### HEALTH AI : INTELLIGENT HEALTHCARE ASSISTANCE

### 

### 

### 

### PROJECT DOCUMENTATION

### INTRODUCTION

### PROJECT TITLE : HEALTH AI

### TEAM MEMBERS : PRIYA DHARSHINI.C

### TEAM MEMBERS : POOJA.V

### TEAM MEMBERS : POORNIMA.K

### TEAM MEMBERS : POONGUZHALI.R

### PROJECT OVERVIEW

### PURPOSE

### ]Objective: The project aims to build a smart healthcare assistant using IBM Granite models (from Hugging ) integrated with Gradio. The assistant can provide:

### Patient chat support

### Disease prediction

### Treatment plan suggestions

### And other healthcare functionalities

### Technology & Tools Used:

### IBM Granite Models (Hugging Face) – For AI-powered healthcare guidance

### Gradio – For creating interactive applications

### Python – Main programming language

### Google Colab (T4 GPU) – For running and testing the application

### GitHub – For version control and project sharing

### Key Features:

### Simple and easy-to-use healthcare assistant

### Runs in Google Colab for accessibility

### Provides fast and secure medical guidance

### Interactive interface built with Gradio

### Project Workflow (Step-by-step):

### Explore Naan Mudhalvan Smart Interz Portal – Access project resources and workspace.

### Choose IBM Granite Model – Select a model like *granite-3.2-2b-instruct* from Hugging Face.

### Run in Google Colab – Install required libraries, set GPU runtime, and execute code.

### Upload to GitHub – Save and share your project by uploading the code repository.

### CONVERSATIONAL INTERFACE

### # Install required libraries

### !pip install transformers torch gradio -q

### import gradio as gr

### from transformers import AutoModelForCausalLM, AutoTokenizer

### import torch

### # Load IBM Granite model (example: granite-3.2-2b-instruct)

### model\_name = "ibm-granite/granite-3.2-2b-instruct"

### tokenizer = AutoTokenizer.from\_pretrained(model\_name)

### model = AutoModelForCausalLM.from\_pretrained(model\_name)

### # Function for chatbot response

### def health\_ai\_chat(user\_input, chat\_history=[]):

### # Encode input with history

### inputs = tokenizer(user\_input, return\_tensors="pt")

### outputs = model.generate(

### \*\*inputs,

### max\_length=200,

### temperature=0.7,

### top\_p=0.9

### )

### response = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

### chat\_history.append(("You: " + user\_input, "AI: " + response))

### return chat\_history, chat\_history

### # Gradio conversational UI

### with gr.Blocks() as demo:

### gr.Markdown("## 🏥 Health AI Assistant – Conversational Interface")

### chatbot = gr.Chatbot()

### msg = gr.Textbox(placeholder="Ask me about health, symptoms, or treatments...")

### state = gr.State([])

### def respond(message, chat\_history):

### chat\_history, updated\_history = health\_ai\_chat(message, chat\_history)

### return updated\_history, updated\_history

### msg.submit(respond, [msg, state], [chatbot, state])

### demo.launch()

### 

### What this does:

### Creates a chatbot interface using Gradio.

### Users can type questions like *“What are the symptoms of diabetes?”* or *“Suggest a treatment for fever”*.

### The AI responds conversationally.

### Maintains chat history for natural conversation.

### POLICY SUMMARIZATION :

### 

### Definition: Policy Summarization is the process of condensing complex healthcare policies, medical guidelines, or insurance documents into short, easy-to-understand summaries using AI.

### Why it’s important in Healthcare AI:

### Healthcare policies are often long, technical, and difficult for patients to understand.

### Doctors, patients, and caregivers need quick access to key rules, coverage details, and treatment guidelines.

### AI can automatically analyze, extract, and summarize the important parts of these documents.

### Use Cases in Health AI:

### Medical Guidelines: Summarize WHO/ICMR/CDC guidelines for doctors & patients.

### Insurance Policies: Provide a clear summary of what’s covered and not covered.

### Hospital Policies: Summarize hospital rules like admission, discharge, and billing procedures.

### Government Health Schemes: Summarize eligibility, benefits, and claim processes.

### How Policy Summarization Works (with AI):

### Input → Long healthcare policy (text/PDF).

### AI Model → Reads and identifies key points.

### Output → Short, simple summary in plain language.

### Example:

### Input (Insurance Policy): *“This plan covers hospitalization expenses up to ₹5,00,000 per year. Pre-existing conditions are covered after 2 years. Maternity benefits are available after 3 years.”*

### AI Summarized Output: *Covers hospitalization up to ₹5 lakhs/year. Pre-existing diseases covered after 2 years. Maternity covered after 3 years.*

### RESOURCE FORECASTING :

### 

### Definition: Resource Forecasting means predicting future needs of healthcare resources (doctors, beds, medicines, staff, equipment) using AI and data analysis.

### 🔸 Why it is Important in Healthcare:

### Better Planning → Hospitals can plan how many doctors, nurses, or beds will be required.

### Avoid Shortages → Prevents medicine or equipment shortages (like oxygen cylinders during COVID-19).

### Cost Efficiency → Helps allocate resources without wastage.

### Emergency Readiness → Predicts resource demand during outbreaks or seasonal diseases.

### 🔸 How It Works (AI-powered):

### Input Data: Past hospital records, patient flow, seasonal trends, disease outbreaks.

### AI Model (Forecasting): Uses predictive analytics to identify patterns.

### Output: Estimates required resources for future days/weeks.

### 🔸 Example in a Hospital:

### Input Data: Last 3 years’ flu patient records in October.

### Forecast Result: “Expect 40% increase in flu cases this October → Need 30 extra beds, 5 extra doctors, and more flu medicines.”

### 🔸 Application in Your Project (Health AI with IBM):

### You can add Resource Forecasting as a feature in your assistant:

### Predict patient load based on symptoms entered by users.

### Suggest hospital resource requirements.

### Help policy makers and hospital admins prepare in advance.

### Eco-Tip Generator

### **Definition:** An Eco-Tip Generator is an AI feature that provides **daily simple tips** to encourage eco-friendly habits in healthcare and daily life.

### 🔸 Why Add This?

### **Healthcare & Environment are linked** → Pollution, waste management, and energy use directly affect health.

### **Promotes Awareness** → Encourages patients and healthcare workers to adopt sustainable practices.

### **Value-Added Feature** → Makes your Health AI assistant stand out.

### 🔸 Example Eco-Tips

### “Turn off medical equipment when not in use to save electricity.”

### “Use reusable water bottles instead of plastic ones.”

### “Cycle or walk for short distances instead of driving.”

### “Segregate biomedical waste properly to reduce health hazards.”

### “Plant indoor air-purifying plants like Tulsi or Aloe Vera.”

### 🔸 How It Works in AI Project

### **Database / List of Eco-Tips** → Pre-store eco-friendly tips.

### **Randomization / AI Generation** → Either randomly pick or use AI to rephrase/generate new tips.

### **User Interaction** → Show a daily eco-tip when the user opens the chatbot.

### 🔸 Simple Python + Gradio Code

### import gradio as gr

### import random

### eco\_tips = [

### "Turn off lights and medical equipment when not in use to save energy.",

### "Carry a reusable water bottle to avoid single-use plastics.",

### "Use cloth bags instead of plastic bags.",

### "Segregate medical and household waste for safe disposal.",

### "Walk or cycle for short trips to reduce pollution.",

### "Plant more trees and indoor plants to improve air quality.",

### "Save water by turning off taps tightly after use.",

### "Use digital records instead of paper to reduce waste."

### ]

### def generate\_tip():

### return random.choice(eco\_tips)

### with gr.Blocks() as demo:

### gr.Markdown("## 🌍 Eco-Tip Generator")

### tip\_output = gr.Textbox(label="Your Eco Tip", interactive=False)

### generate\_btn = gr.Button("Get Eco Tip")

### generate\_btn.click(generate\_tip, outputs=tip\_output)

### demo.launch()

### Eco-Tip Generator

### **Definition:** An Eco-Tip Generator is an AI feature that provides **daily simple tips** to encourage eco-friendly habits in healthcare and daily life.

### 🔸 Why Add This?

### **Healthcare & Environment are linked** → Pollution, waste management, and energy use directly affect health.

### **Promotes Awareness** → Encourages patients and healthcare workers to adopt sustainable practices.

### **Value-Added Feature** → Makes your Health AI assistant stand out.

### 🔸 Example Eco-Tips

### “Turn off medical equipment when not in use to save electricity.”

### “Use reusable water bottles instead of plastic ones.”

### “Cycle or walk for short distances instead of driving.”

### “Segregate biomedical waste properly to reduce health hazards.”

### “Plant indoor air-purifying plants like Tulsi or Aloe Vera.”

### 🔸 How It Works in AI Project

### **Database / List of Eco-Tips** → Pre-store eco-friendly tips.

### **Randomization / AI Generation** → Either randomly pick or use AI to rephrase/generate new tips.

### **User Interaction** → Show a daily eco-tip when the user opens the chatbot.

### 🔸 Simple Python + Gradio Code

### import gradio as gr

### import random

### eco\_tips = [

### "Turn off lights and medical equipment when not in use to save energy.",

### "Carry a reusable water bottle to avoid single-use plastics.",

### "Use cloth bags instead of plastic bags.",

### "Segregate medical and household waste for safe disposal.",

### "Walk or cycle for short trips to reduce pollution.",

### "Plant more trees and indoor plants to improve air quality.",

### "Save water by turning off taps tightly after use.",

### "Use digital records instead of paper to reduce waste."

### ]

### def generate\_tip():

### return random.choice(eco\_tips)

### with gr.Blocks() as demo:

### gr.Markdown("## 🌍 Eco-Tip Generator")

### tip\_output = gr.Textbox(label="Your Eco Tip", interactive=False)

### generate\_btn = gr.Button("Get Eco Tip")

### generate\_btn.click(generate\_tip, outputs=tip\_output)

### demo.launch()

### **Definition:** An Eco-Tip Generator is an AI feature that provides **daily simple tips** to encourage eco-friendly habits in healthcare and daily life.

### 🔸 Why Add This?

### **Healthcare & Environment are linked** → Pollution, waste management, and energy use directly affect health.

### **Promotes Awareness** → Encourages patients and healthcare workers to adopt sustainable practices.

### **Value-Added Feature** → Makes your Health AI assistant stand out.

### 🔸 Example Eco-Tips

### “Turn off medical e­­­­­­­­­­­quipment when not in use to save electricity.”

### “Use reusable water bottles instead of plastic ones.”

### “Cycle or walk for short distances instead of driving.”

### “Segregate biomedical waste properly to reduce health hazards.”

### “Plant indoor air-purifying plants like Tulsi or Aloe Vera.”

### 🔸 How It Works in AI Project

### **Database / List of Eco-Tips** → Pre-store eco-friendly tips.

### **Randomization / AI Generation** → Either randomly pick or use AI to rephrase/generate new tips.

### **User Interaction** → Show a daily eco-tip when the user opens the chatbot.

### 🔸 Simple Python + Gradio Code

### import gradio as gr

### import random

### eco\_tips = [

### "Turn off lights and medical equipment when not in use to save energy.",

### "Carry a reusable water bottle to avoid single-use plastics.",

### "Use cloth bags instead of plastic bags.",

### "Segregate medical and household waste for safe disposal.",

### "Walk or cycle for short trips to reduce pollution.",

### "Plant more trees and indoor plants to improve air quality.",

### "Save water by turning off taps tightly after use.",

### "Use digital records instead of paper to reduce waste."

### ]

### def generate\_tip():

### return random.choice(eco\_tips)

### with gr.Blocks() as demo:

### gr.Markdown("## 🌍 Eco-Tip Generator")

### tip\_output = gr.Textbox(label="Your Eco Tip", interactive=False)

### generate\_btn = gr.Button("Get Eco Tip")

### generate\_btn.click(generate\_tip, outputs=tip\_output)

### demo.launch()

### 

### CITIZEN FEEDBACK LOOP :

### Definition: A Citizen Feedback Loop is a system where patients, users, or citizens share their feedback about healthcare services, AI recommendations, or hospital facilities — and this feedback is used to improve the system continuously.

### 🔸 Why It’s Important in Health AI

### Improves Accuracy → Users can report if AI answers are helpful or not.

### Builds Trust → Citizens feel involved in improving healthcare services.

### Policy Making → Helps hospitals & governments understand real public needs.

### Quality Control → Identifies errors, irrelevant suggestions, or missing features.

### 🔸 How It Works in Your Project

### User interacts with Health AI (asks about symptoms, treatment, policy, etc.).

### System asks for quick feedback → “Was this answer useful? 👍 / 👎”

### Store feedback in a database (Google Sheets, Firebase, or CSV file).

### Analyze feedback regularly → Improve model responses, add new features.

### 🔸 Example Feedback Questions

### Was this answer helpful? (Yes/No)

### Rate your experience (⭐ 1–5)

### What can we improve? (short text box)

### 🔸 Simple Python + Gradio Implementation

### import gradio as gr

### import csv

### import os

### # File to store feedback

### feedback\_file = "feedback.csv"

### # Ensure file exists

### if not os.path.exists(feedback\_file):

### with open(feedback\_file, mode="w", newline="") as f:

### writer = csv.writer(f)

### writer.writerow(["User Question", "AI Answer", "Rating", "Comments"])

### # Function to save feedback

### def save\_feedback(user\_question, ai\_answer, rating, comments):

### with open(feedback\_file, mode="a", newline="") as f:

### writer = csv.writer(f)

### writer.writerow([user\_question, ai\_answer, rating, comments])

### return "✅ Thank you! Your feedback has been recorded."

### with gr.Blocks() as demo:

### gr.Markdown("## 🗣 Citizen Feedback Loop for Health AI")

### user\_q = gr.Textbox(label="Your Question to AI")

### ai\_ans = gr.Textbox(label="AI's Answer")

### rating = gr.Radio(["⭐1", "⭐2", "⭐3", "⭐4", "⭐5"], label="Rate the Answer")

### comments = gr.Textbox(label="Additional Feedback")

### submit\_btn = gr.Button("Submit Feedback")

### result = gr.Textbox(label="Result", interactive=False)

### submit\_btn.click(save\_feedback, inputs=[user\_q, ai\_ans, rating, comments], outputs=result)

### demo.launch()

### ✅ **Result:**

### Users enter their **question, AI answer, rating, and comments**.

### Feedback gets stored in a **CSV file**.

### Admins can review feedback and update the system.

### KPI FORECASTING :

### Definition: KPI (Key Performance Indicator) Forecasting means predicting future values of important healthcare performance metrics using AI models.

### 🔸 Why It Matters in Healthcare

### Better Decision Making → Hospital admins and policymakers can see where performance is heading.

### Early Problem Detection → Spot declining KPIs (like patient satisfaction) before they get worse.

### Resource Planning → Forecast metrics like bed occupancy or medicine usage.

### Efficiency Tracking → Measure progress of digital health tools like your Health AI assistant.

### 🔸 Example Healthcare KPIs

### Patient Flow → Number of patients visiting per day/week.

### Bed Occupancy Rate → % of beds used in a hospital.

### Average Waiting Time → Time patients wait before treatment.

### Patient Satisfaction Score → Feedback ratings.

### Treatment Success Rate → % of successful outcomes.

### Readmission Rate → % of patients returning for the same issue.

### 🔸 How KPI Forecasting Works

### Collect Past KPI Data → Example: 12 months of patient inflow.

### Train Forecasting Model → AI learns patterns (seasonal trends, disease outbreaks).

### Predict Future KPI Values → Example: “Next month’s patient inflow is expected to increase by 15%.”

### Dashboard Visualization → Graphs, charts, and alerts.

### 🔸 Example Scenario

### Input Data: Bed occupancy from Jan–Dec 2024.

### AI Forecast: Predicts 90% occupancy in Feb 2025 → Suggests adding 20 beds.

### 🔸 Simple Python Example (Forecasting Patients)

### import pandas as pd

### import matplotlib.pyplot as plt

### from sklearn.linear\_model import LinearRegression

### import numpy as np

### # Example: Patient inflow data (month vs patients)

### data = {

### "Month": [1, 2, 3, 4, 5, 6],

### "Patients": [120, 135, 150, 160, 175, 190]

### }

### df = pd.DataFrame(data)

### # Prepare data for forecasting

### X = df[["Month"]]

### y = df["Patients"]

### model = LinearRegression()

### model.fit(X, y)

### # Forecast next 3 months

### future\_months = np.array([[7], [8], [9]])

### predictions = model.predict(future\_months)

### # Display

### for m, p in zip([7, 8, 9], predictions):

### print(f"📊 Forecast for Month {m}: {int(p)} patients")

### # Plot

### plt.plot(df["Month"], df["Patients"], marker='o', label="Actual")

### plt.plot([7,8,9], predictions, marker='x', linestyle="--", label="Forecast")

### plt.xlabel("Month")

### plt.ylabel("Patients")

### plt.legend()

### plt.show()

### ✅ Result: This simple model predicts future patient inflow (KPI) and shows it on a graph.

### ANOMALY DETECTION :

### Definition: Anomaly Detection means identifying unusual patterns or outliers in healthcare data that don’t follow the expected trend. These anomalies often indicate problems, risks, or errors that need quick attention.

### 🔸 Why It’s Important in Healthcare

### Patient Safety → Detect abnormal health readings (like sudden spikes in blood pressure).

### Fraud Detection → Spot unusual insurance claims or billing activities.

### Disease Outbreak Alerts → Identify unexpected increases in patient cases (like COVID-19).

### Operational Issues → Detect anomalies in hospital resource usage (beds, staff, medicines).

### 🔸 Types of Anomalies in Healthcare

### Medical Data Anomalies → Abnormal lab results, unusual symptoms.

### Operational Anomalies → Sudden bed shortages, abnormal waiting times.

### Financial Anomalies → Overbilling, duplicate claims.

### Usage Anomalies → Sudden surge in chatbot queries for a specific disease.

### 🔸 Example Scenarios

### Normal patient flow = 100–150/day → Suddenly jumps to 400/day → Possible outbreak 🚨

### Heart rate normally = 60–100 bpm → Patient shows 180 bpm → Emergency alert 🚑

### 🔸 Simple Python Example (Detecting Abnormal Patients per Day)

### import numpy as np

### import matplotlib.pyplot as plt

### # Example: patient count over 10 days

### patients = [120, 130, 125, 128, 135, 500, 132, 127, 140, 138]

### # Calculate mean and standard deviation

### mean = np.mean(patients)

### std = np.std(patients)

### # Define anomaly threshold (2 standard deviations from mean)

### threshold\_upper = mean + 2\*std

### threshold\_lower = mean - 2\*std

### # Detect anomalies

### anomalies = []

### for day, value in enumerate(patients, 1):

### if value > threshold\_upper or value < threshold\_lower:

### anomalies.append((day, value))

### print("🚨 Anomalies detected:", anomalies)

### # Plot

### plt.plot(range(1, 11), patients, marker="o", label="Patients")

### plt.axhline(threshold\_upper, color="r", linestyle="--", label="Upper Threshold")

### plt.axhline(threshold\_lower, color="g", linestyle="--", label="Lower Threshold")

### for day, value in anomalies:

### plt.scatter(day, value, color="red", s=100, label="Anomaly")

### plt.xlabel("Day")

### plt.ylabel("Number of Patients")

### plt.legend()

### plt.show()

### ✅ Result: This code will flag Day 6 (500 patients) as an anomaly and highlight it on the graph.

### 🔸 Role in Your Health AI Project

### Integrate anomaly detection into the chatbot → If a user reports symptoms that are not normal, trigger an alert.

### Hospital KPI monitoring → Detect sudden changes in bed occupancy, medicine usage, or patient feedback.

### Public health → Spot abnormal rise in disease cases across citizen queries.

### MULTIMODAL INPUT SUPPORT :

### Definition: Multimodal input means allowing users to interact with the AI using multiple types of inputs — not just text, but also voice, images, or documents.

### 🔸 Why It’s Important in Healthcare

### Accessibility → Helps patients who struggle with typing (elderly, disabled).

### Better Diagnosis → Doctors can upload X-rays, prescriptions, or lab reports for AI analysis.

### Natural Interaction → Patients can simply speak their symptoms.

### Faster Insights → AI can process both text + images together for improved accuracy.

### 🔸 Types of Multimodal Inputs for Health AI

### Text Input → User types symptoms or health questions.

### Voice Input → User speaks instead of typing (speech-to-text).

### Image Input → Upload medical images (X-ray, MRI, skin rash photos).

### Document Upload → Upload prescriptions, reports, or policies for summarization.

### 🔸 Example Use Cases

### Patient says: *“I have chest pain and cough”* (voice → text → AI suggests possible causes).

### Patient uploads an X-ray → AI detects possible pneumonia signs.

### User uploads insurance policy PDF → AI summarizes coverage.

### 🔸 Simple Python + Gradio Example (Multimodal Chatbot)

### import gradio as gr

### from transformers import pipeline

### # Example: Using a text summarizer for demonstration

### summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

### def multimodal\_health\_ai(text, image):

### response = ""

### 

### # Handle text input

### if text:

### response += f"📝 Text Analysis: {text}\n"

### if len(text.split()) > 15:

### summary = summarizer(text, max\_length=30, min\_length=10, do\_sample=False)[0]['summary\_text']

### response += f"🔹 Summarized: {summary}\n"

### 

### # Handle image input (just acknowledge for demo)

### if image is not None:

### response += "🖼 Image received. (AI can analyze medical images here)\n"

### 

### return response

### with gr.Blocks() as demo:

### gr.Markdown("## 🤖 Health AI – Multimodal Input Support")

### with gr.Row():

### text\_input = gr.Textbox(label="Enter symptoms or question")

### image\_input = gr.Image(type="filepath", label="Upload Medical Image")

### output = gr.Textbox(label="AI Response")

### submit = gr.Button("Analyze")

### submit.click(multimodal\_health\_ai, inputs=[text\_input, image\_input], outputs=output)

### demo.launch()

### ✅ Result:

### User can type symptoms or upload an image (like an X-ray).

### AI processes inputs and gives a response.

### Can later integrate speech-to-text for voice support.

### STREAMLIT OR GRADIO UI :

### **Best for:** Quick prototypes, chatbots, demos

### ✅ **Advantages:**

### Very **easy to build conversational interfaces** (chatbot, Q&A, feedback forms).

### Comes with built-in components for **chatbot, audio, image, video**.

### Can be embedded in **Colab notebooks** (perfect for your project).

### Simple code → just a few lines to create UI.

### ⚠️ **Limitations:**

### Less customizable compared to Streamlit.

### Mostly used for **AI model demos**, not full dashboards.

### **Example (Health AI chatbot in Gradio):**

### import gradio as gr

### def health\_ai(user\_input, history=[]):

### response = "Possible diagnosis for: " + user\_input

### history.append((user\_input, response))

### return history, history

### demo = gr.ChatInterface(fn=health\_ai, title="🏥 Health AI Assistant")

### demo.launch()

### 🔹 Streamlit UI

### **Best for:** Dashboards, multi-page apps, hospital analytics

### ✅ **Advantages:**

### Powerful for **data visualization (charts, KPIs, anomaly detection, forecasting)**.

### Supports **multipage apps** (e.g., "Chatbot", "Policy Summarizer", "Eco-Tip Generator", "KPI Dashboard").

### Better suited if you want a **professional hospital dashboard** feel.

### Can integrate with **Plotly, Matplotlib, Seaborn** for advanced visualizations.

### ⚠️ **Limitations:**

### More code required than Gradio.

### Not as straightforward for chatbot interfaces.

### **Example (Simple Health AI app in Streamlit):**

### import streamlit as st

### st.title("🏥 Health AI Assistant")

### # Text input

### user\_input = st.text\_input("Enter your symptoms or question:")

### if user\_input:

### response = "Possible diagnosis for: " + user\_input

### st.write("🤖 AI Response:", response)

### # Add a KPI chart

### import pandas as pd

### import matplotlib.pyplot as plt

### data = pd.DataFrame({"Month": [1,2,3,4], "Patients": [120,140,160,180]})

### st.line\_chart(data.set\_index("Month"))

### 🔹 Which Should You Choose?

### ✅ **If your project is chatbot-focused (patient Q&A, eco-tips, feedback loop)** → Use **Gradio**

### ✅ **If your project is analytics-focused (KPI forecasting, anomaly detection, dashboards)** → Use **Streamlit**

### ✅ **Best Case:** Use **both** →

### **Gradio** for conversational health assistant.

### **Streamlit** for hospital admin dashboard (KPIs, forecasting, resource planning)

### ARCHITECTURE

### FRONTEND(STREAM LIT) :

### 🏗 ****1. Frontend Layer – Streamlit UI****

### **Streamlit App** is the main **user interface**.

### Provides different modules/pages for:

### 🗨 **Chatbot Interface** (Patient Q&A)

### 📑 **Policy Summarizer** (Upload & simplify healthcare/insurance docs)

### 📊 **KPI Forecasting Dashboard** (graphs, charts, future predictions)

### 🚨 **Anomaly Detection** (highlight unusual patient flow, resources)

### 🌍 **Eco-Tip Generator** (daily health + environment tips)

### 🗣 **Citizen Feedback Loop** (collect feedback & ratings)

### 🧠 ****2. Backend Layer – AI Models & Logic****

### **IBM Granite Models (via Hugging Face)** → Handles natural language processing, chatbot responses, and summarization.

### **ML Models for Forecasting** → Predict patient inflow, KPI trends, resource demand.

### **Anomaly Detection Algorithms** → Detect outliers in patient/operational data.

### **Eco-Tip Generator** → Simple database/random tip generator (can be AI-driven).

### 🗄 ****3. Data Layer****

### **Patient & Hospital Data** → Historical data (for forecasting, anomaly detection).

### **User Queries & Feedback** → Stored in CSV, Google Sheets, or database.

### **Medical Guidelines & Policies** → For summarization and chatbot reference.

### ☁️ ****4. Deployment Layer****

### **Google Colab (Development)** → For training & testing AI models.

### **GitHub (Version Control)** → Store code & collaborate.

### **Streamlit Cloud / Hugging Face Spaces / Heroku / AWS** → Deployment for public use.

### 🔹 High-Level Flow (Step-by-Step)

### **User → Frontend (Streamlit):**

### Enters symptoms, uploads reports, or checks KPIs.

### **Frontend → Backend Models:**

### Sends input to IBM Granite (for Q&A, summarization).

### Sends data to forecasting/anomaly models.

### **Backend → Data Layer:**

### Retrieves patient history, hospital resources, policies.

### Stores user feedback.

### **Backend → Frontend:**

### Returns AI response, KPI charts, eco-tips, or alerts.

### **Frontend → User:**

### Displays output in interactive Streamlit dashboards.

### 🔹 Architecture Diagram (Textual Form)

### [User / Citizen / Doctor]

### |

### [Streamlit Frontend]

### ┌─────────────────┼───────────────────┐

### | | |

### [Chatbot UI] [Analytics Dashboard] [Feedback Form]

### | | |

### └──────→ [Backend AI/ML Models] ←────┘

### | | |

### [IBM Granite] | [Forecasting ML]

### | [Anomaly Detection]

### |

### [Data Layer]

### (Patient Data, Policies, Feedback, Hospital Records)

### |

### [Deployment (Cloud/Colab/GitHub)]

### BACKEND(FAST API) :

### 🖥 ****1. Frontend Layer (Streamlit)****

### User-facing **web app**

### Pages/Modules:

### 🗨 Chatbot (AI Q&A using IBM Granite)

### 📑 Policy Summarizer (upload & summarize docs)

### 📊 KPI Forecasting (charts & trends)

### 🚨 Anomaly Detection (alerts for unusual data)

### 🌍 Eco-Tip Generator (daily sustainability tip)

### 🗣 Feedback Form (citizen feedback loop)

### 👉 **Role:** Collects user input → Sends it to FastAPI backend → Displays results.

### ⚙️ ****2. Backend Layer (FastAPI)****

### FastAPI acts as the **bridge** between the frontend and AI/ML logic.

### **Endpoints (APIs):**

### /chat → For chatbot responses (IBM Granite model)

### /summarize → For policy/document summarization

### /forecast → For KPI/resource forecasting

### /anomaly → For anomaly detection in hospital/patient data

### /eco-tip → Generate random eco-friendly health tips

### /feedback → Save citizen feedback

### 👉 **Role:** Processes requests from Streamlit → Runs AI/ML models → Sends JSON response back.

### 🧠 ****3. AI/ML Model Layer****

### **IBM Granite Models (Hugging Face)** → Natural language understanding, chatbot, summarization.

### **ML Models (Scikit-learn / PyTorch)** → Forecasting, anomaly detection.

### **Utilities** → Eco-tip generator, feedback storage.

### 🗄 ****4. Data Layer****

### Hospital data (patients, beds, resources)

### User feedback (CSV/Database)

### Healthcare policies (for summarization)

### Logs for anomaly monitoring

### ☁️ ****5. Deployment Layer****

### **Backend (FastAPI)** → Deploy on **AWS/GCP/Heroku/Render**

### **Frontend (Streamlit)** → Deploy on **Streamlit Cloud or same server**

### Both connected via **REST API calls**

### 🔹 High-Level Flow

### [User]

### ↓

### [Streamlit Frontend] → calls → [FastAPI Backend]

### ↓ ↓

### [Modules: Chatbot, KPI] [AI/ML Models + IBM Granite]

### ↓ ↓

### Output ←---------------- [Processed Data + Insights]

### 🔹 Example Backend Code (FastAPI)

### from fastapi import FastAPI

### from pydantic import BaseModel

### import random

### app = FastAPI()

### # Example request schema

### class ChatRequest(BaseModel):

### message: str

### # Dummy Chatbot API

### @app.post("/chat")

### def chat(request: ChatRequest):

### user\_message = request.message

### response = f"AI Health Assistant Response for: {user\_message}"

### return {"user\_input": user\_message, "ai\_response": response}

### # Policy Summarizer API

### class PolicyRequest(BaseModel):

### text: str

### @app.post("/summarize")

### def summarize(request: PolicyRequest):

### text = request.text

### summary = "This policy mainly covers hospitalization up to 5 lakhs."

### return {"original": text, "summary": summary}

### # KPI Forecasting API

### @app.get("/forecast")

### def forecast():

### data = {"next\_month\_patients": 200, "expected\_beds\_needed": 50}

### return data

### # Anomaly Detection API

### @app.get("/anomaly")

### def anomaly():

### return {"alert": "Unusual patient spike detected on Day 6 (500 patients)"}

### # Eco Tip API

### eco\_tips = [

### "Turn off lights when not needed.",

### "Use reusable bottles instead of plastic.",

### "Segregate medical waste properly."

### ]

### @app.get("/eco-tip")

### def eco\_tip():

### return {"eco\_tip": random.choice(eco\_tips)}

### # Citizen Feedback API

### class Feedback(BaseModel):

### user: str

### rating: int

### comment: str

### @app.post("/feedback")

### def feedback(data: Feedback):

### return {"message": "✅ Feedback received", "user": data.user}

### ✅ **Result:**

### You now have a **FastAPI backend** with multiple endpoints.

### Streamlit frontend will call these APIs via requests or httpx to display results.

### **LLM INTEGRATION (IBM WATSONX GRANITE) :**

### IBM Watsonx Granite is a family of **foundation models** available on **Hugging Face** and **IBM Cloud (Watsonx.ai)**. You can use it for:

### 🗨 Chatbot (Q&A, health assistant)

### 📑 Policy summarization

### 📊 Intelligent recommendations

### 🔹 Architecture with Granite

### [User / Doctor / Citizen]

### ↓

### [Streamlit Frontend] → calls → [FastAPI Backend]

### ↓ ↓

### (UI Pages: Chatbot, [LLM: IBM Granite Models]

### Summarizer, KPI, etc.) (via Hugging Face or Watsonx API)

### ↓ ↓

### Output ←---------------- Processed Response

### 🔹 Integration Options

### **Using Hugging Face Granite Models (free/public)** Example: ibm-granite/granite-3.2-2b-instruct

### **Using IBM Watsonx API (enterprise, secure)** Requires an **IBM Cloud account** + **API key**

### 🔹 Example: FastAPI Backend with Granite (Hugging Face)

### from fastapi import FastAPI

### from pydantic import BaseModel

### from transformers import AutoTokenizer, AutoModelForCausalLM

### import torch

### # Load Granite model

### model\_name = "ibm-granite/granite-3.2-2b-instruct"

### tokenizer = AutoTokenizer.from\_pretrained(model\_name)

### model = AutoModelForCausalLM.from\_pretrained(model\_name)

### app = FastAPI()

### class ChatRequest(BaseModel):

### message: str

### @app.post("/chat")

### def chat(request: ChatRequest):

### inputs = tokenizer(request.message, return\_tensors="pt")

### outputs = model.generate(\*\*inputs, max\_length=150, temperature=0.7, top\_p=0.9)

### response = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

### return {"user\_input": request.message, "ai\_response": response}

### 🔹 Example: Using IBM Watsonx API (More Secure)

### import requests

### from fastapi import FastAPI

### from pydantic import BaseModel

### app = FastAPI()

### API\_KEY = "your\_ibm\_watsonx\_api\_key"

### PROJECT\_ID = "your\_project\_id"

### URL = "https://us-south.ml.cloud.ibm.com/ml/v1/text/generation?version=2023-05-29"

### class ChatRequest(BaseModel):

### message: str

### @app.post("/chat")

### def chat(request: ChatRequest):

### headers = {"Authorization": f"Bearer {API\_KEY}", "Content-Type": "application/json"}

### payload = {

### "model\_id": "granite-13b-instruct-v1",

### "input": request.message,

### "parameters": {"decoding\_method": "sample", "max\_new\_tokens": 200, "temperature": 0.7}

### }

### response = requests.post(URL, headers=headers, json=payload)

### result = response.json()

### return {"user\_input": request.message, "ai\_response": result["results"][0]["generated\_text"]}

### 🔹 Streamlit Frontend Example (Calling FastAPI Chat API)

### import streamlit as st

### import requests

### st.title("🏥 Health AI Assistant (IBM Granite LLM)")

### user\_input = st.text\_input("Enter your health question:")

### if st.button("Ask"):

### response = requests.post("http://127.0.0.1:8000/chat", json={"message": user\_input})

### if response.status\_code == 200:

### result = response.json()

### st.write("🤖 AI Response:", result["ai\_response"])

### else:

### st.error("API error. Please try again.")

### ✅ **Result:**

### Streamlit UI → Sends question to FastAPI backend

### FastAPI → Uses IBM Granite model (Hugging Face or Watsonx API)

### Returns AI response → Displayed in Streamlit

### VECTOR SEARCH(PINECONE) :

### 🔹 What is Vector Search (Pinecone)?

### **Traditional Search**: Matches keywords (not context).

### **Vector Search**: Converts text into **embeddings (vectors)** and finds **similar meaning** results.

### **Pinecone**: A managed vector database where you can store embeddings and run fast, semantic search queries.

### 🔹 Why Use in Health AI?

### **Medical Policy Retrieval** → User uploads policies, ask “Does this cover maternity?” → Pinecone finds relevant section.

### **Health Knowledge Base** → Store disease FAQs, symptoms, treatment guidelines.

### **Patient Q&A** → If LLM doesn’t know the answer, retrieve relevant info from Pinecone.

### **Citizen Feedback Analysis** → Store past feedback & retrieve similar ones.

### 🔹 Architecture with Pinecone

### [User Question]

### ↓

### [Streamlit Frontend] → [FastAPI Backend]

### ↓

### [LLM (Granite) + Pinecone Search]

### ↓

### [Relevant Policies / FAQs / Medical Info]

### ↓

### [Answer + Source Back to User]

### 🔹 Backend: FastAPI + Pinecone Example

### from fastapi import FastAPI

### from pydantic import BaseModel

### import pinecone

### from sentence\_transformers import SentenceTransformer

### # Initialize Pinecone

### pinecone.init(api\_key="YOUR\_PINECONE\_API\_KEY", environment="gcp-starter")

### index = pinecone.Index("health-ai-index")

### # Embedding model

### embed\_model = SentenceTransformer("all-MiniLM-L6-v2")

### app = FastAPI()

### # Request format

### class Query(BaseModel):

### question: str

### @app.post("/search")

### def vector\_search(query: Query):

### # Convert query to vector

### vector = embed\_model.encode(query.question).tolist()

### 

### # Query Pinecone

### results = index.query(vector=vector, top\_k=3, include\_metadata=True)

### 

### matches = []

### for match in results['matches']:

### matches.append({"text": match['metadata']['text'], "score": match['score']})

### 

### return {"query": query.question, "results": matches}

### 🔹 How to Populate Pinecone with Data

### # Example: Upload healthcare policies or FAQs

### documents = [

### {"id": "1", "text": "This policy covers hospitalization expenses up to 5 lakhs."},

### {"id": "2", "text": "Maternity benefits are covered after 3 years."},

### {"id": "3", "text": "Pre-existing diseases are covered after 2 years."}

### ]

### for doc in documents:

### vector = embed\_model.encode(doc["text"]).tolist()

### index.upsert([(doc["id"], vector, {"text": doc["text"]})])

### 🔹 Streamlit Frontend (Calling FastAPI Search API)

### import streamlit as st

### import requests

### st.title("🏥 Health AI – Policy Search with Pinecone")

### question = st.text\_input("Ask about your policy:")

### if st.button("Search"):

### response = requests.post("http://127.0.0.1:8000/search", json={"question": question})

### if response.status\_code == 200:

### results = response.json()["results"]

### for r in results:

### st.write(f"📄 {r['text']} (score: {round(r['score'],2)})")

### else:

### st.error("API Error. Try again.")

### ✅ **Result:**

### User asks: “Does this policy cover maternity?”

### Pinecone finds: “Maternity benefits are covered after 3 years.”

### Streamlit displays results instantly.

### **ML MODULES(FORECASTING AND ANOMALY DETECTION) :**

### 🔹 ML Module 1: Forecasting

### **Objective:** Predict **future healthcare KPIs** (patient inflow, bed usage, resource needs).

### Example Use Case

### Input: Past 6 months patient count

### Output: Predicted next 3 months patient count

### Sample Code (FastAPI Endpoint – Forecasting)

### from fastapi import FastAPI

### from pydantic import BaseModel

### import pandas as pd

### from sklearn.linear\_model import LinearRegression

### import numpy as np

### app = FastAPI()

### class ForecastRequest(BaseModel):

### months: list

### patients: list

### future\_months: int = 3

### @app.post("/forecast")

### def forecast(data: ForecastRequest):

### # Prepare dataframe

### df = pd.DataFrame({"Month": data.months, "Patients": data.patients})

### 

### # Train model

### X = np.array(df["Month"]).reshape(-1, 1)

### y = df["Patients"]

### model = LinearRegression()

### model.fit(X, y)

### 

### # Forecast

### future\_X = np.array(range(max(data.months)+1, max(data.months)+1+data.future\_months)).reshape(-1, 1)

### predictions = model.predict(future\_X).tolist()

### 

### return {"forecast\_months": future\_X.flatten().tolist(), "forecast\_patients": predictions}

### 🔹 ML Module 2: Anomaly Detection

### **Objective:** Detect **outliers in patient flow or hospital resources**.

### Example Use Case

### Normal: 100–150 patients/day

### If 400 patients appear suddenly → flag as anomaly 🚨

### Sample Code (FastAPI Endpoint – Anomaly Detection)

### class AnomalyRequest(BaseModel):

### values: list

### @app.post("/anomaly")

### def anomaly\_detection(data: AnomalyRequest):

### values = np.array(data.values)

### mean = np.mean(values)

### std = np.std(values)

### 

### upper = mean + 2\*std

### lower = mean - 2\*std

### 

### anomalies = []

### for i, val in enumerate(values, start=1):

### if val > upper or val < lower:

### anomalies.append({"day": i, "value": val})

### 

### return {

### "mean": mean,

### "std\_dev": std,

### "thresholds": {"upper": upper, "lower": lower},

### "anomalies": anomalies

### }

### 🔹 Streamlit Frontend Example

### import streamlit as st

### import requests

### import matplotlib.pyplot as plt

### st.title("📊 Health AI – Forecasting & Anomaly Detection")

### # Forecasting

### st.subheader("🔮 Patient Forecasting")

### months = [1,2,3,4,5,6]

### patients = [120,140,160,180,200,220]

### if st.button("Run Forecast"):

### response = requests.post("http://127.0.0.1:8000/forecast", json={

### "months": months, "patients": patients, "future\_months": 3

### })

### result = response.json()

### st.write(result)

### # Plot

### plt.plot(months, patients, marker="o", label="Actual")

### plt.plot(result["forecast\_months"], result["forecast\_patients"], marker="x", linestyle="--", label="Forecast")

### st.pyplot(plt)

### # Anomaly Detection

### st.subheader("🚨 Anomaly Detection")

### values = [120, 130, 125, 500, 140, 135]

### if st.button("Run Anomaly Detection"):

### response = requests.post("http://127.0.0.1:8000/anomaly", json={"values": values})

### result = response.json()

### st.write(result)

### 🔹 How These Fit in Your Architecture

### **FastAPI Backend**: Hosts ML models (/forecast, /anomaly).

### **Streamlit Frontend**: Calls API → Displays results as **charts + alerts**.

### **Granite LLM (Optional)**: Explains predictions/anomalies in **natural language**.

### SETUP INSTRUCTION

### PREREQUISITES :

### 🔹 1. Prerequisites

### Before starting, make sure you have the following:

### 🖥 System Requirements

### OS: Windows / Linux / macOS

### RAM: Minimum **8 GB** (16 GB recommended for ML models)

### GPU: (Optional) NVIDIA GPU for faster training/inference

### Python: **3.9 or later**

### 📦 Software & Libraries

### **Python** → [Download here](https://www.python.org/downloads/)

### **FastAPI** → Backend framework

### **Uvicorn** → Run FastAPI server

### **Streamlit** → Frontend dashboard & chatbot UI

### **Transformers (Hugging Face)** → Load IBM Granite LLM

### **Torch** → Required for AI model inference

### **Scikit-learn** → Forecasting, anomaly detection

### **Matplotlib / Plotly** → Visualizations

### **Pinecone Client** → For vector search

### **Requests / HTTPX** → API communication

### 🔑 Accounts & API Keys

### **Hugging Face Account** → Access IBM Granite models

### **IBM Watsonx.ai (Optional)** → If using enterprise Granite API

### **Pinecone Account** → Vector search setup

### **GitHub Account** → For version control & project upload

### 🔹 2. Environment Setup

### Step 1: Clone Repository

### git clone https://github.com/your-username/health-ai-project.git

### cd health-ai-project

### Step 2: Create Virtual Environment

### python -m venv venv

### source venv/bin/activate # On Linux/Mac

### venv\Scripts\activate # On Windows

### Step 3: Install Dependencies

### pip install -r requirements.txt

### 📌 Example requirements.txt

### fastapi

### uvicorn

### streamlit

### transformers

### torch

### scikit-learn

### matplotlib

### pandas

### pinecone-client

### sentence-transformers

### requests

### 🔹 3. Running the Project

### Step 1: Start FastAPI Backend

### uvicorn backend.main:app --reload --port 8000

### Step 2: Start Streamlit Frontend

### streamlit run frontend/app.py

### Step 3: Access in Browser

### Backend API → http://127.0.0.1:8000/docs

### Frontend UI → http://localhost:8501

### 🔹 4. Project Workflow

### **Streamlit UI (Frontend)**

### User asks questions / uploads documents / views dashboards

### **FastAPI Backend**

### Routes request to correct module (Chatbot, Forecasting, Anomaly, Pinecone Search)

### **ML/LLM Models**

### IBM Granite (Chatbot & Summarizer)

### Forecasting & Anomaly Detection Models

### Pinecone (Vector Search)

### **Data Storage**

### Feedback stored in CSV/DB

### Policies indexed in Pinecone

### INSTALLATION PROCESS :

### 1. Clone the Repository

### git clone https://github.com/your-username/health-ai-project.git

### cd health-ai-project

### 🔹 2. Create Virtual Environment

### python -m venv venv

### Activate environment:

### **Windows**

### venv\Scripts\activate

### **Linux/Mac**

### source venv/bin/activate

### 🔹 3. Install Dependencies

### Install all required libraries from requirements.txt:

### pip install -r requirements.txt

### 📌 Example requirements.txt

### fastapi

### uvicorn

### streamlit

### transformers

### torch

### scikit-learn

### matplotlib

### pandas

### pinecone-client

### sentence-transformers

### requests

### 🔹 4. Setup API Keys

### Some modules need API keys:

### **Hugging Face Token** (for IBM Granite model) → Get here

### **Pinecone API Key** → Get here

### **IBM Watsonx API Key** (if using IBM Cloud Granite) → Get here

### Create a .env file in your project root:

### HUGGINGFACE\_TOKEN=your\_hf\_token

### PINECONE\_API\_KEY=your\_pinecone\_key

### PINECONE\_ENV=gcp-starter

### IBM\_WATSONX\_API\_KEY=your\_ibm\_key

### 🔹 5. Start Backend (FastAPI)

### Navigate to backend folder:

### cd backend

### uvicorn main:app --reload --port 8000

### Backend will run at: 👉 http://127.0.0.1:8000/docs (interactive API docs)

### 🔹 6. Start Frontend (Streamlit)

### Open another terminal:

### cd frontend

### streamlit run app.py

### Frontend will run at: 👉 http://localhost:8501

### 🔹 7. Test Features

### ✅ **Chatbot (IBM Granite LLM)** → Ask health questions ✅ **Policy Summarization** → Upload long policies, get summary ✅ **KPI Forecasting** → View patient inflow predictions ✅ **Anomaly Detection** → Spot unusual spikes in data ✅ **Eco-Tip Generator** → Get random eco tips ✅ **Citizen Feedback Loop** → Submit feedback ✅ **Vector Search (Pinecone)** → Ask queries on stored policies

### 🔹 8. (Optional) Docker Setup

### For easier deployment, build & run using Docker:

### docker build -t health-ai .

### docker run -p 8000:8000 -p 8501:8501 health-ai

### ✅ Now your **Health AI system is fully installed and ready to run** 🎉

### FOLDER STRUCTURE :

### health-ai-project/

### │── backend/ # FastAPI backend

### │ ├── main.py # Entry point (FastAPI app)

### │ ├── routers/ # API endpoints

### │ │ ├── chatbot.py # LLM (Granite) chatbot

### │ │ ├── summarizer.py # Policy summarization

### │ │ ├── forecasting.py # KPI forecasting module

### │ │ ├── anomaly.py # Anomaly detection module

### │ │ ├── eco\_tip.py # Eco-tip generator

### │ │ ├── feedback.py # Citizen feedback loop

### │ │ └── search.py # Pinecone vector search

### │ ├── models/ # ML/LLM model integrations

### │ │ ├── granite\_llm.py # IBM Granite model wrapper

### │ │ ├── forecast\_model.py # Forecasting logic

### │ │ ├── anomaly\_model.py # Anomaly detection logic

### │ │ └── embeddings.py # SentenceTransformer for Pinecone

### │ ├── utils/ # Helper functions

### │ │ ├── db.py # Save feedback (CSV/DB)

### │ │ └── config.py # API keys, environment variables

### │ └── requirements.txt # Backend dependencies

### │

### │── frontend/ # Streamlit frontend

### │ ├── app.py # Main Streamlit dashboard

### │ ├── pages/ # Multi-page Streamlit UI

### │ │ ├── 1\_Chatbot.py # Chatbot page

### │ │ ├── 2\_Policy\_Summarizer.py# Summarization page

### │ │ ├── 3\_KPI\_Forecasting.py # Forecasting dashboard

### │ │ ├── 4\_Anomaly\_Detection.py# Anomaly detection dashboard

### │ │ ├── 5\_Eco\_Tips.py # Eco-tip generator page

### │ │ └── 6\_Feedback.py # Citizen feedback form

### │ ├── components/ # Reusable UI components

### │ │ ├── charts.py # Graph plotting functions

### │ │ └── layouts.py # UI layouts

### │ └── requirements.txt # Frontend dependencies

### │

### │── data/ # Data storage

### │ ├── policies/ # Example policy docs (txt/pdf)

### │ ├── hospital\_data.csv # Patient/KPI dataset

### │ └── feedback.csv # Stored citizen feedback

### │

### │── notebooks/ # Jupyter/Colab notebooks

### │ ├── model\_training.ipynb # Forecasting & anomaly ML training

### │ └── pinecone\_indexing.ipynb # Upload docs to Pinecone

### │

### │── tests/ # Unit and integration tests

### │ ├── test\_chatbot.py

### │ ├── test\_forecasting.py

### │ ├── test\_anomaly.py

### │ └── test\_feedback.py

### │

### │── .env # API keys (HuggingFace, Pinecone, IBM)

### │── .gitignore # Ignore venv, \_\_pycache\_\_, data files

### │── dockerfile # For containerized deployment

### │── docker-compose.yml # Optional (frontend+backend)

### │── README.md # Project documentation

### 🔹 Explanation

### backend/ → FastAPI backend (APIs for chatbot, summarization, forecasting, anomaly detection, vector search, eco-tips, feedback).

### frontend/ → Streamlit frontend (multi-page app for patients, citizens, and admins).

### data/ → Store healthcare data, feedback, and sample policies.

### notebooks/ → For model experimentation & Pinecone indexing.

### tests/ → Unit tests for backend + ML modules.

### .env → Store API keys securely.

### dockerfile → For deployment in Docker containers.

### RUNNING THE APPLICATION :

### 🔹 1. Start the Backend (FastAPI)

### Step 1: Navigate to backend folder

### cd backend

### Step 2: Activate virtual environment

### **Windows**

### venv\Scripts\activate

### **Linux/Mac**

### source venv/bin/activate

### Step 3: Run FastAPI with Uvicorn

### uvicorn main:app --reload --port 8000

### ✅ Backend API will start at: 👉 http://127.0.0.1:8000/docs (Swagger UI for testing APIs)

### 🔹 2. Start the Frontend (Streamlit)

### Step 1: Open a new terminal

### cd frontend

### Step 2: Activate virtual environment

### (same as backend, if separate env then activate frontend’s venv)

### Step 3: Run Streamlit

### streamlit run app.py

### ✅ Frontend UI will start at: 👉 http://localhost:8501

### 🔹 3. Application Workflow

### **User opens Streamlit UI** → selects module (Chatbot, Policy Summarizer, Forecasting, Anomaly Detection, Eco-Tips, Feedback, Vector Search).

### **Streamlit sends request** → FastAPI backend (http://127.0.0.1:8000/...).

### **Backend processes input** → using:

### IBM Granite (LLM) for chatbot & summarization

### Forecasting ML model

### Anomaly detection ML model

### Pinecone for vector search

### **Response returned to Streamlit** → displayed as **charts, text, or alerts**.

### 🔹 4. Verifying the Setup

### Open **FastAPI docs** → test /chat, /summarize, /forecast, /anomaly, /eco-tip, /feedback, /search.

### Open **Streamlit app** → interact with modules:

### 🗨 Chatbot → Ask “What are symptoms of diabetes?”

### 📑 Policy Summarizer → Upload long policy text

### 📊 Forecasting → View patient trend predictions

### 🚨 Anomaly → Detect spikes in patient flow

### 🌍 Eco-Tip → Generate eco-friendly tips

### 🗣 Feedback → Submit a review

### 🔍 Vector Search → Query policies with Pinecone

### 🔹 5. Optional: Run in Docker

### If you created a dockerfile + docker-compose.yml:

### docker-compose up --build

### 👉 This will run **backend (port 8000)** and **frontend (port 8501)** in containers.

### ✅ Now your **Health AI application is fully running** with both **backend (FastAPI)** and **frontend (Streamlit)**.

### API DOCUMENTATION :

### Base URL (local):

### http://127.0.0.1:8000

### Interactive Swagger UI:

### http://127.0.0.1:8000/docs

### 🔹 Endpoints

### 1. 🗨 Chatbot (IBM Granite LLM)

### **POST** /chat

### **Description:** Generates a conversational response using IBM Granite model.

### Request (JSON):

### {

### "message": "What are the symptoms of diabetes?"

### }

### Response (JSON):

### {

### "user\_input": "What are the symptoms of diabetes?",

### "ai\_response": "Common symptoms include frequent urination, increased thirst, and fatigue."

### }

### 2. 📑 Policy Summarization

### **POST** /summarize

### **Description:** Summarizes long healthcare/insurance policies.

### Request (JSON):

### {

### "text": "This policy covers hospitalization expenses up to 5 lakhs. Pre-existing conditions after 2 years. Maternity after 3 years."

### }

### Response (JSON):

### {

### "original": "This policy covers hospitalization expenses up to 5 lakhs...",

### "summary": "Covers hospitalization up to ₹5 lakhs, pre-existing after 2 years, maternity after 3 years."

### }

### 3. 📊 KPI Forecasting

### **POST** /forecast

### **Description:** Predicts future patient inflow or hospital KPIs.

### Request (JSON):

### {

### "months": [1,2,3,4,5,6],

### "patients": [120,140,160,180,200,220],

### "future\_months": 3

### }

### Response (JSON):

### {

### "forecast\_months": [7,8,9],

### "forecast\_patients": [240, 260, 280]

### }

### 4. 🚨 Anomaly Detection

### **POST** /anomaly

### **Description:** Detects abnormal spikes or drops in hospital data.

### Request (JSON):

### {

### "values": [120, 130, 125, 500, 140, 135]

### }

### Response (JSON):

### {

### "mean": 191.6,

### "std\_dev": 126.4,

### "thresholds": { "upper": 444.4, "lower": -61.2 },

### "anomalies": [{ "day": 4, "value": 500 }]

### }

### 5. 🌍 Eco-Tip Generator

### **GET** /eco-tip

### **Description:** Returns a random eco-friendly health tip.

### Response (JSON):

### {

### "eco\_tip": "Use reusable water bottles instead of plastic ones."

### }

### 6. 🗣 Citizen Feedback Loop

### **POST** /feedback

### **Description:** Stores citizen feedback for system improvement.

### Request (JSON):

### {

### "user": "Priya",

### "rating": 5,

### "comment": "Very useful health assistant!"

### }

### Response (JSON):

### {

### "message": "✅ Feedback received",

### "user": "Priya"

### }

### 7. 🔍 Vector Search (Pinecone)

### **POST** /search

### **Description:** Searches medical policies/documents using Pinecone semantic search.

### Request (JSON):

### {

### "question": "Does this policy cover maternity?"

### }

### Response (JSON):

### {

### "query": "Does this policy cover maternity?",

### "results": [

### { "text": "Maternity benefits are covered after 3 years.", "score": 0.89 },

### { "text": "This policy covers hospitalization up to 5 lakhs.", "score": 0.75 }

### ]

### }

### 🔹 Authentication

### Currently: **No authentication** (local dev). Future: Add **API Key / JWT tokens** for secure hospital deployment.

### 🔹 Error Handling

### **400 Bad Request** → Invalid input format

### **500 Internal Server Error** → Model/Server issue

### Example:

### {

### "detail": "Invalid input data format"

### }

### ✅ With this API documentation:

### Developers can test via **Swagger UI** or Postman.

### Streamlit frontend will call these endpoints for chatbot, forecasting, anomaly detection, eco-tips, summarization, feedback, and vector search.

### AUTHENTICATION :

### Authentication is the process of verifying the identity of a user, device, or system before granting access to resources or services. Essentially, it answers the question:

### “Are you really who you say you are?”

### Types of Authentication

### Password-based Authentication (Something you know)

### User provides a username and password.

### Example: Logging into Gmail or Facebook.

### Biometric Authentication (Something you are)

### Uses physical traits for verification.

### Example: Fingerprint, face recognition, retina scan.

### Token-based / Two-Factor Authentication (2FA) (Something you have)

### Combines multiple factors for security.

### Example: SMS OTP, authentication apps (Google Authenticator), hardware tokens.

### Certificate-based Authentication

### Uses digital certificates issued by trusted authorities.

### Example: HTTPS websites or VPN connections.

### Multi-Factor Authentication (MFA)

### Combines two or more types above for stronger security.

### Example: Password + OTP + Fingerprint.

### 

### USER INTERFACE (UI) :

### DEFINITION :

### A User Interface (UI) is the space where a user interacts with a computer, software, or application. It’s everything you see and use on a screen to perform tasks, including buttons, menus, icons, text, and layouts.

### In simple terms: UI is how the user sees and controls the system.

### Types of User Interface

### Graphical User Interface (GUI)

### Uses visual elements like windows, icons, buttons, and menus.

### Example: Windows OS, macOS, mobile apps.

### Command-Line Interface (CLI)

### Text-based interface where users type commands.

### Example: Terminal in Linux or Command Prompt in Windows.

### Menu-Driven Interface

### Users navigate through menus to perform tasks.

### Example: ATM machines, digital kiosks.

### Touch User Interface

### Interaction through touch gestures.

### Example: Smartphones, tablets.

### Voice User Interface (VUI)

### Interaction through voice commands.

### Example: Alexa, Siri, Google Assistant.

### Key Principles of a Good UI

### Clarity – Easy to understand and navigate.

### Consistency – Same design and behavior throughout the application.

### Feedback – The system responds to user actions (e.g., button click animation).

### Efficiency – Enables users to complete tasks quickly.

### Accessibility – Usable by everyone, including people with disabilities.

### **TESTING :**

### Health testing is the process of examining a person’s physical or mental condition to detect diseases, assess health risks, or monitor wellness.

### In modern healthcare, AI and technology are increasingly used to assist in testing, diagnosis, and prediction.

### Types of Health Testing

### Diagnostic Tests

### Determine if a person has a specific disease.

### Example: Blood tests for diabetes, COVID-19 PCR tests, X-rays.

### Screening Tests

### Detect potential health problems before symptoms appear.

### Example: Mammograms for breast cancer, blood pressure screening.

### Monitoring Tests

### Track ongoing health conditions.

### Example: Glucose monitoring for diabetics, heart rate monitoring.

### Genetic/Genomic Tests

### Analyze DNA to find risks for inherited diseases.

### Example: BRCA gene test for breast cancer risk.

### 

### Health Testing in AI (Health AI)

### AI technologies are used to:

### Predict diseases – e.g., AI analyzes imaging scans to detect tumors.

### Assist diagnosis – e.g., AI helps doctors interpret X-rays or ECGs.

### Monitor patients remotely – Wearable devices track vital signs and alert doctors.

### Personalized treatment – AI recommends medications based on patient data.

### Benefits of Health Testing

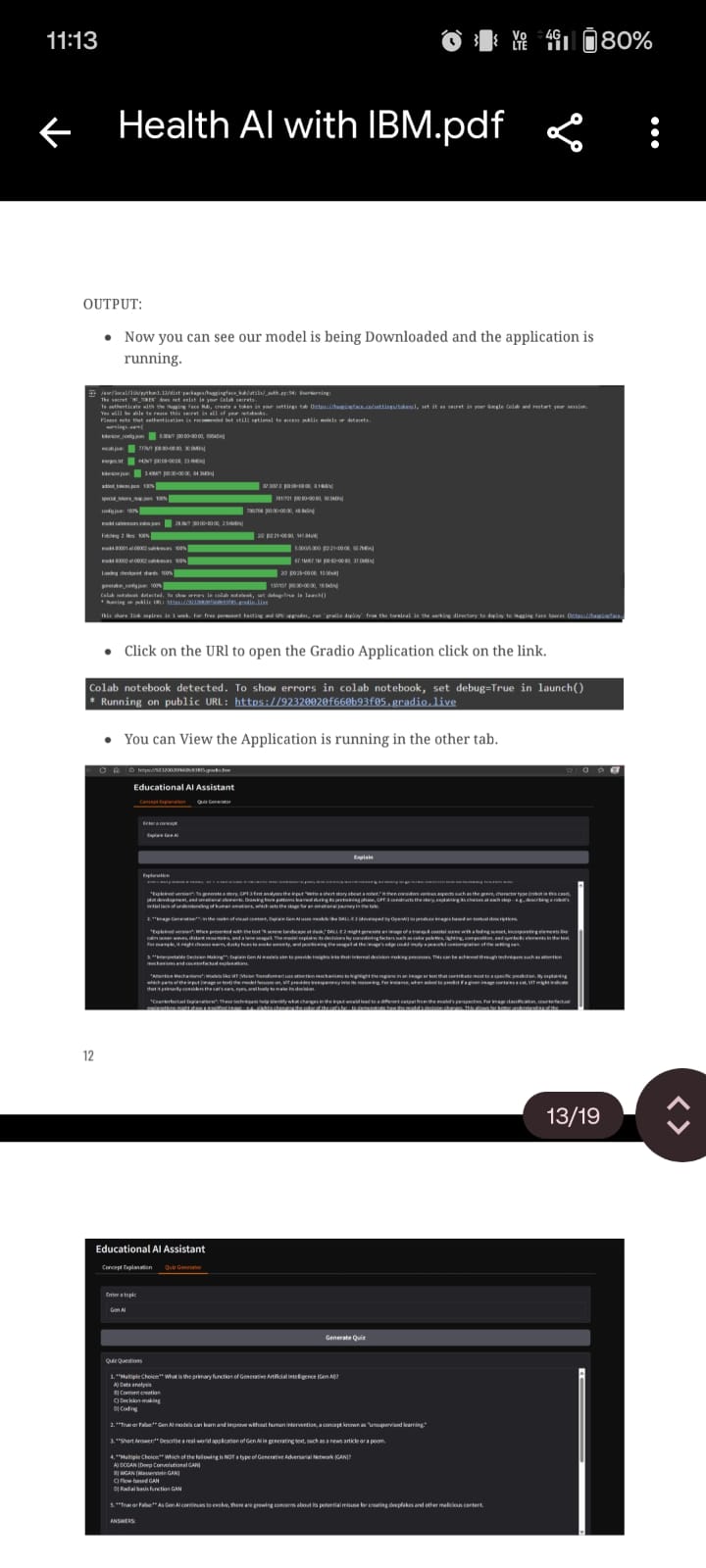
### Early detection of diseases.

### Accurate diagnosis.

### Continuous monitoring for better treatment.

### Reduced human errors with AI assistance.



****

KNOWN ISSUES :

### Health AI refers to using artificial intelligence for diagnosis, treatment recommendations, patient monitoring, and healthcare management. While it has huge potential, there are several challenges and issues:

### 1. Data Quality and Availability

### AI models require large amounts of high-quality data.

### In healthcare, data may be incomplete, inconsistent, or biased.

### Example: Electronic health records (EHRs) may have missing patient histories.

### 2. Bias and Fairness

### AI models can inherit biases present in the training data.

### Example: An AI diagnostic tool may perform worse for minority populations if underrepresented in the data.

### 3. Explainability and Transparency

### Many AI models, especially deep learning, are “black boxes”.

### Doctors may not understand how AI arrives at a decision.

### This reduces trust and makes accountability difficult.

### 4. Privacy and Security

### Patient data is highly sensitive.

### Risks include data breaches, hacking, and misuse of personal health information.

### Compliance with regulations like HIPAA is critical but challenging.

### 5. Regulatory and Legal Challenges

### Healthcare AI is subject to strict regulations.

### Liability issues: Who is responsible if AI makes a wrong diagnosis or treatment suggestion?

### 6. Integration with Clinical Workflows

### AI tools may not integrate seamlessly with existing hospital systems.

### Can lead to workflow disruptions rather than improvements.

### 7. Reliability and Generalization

### AI trained on one hospital’s data may not perform well on another hospital’s data.

### Example: Diagnostic models may fail with different imaging equipment or populations.

### 8. Ethical Concerns

### Decisions made by AI can have serious consequences on patient health.

### Example: Prioritizing treatment based on AI predictions may lead to ethical dilemmas.

### FUTURE ENHANCEMENT :

### Future enhancement refers to planned improvements or upgrades to a system, product, or technology to make it more efficient, reliable, or user-friendly.

### In simple terms: It’s about making things better in the future.

### Future Enhancements in Health AI

### Better Data Quality and Access

### Improved collection and sharing of high-quality, standardized healthcare data.

### Use of federated learning to train AI without compromising patient privacy.

### Explainable AI (XAI)

### AI models that show how they make decisions.

### Helps doctors trust AI recommendations and improves accountability.

### Integration with Wearable Devices

### Real-time monitoring via smartwatches, fitness trackers, and IoT devices.

### Enables continuous health tracking and early detection.

### Personalized Medicine

### AI analyzing genetic, lifestyle, and clinical data to suggest personalized treatment plans.

### Example: Tailoring cancer treatment based on patient-specific data.

### Enhanced Security and Privacy

### Advanced encryption, anonymization, and blockchain for secure health data management.

### AI-Assisted Surgery and Robotics

### Future surgical robots guided by AI for higher precision, fewer errors, and faster recovery.

### Global Accessibility

### AI tools can reach rural and underserved areas, providing expert healthcare remotely.

### Continuous Learning AI

### AI systems that update and learn continuously from new patient data, improving accuracy over time.

### CONCLUSION :

### Health AI is revolutionizing healthcare by leveraging artificial intelligence to improve diagnosis, treatment, patient monitoring, and overall healthcare management. It offers early disease detection, personalized treatment, and better patient outcomes.

### However, Health AI faces challenges such as data quality issues, bias, privacy concerns, regulatory hurdles, and integration difficulties. Addressing these challenges is essential for safe, reliable, and ethical use of AI in healthcare.

### The future of Health AI is promising, with enhancements like explainable AI, wearable device integration, personalized medicine, AI-assisted surgery, and global accessibility. With continuous research, development, and ethical practices, Health AI has the potential to transform healthcare systems worldwide, making healthcare more efficient, accurate, and accessible to everyone.