HealthAI
Intelligent Healthcare Assistant

1.Introduction:

HealthAI is a medical AI assistant built using **IBM Granite LLM**, **Gradio**, and **PyTorch**.
It provides two primary features:

- **Disease Prediction** – Suggests possible conditions and general recommendations based on user-reported symptoms.
- **Treatment Plan** – Generates personalized treatment guidelines based on patient details.

Disclaimer: This application is for **informational purposes only** and should not replace professional medical consultation.

- Project Tittle :HealthAI
- Team Leader :Harishsarathi
- Team Member : Kamalakannan
- Team Member : Lingeshwaran
- Team Member : Logeshwaran

2.Project Overview:

Purpose:

HealthAI is designed to provide **AI-powered medical assistance** by analyzing user-reported symptoms and generating **personalized treatment suggestions**. It leverages **IBM Granite LLM** for natural language understanding and **Gradio** for an interactive interface, making healthcare information more **accessible and user-friendly**.

Note: The tool is **not a substitute for professional medical advice**. It is meant for **informational and educational purposes only**.

Key Features:

- **Disease Prediction:**
    - Analyzes symptoms and suggests possible medical conditions
    - Provides general recommendations for care
- **Personalized Treatment Plans:**
    - Generates treatment suggestions based on patient profile (age, gender, medical history)
    - Includes home remedies and general medication guidelines
- **User-Friendly Web Interface:**
    - Built with **Gradio Tabs** for smooth navigation
    - Simple text-based input and output fields

Technology Stack:

- **AI Model:** IBM Granite (granite-3.2-2b-instruct) via Hugging Face
- **Frameworks:** PyTorch, Transformers, Gradio
- **Interface:** Web-based (accessible via browser)
- **Device Support:** CPU & GPU

Target Users:

- Individuals seeking **basic medical guidance** before consulting a doctor
- Healthcare enthusiasts exploring **AI in medicine**
- Developers and researchers experimenting with **medical AI applications**

Benefits:

- **Instant suggestions** for common health issues
- **Personalized recommendations** tailored to patient profile
- **Accessible anywhere** via a simple browser interface
- Can serve as a **foundation** for more advanced healthcare systems

3.Project Workflow:

Activity-1: Exploring Naan Mudhalavan Smart Interz Portal.

Activity-2: Choosing a IBM Granite Model From Hugging Face.

Activity-3: Running Application In Google Colab.

Activity-4: Upload your Project in Github.

4.Architecture:

User Layer (Frontend – Gradio UI):

- Users interact via a **web interface** built with Gradio.
- Two main tabs:
    1. **Disease Prediction** → Users enter symptoms.
    2. **Treatment Plans** → Users enter condition, age, gender, and medical history.
- Inputs are collected through textboxes, dropdowns, and number fields.

Application Layer (Logic & Functions in Python):

- **Functions**:
    - disease_prediction(symptoms)
        - Builds a medical prompt.
        - Calls generate_response().
    - treatment_plan(condition, age, gender, medical_history)
        - Builds a personalized treatment prompt.
        - Calls generate_response().
    - generate_response(prompt, max_length)
        - Tokenizes input.
        - Sends it to the model for inference.
        - Decodes and returns output.

AI Layer (Model & NLP Processing):

- **IBM Granite 3.2 2B Instruct** (Hugging Face model).
- Handles:
    - Natural language understanding (symptoms, conditions).

- Natural language generation (recommendations, plans).
- Runs on:
  - **GPU (float16)** if available → faster, optimized.
  - **CPU (float32)** otherwise.

Infrastructure Layer:

- **PyTorch** → Executes the deep learning model.
- **Transformers** → Loads model & tokenizer from Hugging Face.
- **Gradio** → Hosts interactive UI & connects users with AI functions.

Flow of Execution:

1. **User Input** (symptoms or patient info) → entered in Gradio UI.
2. **Prompt Builder** (inside disease_prediction / treatment_plan) → prepares context.
3. **Tokenizer** → Converts text to tokens.
4. **AI Model (Granite)** → Generates output tokens.
5. **Tokenizer Decode** → Converts tokens back to text.
6. **Response Returned** → Displayed in Gradio textbox.

Architecture Diagram (Textual)

User (Web Browser)

▼

Gradio Fronten

- Disease Prediction

- Treatment Plans

▼

Application Logic

- disease_prediction

- treatment_plan

- generate_response

▼

Hugging Face Model

IBM Granite 3.2 2B

▼

Tokenizer + Decoder

(Transformers)

▼

AI Generated Text

(Displayed in UI)

5.Setup Instructions:

Progress-1: Exploring Naan Mudhalava Smart Interz Portal.
Search for “Naan Mudhalavan Smart Interz” Portal in any Browser.

Progress-2: Choose a IBM Granite model From Hugging Face.
Search for “Hugging face” in any browser.

Activity-3: Running Application in Google Collab.
Search for “Google collab” in any browser.

Choose a IBM Granite model From Hugging Face.

- Search for “Hugging face” in any browser.
- Then click on the first link (Hugging Face), then click on signup and create your own account in Hugging Face. Then search for “IBM-Granite models” and choose any model.

- Here for this project we are using “granite-3.2-2b-instruct” which is compatible fast and light weight.
- Now we will start building our project in Google collab

Running Application in Google Collab.

- Search for “Google collab” in any browser.
- Click on the first link (Google Colab), then click on “Files” and then “Open Notebook”.
- Click on “New Notebook”
- Change the title of the notebook “Untitled” to “Health AI”.
- Then click on “Runtime”, then go to “Change Runtime Type”.
- Choose “T4 GPU” and click on “Save”
- Then run this command in the first cell “!pip install transformers torch gradio -q”.
- To install the required libraries to run our application.
- Then run the rest of the code in the next cell.

6.Folder Structure:

```
project-root/
│
├── healthai.py         # Main application script
│                       # - Loads model & tokenizer
│                       # - Defines disease prediction & treatment functions
│                       # - Creates Gradio UI and launches app
│
├── requirements.txt    # (Optional) List of Python dependencies
│
└── README.md           # (Optional) Documentation for setup & usage
```

Explanation:

healthai.py: The only main file; contains all app logic, AI inference, and UI design.

requirements.txt: Can include torch, transformers, and gradio for easy setup.

README.md: For instructions, usage guide, and disclaimers.

7.Running the Application:

Open a terminal and run:

python healthai.py

The Gradio app will start and give you a shareable link.

Open the link in a browser.

Enter symptoms or medical details to get AI-generated health suggestions.

For information only – always consult a doctor.

1. Backend (FastAPI) – backend.py:

Purpose: Runs a server that loads your Hugging Face model once, processes user input, and returns AI-generated responses.

Key Components:

generate_response(): Same function you had, used to create responses.

/predict_disease: API endpoint that accepts symptoms and returns possible conditions.

/treatment_plan: API endpoint that accepts condition, age, gender, and history, and returns a treatment plan.

Pydantic Models: SymptomsRequest and TreatmentRequest make sure input data is validated before processing.

Benefit: The model is loaded once and stays in memory, making it much faster for multiple users.

2. Frontend (Streamlit) – frontend.py:

Purpose: A modern UI for your app where users can type symptoms or medical details, and see AI results.

Features:

Uses st.tabs() to create two sections:

Disease Prediction

Treatment Plan

Sends user inputs to the backend via requests.post().

Displays AI responses clearly.

Benefit: Streamlit is visually more appealing and customizable than Gradio.

3. How They Work Together:

You run the backend using:

uvicorn backend:app --reload

This starts a FastAPI server at http://127.0.0.1:8000.

You can visit /docs to see Swagger UI for testing endpoints.

You run the frontend using:

streamlit run frontend.py

This launches the web UI at http://localhost:8501.

The frontend sends HTTP requests to the backend, gets responses, and displays them in the browser.

8. API Documentation:

The healthai.py application exposes two primary functionalities through a **Gradio-based API**: **Disease Prediction** and **Treatment Planning**. The API works by accepting text or structured inputs, processing them with the IBM Granite AI model, and returning informative responses.

1. **Disease Prediction Endpoint:**
   - **Input**: symptoms *(string, comma-separated)*

- **Output**: Text describing possible conditions and general medication suggestions. Always contains a disclaimer urging users to consult a healthcare professional.
- **Example**: Sending "fever, headache, cough" returns possible flu-related conditions and supportive care notes.

2. **Treatment Plan Endpoint:**
   - **Input**: JSON-like structure containing:
     - condition *(string)* – medical issue,
     - age *(integer)* – patient's age,
     - gender *(Male/Female/Other)*,
     - medical_history *(string)* – relevant background.
   - **Output**: Personalized plan including home remedies, lifestyle tips, and general medication guidelines. Includes a mandatory disclaimer.

Both endpoints are **accessible via the Gradio web UI** or programmatically by sending requests to the underlying Python functions. Responses are generated with safe temperature-controlled sampling and capped at 1200 tokens to maintain relevance. Security should be enforced with **authentication and HTTPS** if exposed publicly.

9. Authentication:

Disable public sharing.
Change app.launch(share=True) → share=False. The share tunnel is public and unsafe for PHI.

Add simple Gradio auth (quick guard).
Use environment variables, not hardcoded secrets:

```
import os
GRADIO_USER = os.environ.get("GRADIO_USER","admin")
GRADIO_PASS = os.environ.get("GRADIO_PASS")
if not GRADIO_PASS:
   raise RuntimeError("Set GRADIO_PASS")
app.launch(share=False, auth=(GRADIO_USER, GRADIO_PASS),
        server_name="0.0.0.0", server_port=int(os.environ.get("PORT",7860)))
```

This is fine for internal use or small teams.

Use HTTPS / TLS.
Terminate TLS at a reverse proxy (NGINX/Caddy) or cloud load balancer — never run plaintext HTTP on the public internet.

Prefer industry auth for production.
Replace Gradio's basic auth with a proper auth layer (FastAPI + OAuth2 / JWT, or integrate with single-sign-on like Google/Okta/Auth0). Mount Gradio inside a FastAPI app and validate tokens before serving the UI/API.

Store secrets securely.
Use environment variables, a secrets manager (AWS Secrets Manager / HashiCorp Vault), and rotate credentials.

Protect credentials & users.
If you manage users, store **hashed** passwords (bcrypt/argon2), enforce strong passwords, and implement account lockouts & 2FA where possible.

Add rate limiting & logging.
Rate limit endpoints to reduce abuse. Log authentication events (not raw PHI) and monitor for suspicious access.

Data minimization & compliance.
Because this handles health data, avoid storing PHI unless necessary. If you must, encrypt data at rest and follow applicable regulations (HIPAA, GDPR)—use Business Associate Agreements when needed.

Test & audit.
Pen-test the deployment, verify TLS configuration (HSTS), and ensure CORS is locked to allowed origins.

Operational hygiene.
Keep dependencies updated, run as a non-root user, and run periodic secret/key rotation.

10. User Interface:

The interface is built using **Gradio Blocks** with **two main tabs**:

1. **Disease Prediction**
    - Input: Free-text symptoms
    - Output: Possible conditions & recommendations
2. **Treatment Plans**

- Input: Condition, Age, Gender, Medical History
- Output: Personalized treatment plan

11. Testing:

1. Testing Objectives:

Verify that the model loads correctly and works with both CPU and GPU.

Validate that disease prediction and treatment plan functions return logical outputs.

Confirm the Gradio UI loads, processes input, and displays responses.

Ensure that prompt engineering works as expected and generates relevant outputs.

Detect performance issues and edge cases.

2. Types of Tests:

A. Unit Testing

Unit testing ensures individual functions in healthai.py work correctly.

1. Test Model Loading:

```
def test_model_loading():
    assert model is not None, "Model failed to load"
    assert tokenizer is not None, "Tokenizer failed to load"
```

Expected: Model and tokenizer load without errors.

2. Test generate_response:

```
def test_generate_response():
    response = generate_response("Say hello", max_length=50)
    assert isinstance(response, str), "Response should be a string"
```

```
    assert len(response) > 0, "Response is empty"
```

Expected: Returns a non-empty string.

3. Test disease_prediction:

```
def test_disease_prediction():
    response = disease_prediction("fever, cough, headache")
    assert "IMPORTANT" in response, "Response should include disclaimer"
```

Expected: Includes medical disclaimer and condition analysis.

4. Test treatment_plan:

```
def test_treatment_plan():
    response = treatment_plan("diabetes", 45, "Male", "hypertension")
    assert "Treatment Plan" in response or len(response) > 0
```

Expected: Outputs a treatment plan section.

B. Integration Testing:

Focus on how all functions work together.

Test Case	Steps	Expected Output
App Launch	Run python healthai.py	Gradio app launches with a public link.
Disease Prediction Workflow	Enter "fever, cough" → Click Analyze Symptoms	AI response with possible conditions + disclaimer

Treatment Plan Workflow	Enter inputs for condition, age, gender, history → Click Generate Plan	AI-generated plan appears
GPU/CPU Compatibility	Run on GPU and CPU machines	Works in both environments

C. UI Testing (Manual/Automated):

Test	Steps	Expected
Textbox Functionality	Enter symptoms, click buttons	Textbox captures input, triggers backend
Button Click Actions	Click "Analyze Symptoms" and "Generate Treatment Plan"	Correct outputs appear
Tab Navigation	Switch between "Disease Prediction" and "Treatment Plans"	UI switches smoothly
Output Display	Long responses displayed correctly	Scrollable and visible

D. Performance Testing:

Test	How to Test	Expected
Response Time	Use time module to measure generate_response()	< 3s (GPU), < 10s (CPU)
Load Test	Test with 10+ concurrent users via Gradio share link	App remains responsive
Model Memory	Check nvidia-smi or torch.cuda.memory_allocated()	Memory under safe usage

E. Edge Case Testing:

Case	Input	Expected
Empty Input	""	Error message or disclaimer
Very Long Input	500+ words	Should truncate but still respond
Invalid Age	Negative or non-numeric	Handle gracefully

Unsupported Gender	Non-listed value	Default to neutral tone

F. Manual Testing Checklist:

Install dependencies (torch, transformers, gradio).

Launch app with python healthai.py.

Open link and test both tabs.

Try simple and complex symptom inputs.

Validate disclaimers are always shown.

Check app stability over multiple queries.

3. Tools to Use:

pytest: For automated unit tests.

Gradio Test Client: For UI testing in Python.

Postman or cURL: For API endpoint testing (if exposed later).

nvidia-smi: For GPU monitoring.

Locust/JMeter: For load testing.

4. Example Automated Test Script:

You can create a file test_healthai.py:

```
import pytest

from healthai import generate_response, disease_prediction, treatment_plan

def test_generate_response():

    assert len(generate_response("Hello", max_length=50)) > 0

def test_disease_prediction():

    result = disease_prediction("fever, cough")
```

```
    assert "IMPORTANT" in result
def test_treatment_plan():
    result = treatment_plan("asthma", 30, "Female", "none")
    assert len(result) > 0
```

Run:

```
pytest test_healthai.py
```

5. Test Reporting:

Use pytest --html=report.html for HTML reports.

Maintain a Test Log noting test date, environment (CPU/GPU), and any issues.

11. Screenshot:

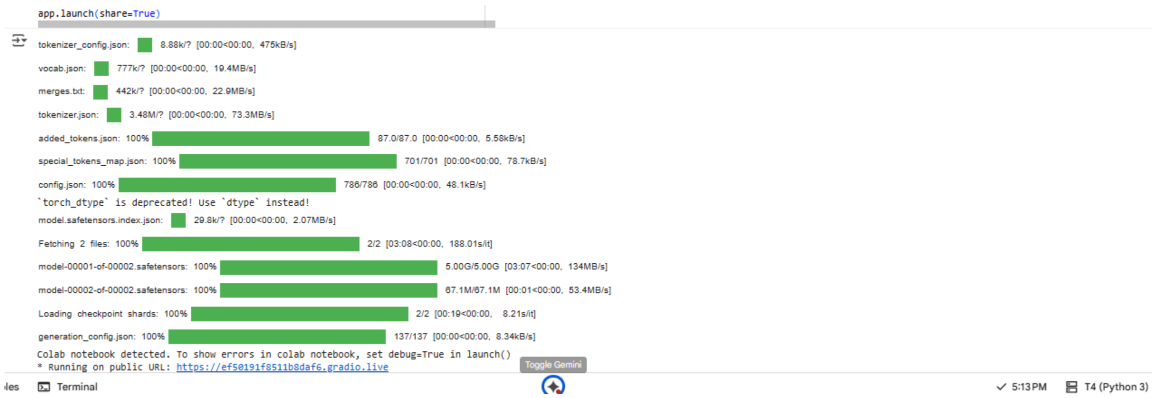
Program:

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

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

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

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

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

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
```

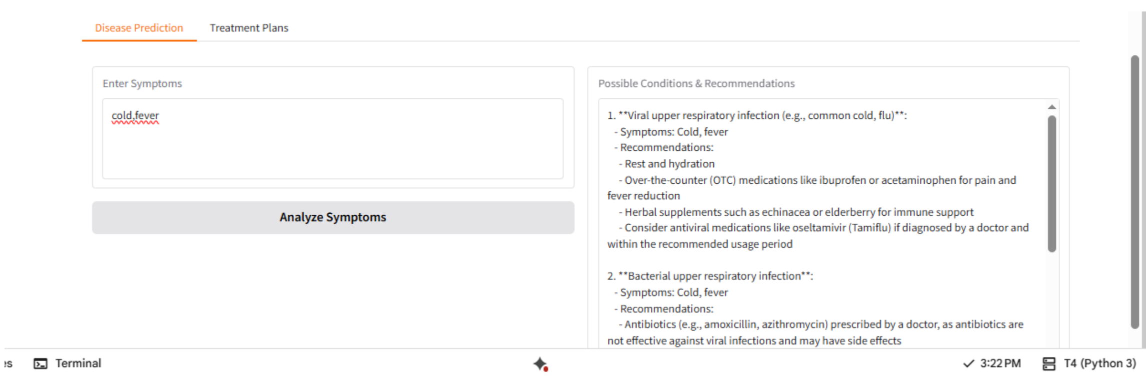
```python
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    response = response.replace(prompt, "").strip()
    return response

def disease_prediction(symptoms):
    prompt = f"Based on the following symptoms, provide possible medical conditions and general medication suggestions. Always emphasize the importance of co
    return generate_response(prompt, max_length=1200)

def treatment_plan(condition, age, gender, medical_history):
    prompt = f"Generate personalized treatment suggestions for the following patient information. Include home remedies and general medication guidelines.\n\
    return generate_response(prompt, max_length=1200)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Medical AI Assistant")
    gr.Markdown("**Disclaimer: This is for informational purposes only. Always consult healthcare professionals for medical advice.**")
```

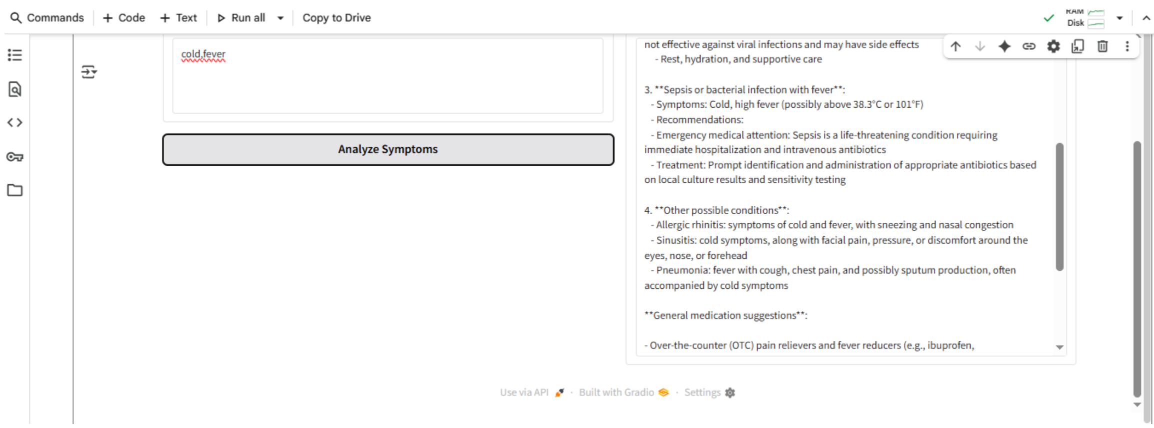
```python
# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Medical AI Assistant")
    gr.Markdown("**Disclaimer: This is for informational purposes only. Always consult healthcare professionals for medical advice.**")

    with gr.Tabs():
        with gr.TabItem("Disease Prediction"):
            with gr.Row():
                with gr.Column():
                    symptoms_input = gr.Textbox(
                        label="Enter Symptoms",
                        placeholder="e.g., fever, headache, cough, fatigue...",
                        lines=4
                    )
                    predict_btn = gr.Button("Analyze Symptoms")

                with gr.Column():
                    prediction_output = gr.Textbox(label="Possible Conditions & Recommendations", lines=20)

            predict_btn.click(disease_prediction, inputs=symptoms_input, outputs=prediction_output)

        with gr.TabItem("Treatment Plans"):
            with gr.Row():
                with gr.Column():
                    condition_input = gr.Textbox(
                        label="Medical Condition",
```

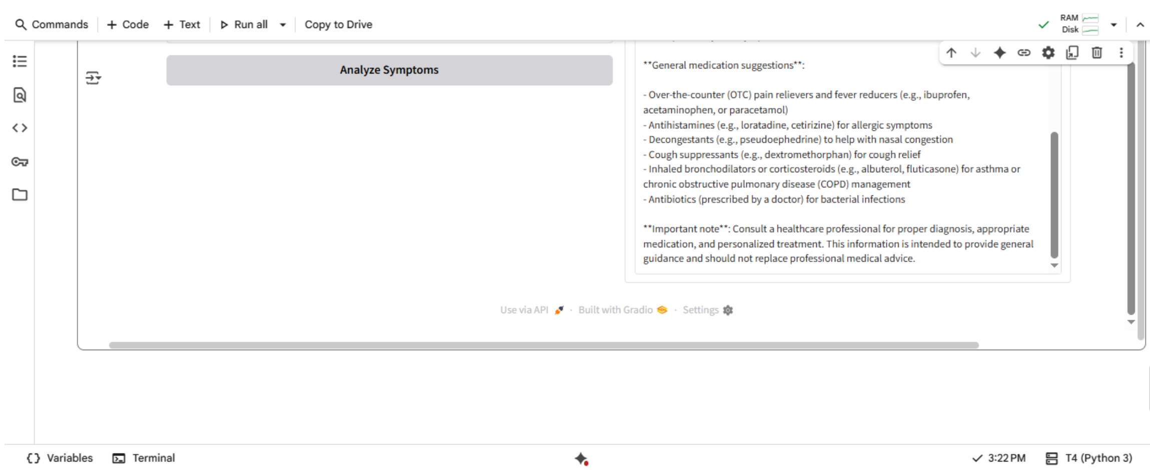
```python
        with gr.TabItem("Treatment Plans"):
            with gr.Row():
                with gr.Column():
                    condition_input = gr.Textbox(
                        label="Medical Condition",
                        placeholder="e.g., diabetes, hypertension, migraine...",
                        lines=2
                    )
                    age_input = gr.Number(label="Age", value=30)
                    gender_input = gr.Dropdown(
                        choices=["Male", "Female", "Other"],
                        label="Gender",
                        value="Male"
                    )
                    history_input = gr.Textbox(
                        label="Medical History",
                        placeholder="Previous conditions, allergies, medications or None",
                        lines=3
                    )
                    plan_btn = gr.Button("Generate Treatment Plan")

                with gr.Column():
                    plan_output = gr.Textbox(label="Personalized Treatment Plan", lines=20)

            plan_btn.click(treatment_plan, inputs=[condition_input, age_input, gender_input, history_input], outputs=plan_output)

app.launch(share=True)
```


app.launch(share=True)
tokenizer_config.json: 8.88k/? [00:00<00:00, 475kB/s]
vocab.json: 777k/? [00:00<00:00, 19.4MB/s]
merges.txt: 442k/? [00:00<00:00, 22.9MB/s]
tokenizer.json: 3.48M/? [00:00<00:00, 73.3MB/s]
added_tokens.json: 100% 87.0/87.0 [00:00<00:00, 5.58kB/s]
special_tokens_map.json: 100% 701/701 [00:00<00:00, 78.7kB/s]
config.json: 100% 786/786 [00:00<00:00, 48.1kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 29.8k/? [00:00<00:00, 2.07MB/s]
Fetching 2 files: 100% 2/2 [03:08<00:00, 188.01s/it]
model-00001-of-00002.safetensors: 100% 5.00G/5.00G [03:07<00:00, 134MB/s]
model-00002-of-00002.safetensors: 100% 67.1M/67.1M [00:01<00:00, 53.4MB/s]
Loading checkpoint shards: 100% 2/2 [00:19<00:00, 8.21s/it]
generation_config.json: 100% 137/137 [00:00<00:00, 8.34kB/s]
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://ef50191f8511b8daf6.gradio.live
Toggle Gemini
Terminal
5:13 PM
T4 (Python 3)

Output:


Disease Prediction
Treatment Plans
Enter Symptoms
cold,fever
Analyze Symptoms
Possible Conditions & Recommendations
1. **Viral upper respiratory infection (e.g., common cold, flu)**:
- Symptoms: Cold, fever
- Recommendations:
- Rest and hydration
- Over-the-counter (OTC) medications like ibuprofen or acetaminophen for pain and fever reduction
- Herbal supplements such as echinacea or elderberry for immune support
- Consider antiviral medications like oseltamivir (Tamiflu) if diagnosed by a doctor and within the recommended usage period
2. **Bacterial upper respiratory infection**:
- Symptoms: Cold, fever
- Recommendations:
- Antibiotics (e.g., amoxicillin, azithromycin) prescribed by a doctor, as antibiotics are not effective against viral infections and may have side effects
Terminal
3:22 PM
T4 (Python 3)





12. Known isseus:

1. Model limitations & hallucinations:

While Granite 3.2 claims improved reasoning, smaller models still sometimes hallucinate, especially with medical claims or specialized domain knowledge. Always verify outputs.

The model's knowledge cutoff: likely sometime in 2024. It may not know of very recent medical guidelines or drugs. Using up-to-date sources is important.

2. Safety / disclaimers:

For medical advice, even though you put a disclaimer ("for informational purposes only…"), the outputs may still assert things with confident but wrong statements. It's good you emphasize consulting a professional.

Be careful with suggested medications: the model may mention things that require prescription, may not consider interactions, allergies, etc. You may want to limit or filter any direct medication suggestions or include more safety guardrails.

3. Controllability of the "thinking" mode:

To use the reasoning ("thinking") features effectively, you'll likely need to include the right system/control prompts or parameters. From what's public, Granite 3.2 uses e.g. a "thinking" flag in some environments, or a system message.

Without this, the model might skip or under-perform the reasoning steps you expect.

4. Performance & Resource Use:

Model uses ~2B parameters. If running on GPU with FP16 (or BF16), it may be okay; on CPU or less capable GPU, might be slow or need quantization. The code uses torch_dtype=torch.float16 if torch.cuda.is_available() etc, which helps.

Also, the max_length is large (1024 or 1200 in your code) and your generate uses default settings for sample + temperature. Long outputs + sampling can be expensive.

5. Prompt / truncation issues:

In generate_response, you truncate the input to max_length=512 tokens. If the symptoms or history become large text, important detail may be lost.

Also, the response you decode removes prompt and skips special tokens—fine, but ambiguity can creep in. E.g., if model echoes or refers back, trimming might cut something needed.

6. Ethical / legal risks, especially for medical domain:

Giving medical suggestions has legal/ethical implications; make clear in UI and possibly restrict certain types of medical claims.

The model may give suggestions not valid in the user's country, or for local medical standards.

13.Future enhancements:

1. **Authentication & RBAC** — integrate OAuth or SSO (Okta, Auth0). Add role-based access for clinicians vs public users.
2. **Deploy as FastAPI microservice** — separate UI from API, easier to secure, test, and scale.
3. **Batch & streaming responses** — support streaming tokens for responsive UI.
4. **Model orchestration** — support multiple models (small fast model for triage, large model for detailed responses).
5. **Feedback & human-in-the-loop** — let clinicians review model suggestions and flag errors to collect labeled data.

6. **Fine-tune / instruction-tune** on curated medical datasets and apply RLHF to reduce hallucinations.
7. **Logging & analytics** — safe analytics to understand common queries and failure modes.
8. **Automated tests & CI/CD** — full pipeline with Docker image build, vulnerability scanning, and staging promotion.
9. **HIPAA compliance package** — hardened deployment, Business Associate Agreement (BAA), encrypted storage, audit logs.
10. **Multi-lingual & localization** — support major languages with translation or multilingual models.

14. Conclusion:

Detailed Conclusion:

The healthai.py script demonstrates a well-structured implementation of a Medical AI Assistant using the IBM Granite 3.2 Instruct LLM and Gradio as the interface framework. The primary objective of this project is to provide informational health guidance based on user input, with a strong emphasis on encouraging professional medical consultation.

Core Features:

Disease Prediction Module

Accepts user-input symptoms and generates a list of possible medical conditions and general recommendations.

Helps users get a basic understanding of potential illnesses based on their symptoms.

Treatment Plan Module

Collects additional patient details such as age, gender, and medical history to provide personalized home remedies and medication guidelines.

Designed to act as a first-level informational resource rather than a replacement for medical expertise.

Technical Highlights:

LLM Integration:

Utilizes the ibm-granite/granite-3.2-2b-instruct model from Hugging Face, enabling context-aware, natural language responses.

Device-Aware Optimization:

The script automatically detects GPU availability, switching between float16 and float32 precision, ensuring efficient performance on both CPU and GPU systems.

Clean Response Processing:

Implements decoding strategies (skip_special_tokens=True) and text cleaning for polishe

d, user-friendly output.

Gradio-Based UI:

Employs Gradio's Blocks API to create a tab-based interface with clear input fields, making it intuitive for non-technical users.

Safety & Ethics:

Each response is prefixed with medical disclaimers, ensuring that users understand the system is for informational purposes only.

Strengths of the Project:

User-Friendly Design: Easy-to-navigate interface for both disease prediction and treatment planning.

Scalable and Modular: Functions are well-separated, making it easy to add new features (like severity scoring or symptom prioritization).

Deployable Anywhere: The app.launch(share=True) feature allows instant public sharing and testing.

Ethical Considerations: Built-in disclaimers prevent misuse and remind users of the importance of consulting healthcare professionals.

Future Scope:

Add Medical Knowledge Base: Integrate with verified medical APIs or datasets (e.g., WHO, Mayo Clinic) for improved accuracy.

Severity Ranking System: Assign severity levels to symptoms or conditions to help prioritize care.

User Authentication & History: Enable users to save and review past reports for tracking symptoms over time.

Mobile-Friendly Deployment: Wrap the Gradio app into a mobile-friendly interface or deploy as a PWA.

Performance Optimization: Apply quantization techniques or lighter models for faster inference on CPU-only environments.