diabetes, hypertension, migraine..."
2. **Age Input:**
   * Number input field for patient age.
   * Default value set to 30; validates positive integers.
3. **Gender Dropdown:**
   * Options: Male, Female, Other.
   * Ensures gender-specific considerations are factored in treatment suggestions.
4. **Medical History Input:**
   * Multi-line textbox for past conditions, allergies, or medications.
   * Optional; default is "None".
5. **Generate Treatment Plan Button:**
   * Calls treatment\_plan() function in the backend.
6. **Output Textbox:**
   * Displays the personalized treatment plan including:
     + Home remedies
     + General medication guidelines
     + Lifestyle recommendations
   * Includes disclaimer emphasizing professional medical consultation.

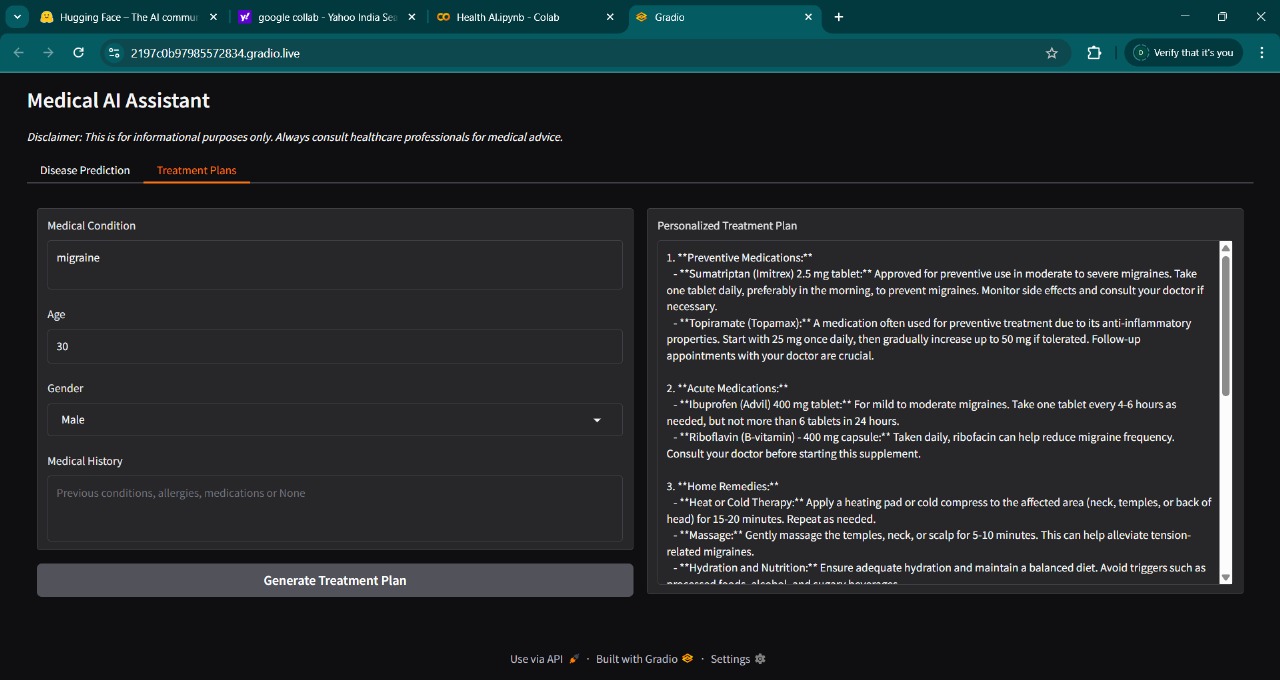
**10.4 Visual and Functional Design**

**Design Principles:**

1. **Minimalist Layout:**
   * Focus on key inputs and outputs.
   * Avoid clutter to improve user comprehension.
2. **Accessibility:**
   * Multi-line input fields for longer text.
   * Clear labels and placeholders.
   * Tabbed layout allows users to switch functions without confusion.
3. **Real-Time Feedback:**
   * Buttons trigger backend calls asynchronously.
   * Outputs appear dynamically without page reloads.
4. **Disclaimers:**
   * Prominent placement in each tab ensures user awareness.
   * Reinforces that predictions are informational, not diagnostic.

**11. Screenshots**

****



**12. Known Issues**

**12.1 AI Model Limitations**

1. **Incomplete or Inaccurate Predictions:**
   * The AI may not always provide a correct or complete diagnosis because it generates outputs based on **trained data and prompts**, not real medical examination.
   * **Impact:** Users could receive suggestions that are partially correct or too generic.
   * **Workaround:** Always emphasize disclaimers and recommend consulting a certified physician.
2. **Bias in AI Responses:**
   * The Granite LLM may reflect biases present in its training data.
   * **Impact:** Some symptoms or conditions may be overrepresented or underrepresented.
   * **Workaround:** Cross-verify critical results with trusted medical sources.
3. **Limited Context Understanding:**
   * The AI can process **current inputs** but cannot fully remember previous queries or maintain long-term patient history.
   * **Impact:** Recommendations may lack continuity over multiple sessions.
   * **Future Fix:** Implement session management and history tracking for continuity.
4. **Token Limitations:**
   * The AI model has a **maximum token limit** per input/output.
   * **Impact:** Very long symptom lists or detailed medical history may be truncated, affecting response quality.
   * **Workaround:** Encourage users to input concise symptoms and history.

**12.2 Frontend Issues**

1. **Scroll and Display Limitations:**
   * Large outputs may require scrolling, and excessive text may affect readability.
   * **Workaround:** Consider pagination or expandable output boxes.
2. **Responsiveness on Low-End Devices:**
   * On devices with low RAM or slower CPUs, the UI may lag.
   * **Future Fix:** Optimize frontend layout and implement lightweight rendering.
3. **Limited Tab Features:**
   * Current tabs only support **Disease Prediction** and **Treatment Plan**.
   * **Future Fix:** Add extra tabs for health tips, feedback, and reports.

**12.3 Backend Issues**

1. **CPU-Only Performance:**
   * Without GPU, AI responses may take longer (8–15 seconds).
   * **Workaround:** Reduce max\_length parameter for faster processing or deploy on GPU/cloud.
2. **Concurrency Limitations:**
   * Gradio’s default server has limited concurrent request handling.
   * **Impact:** Multiple simultaneous users may experience delays.
   * **Future Fix:** Deploy using production-ready backend (FastAPI or Flask with load balancing).
3. **Error Handling:**
   * Edge cases like empty inputs or unsupported characters may sometimes generate warnings or minimal error messages.
   * **Workaround:** Frontend validation ensures proper inputs before submission.

**13. Future Enhancements**

**13.1 Enhanced AI Capabilities**

1. **Context-Aware Predictions**
   * Integrate **session-based memory** to track user inputs over time.
   * Enables the system to provide **longitudinal care suggestions**, taking previous symptoms and treatments into account.
2. **Multi-Language Support**
   * Add support for **regional languages**, making the assistant accessible to a wider population.
   * Use translation APIs or multilingual LLMs to maintain accurate medical guidance.
3. **Adaptive AI Responses**
   * Improve prompt engineering to adjust AI responses based on:
     + Patient age
     + Gender
     + Medical history
   * Ensures more **personalized and relevant outputs**.

**13.2 Expanded Health Modules**

1. **Nutrition and Lifestyle Guidance**
   * Provide **dietary recommendations, exercise tips, and wellness strategies** tailored to specific conditions.
   * Example: Diabetes → carbohydrate control and exercise routines.
2. **Mental Health Module**
   * Include **stress management, sleep improvement, and mental wellness tips**.
   * Can integrate with AI-driven mood analysis and mindfulness recommendations.
3. **Preventive Health Monitoring**
   * Add features for **tracking vitals**, suggesting regular checkups, and generating **preventive care alerts**.
   * Integration with wearable devices or IoT sensors is possible for real-time monitoring.

**13.3 User Interface Enhancements**

1. **Interactive Dashboards**
   * Visualize predictions, treatment plans, and health trends in **charts and graphs**.
   * Add **color-coded alerts** for urgent conditions or critical patterns.
2. **Report Generation**
   * Allow users to **download AI-generated reports** for personal records or sharing with doctors.
   * Include charts, summaries, and treatment recommendations in PDF or Excel format.
3. **Mobile Application Support**
   * Develop **Android/iOS versions** of the assistant for on-the-go access.
   * Ensure synchronization with the web version and cloud backend.

**13.4 Backend and API Improvements**

1. **Secure Authentication and User Management**
   * Implement **JWT tokens, OAuth2, and multi-factor authentication**.
   * Role-based access to ensure **data privacy and secure information flow**.
2. **Performance Optimization**
   * Deploy LLM models on **GPU or cloud infrastructure** to reduce latency.
   * Enable **batch processing** and **caching** for repeated queries.
3. **Scalable Architecture**
   * Implement **microservices architecture** for AI inference, analytics, and reporting.
   * Easier maintenance, modular updates, and independent scaling of components.