Health AI: Intelligent Healthcare Assistant

Generative AI with IBM

# 1.INTRODUCTION:



Health AI (Artificial Intelligence in Health AI Its purpose is to improve diagnostic accuracy, personalize care, enhance patient outcomes, and streamline administrative processes by analyzing large datasets and identifying patterns that humans might miss. Key applications include medical imaging analysis for disease detection

* Project Tittle :HealthAI
* Team Leader :Logesh C :NM\_ID: 4DDC5E16EC4E9178F1D4E13DC8FB1E1E
* Team Member : Kalaiyarasan S:NM\_ID: 4A4F027A83D0E08AD094E35F481690C7
* Team Member :Manikandan P: NM\_ID: 16189F433263553AF0E446D5B3B76482
* Team Member :Jothikrishnan R: NM\_ID: 1D8A7F332182E6F9A694D64408BC3A76

# 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:** Transformers, Gradio
* **Interface:** Web-based (accessible via browser)

## 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.

# 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.

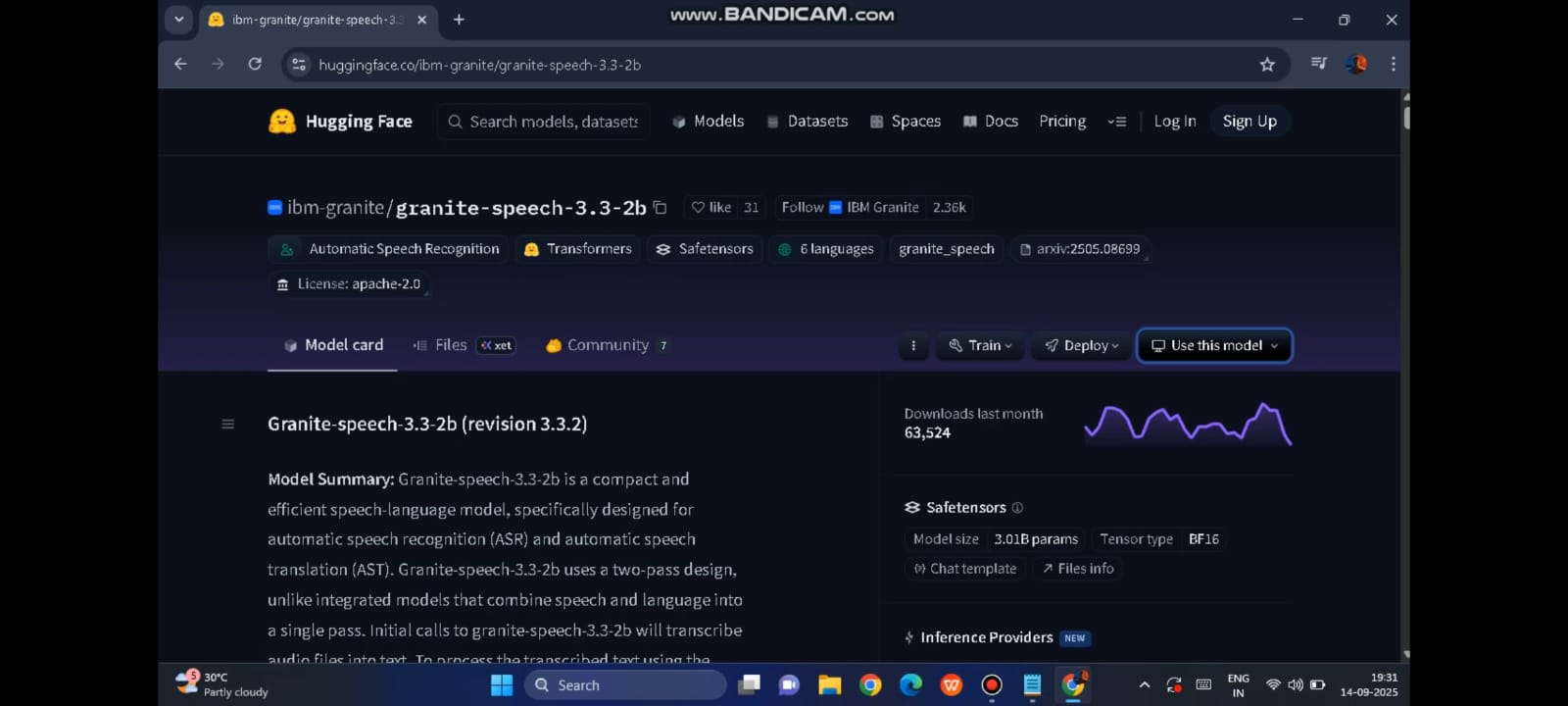
Search for “Hugging face” in any browser.

# Activity-3: Running Application in Google Collab.

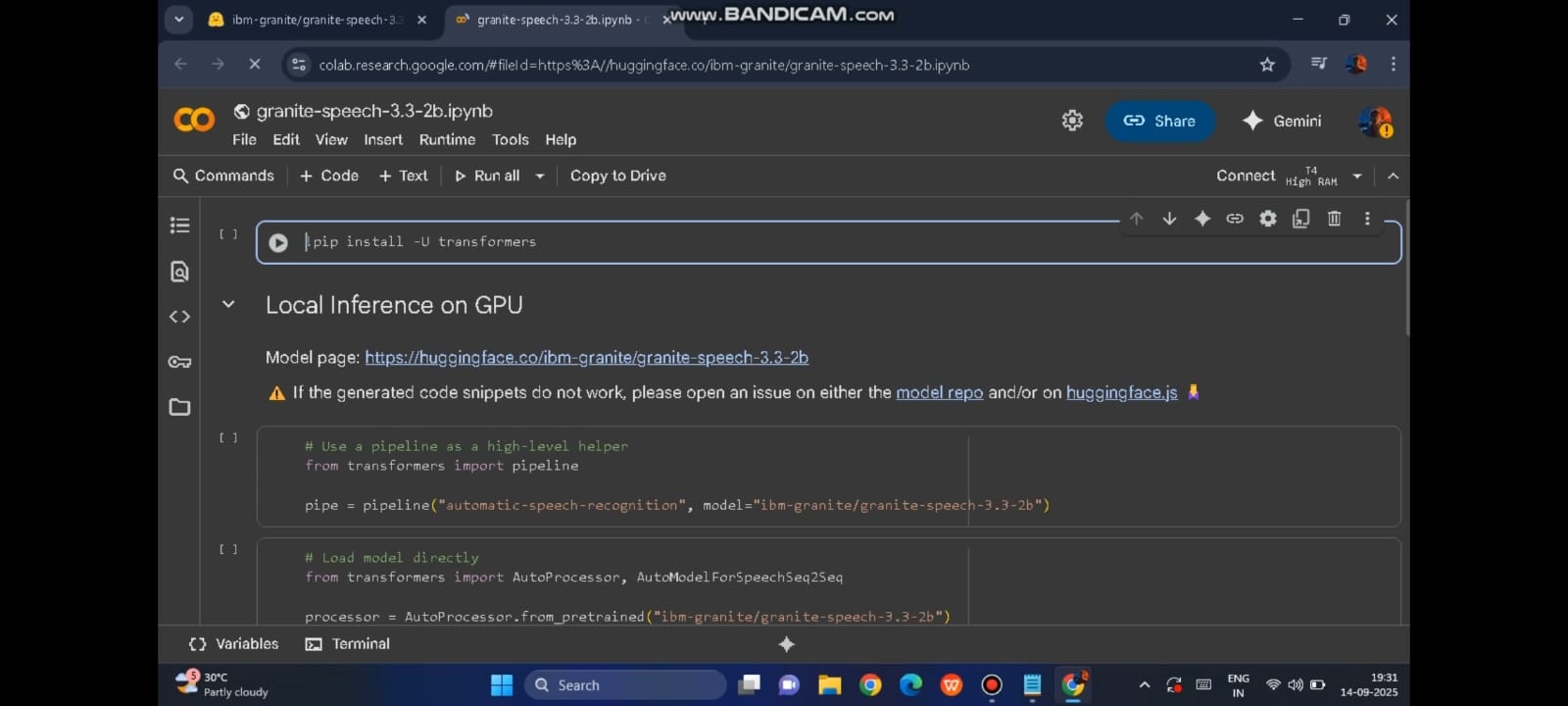
Search for “Google collab” in any browser.

## 

Choose a IBM Granite model From Hugging Face.



Running Application in Google Collab.



**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

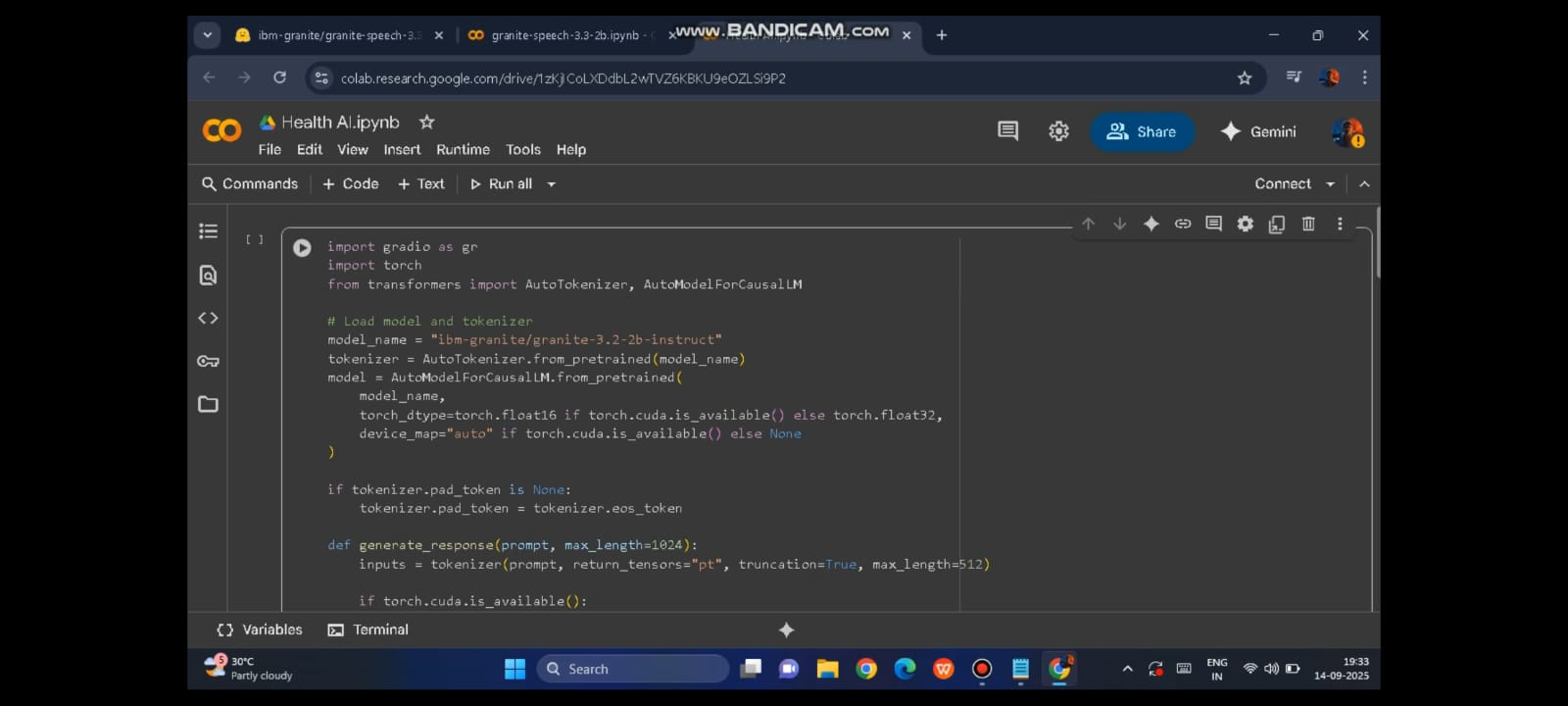
**5. Test Reporting:**

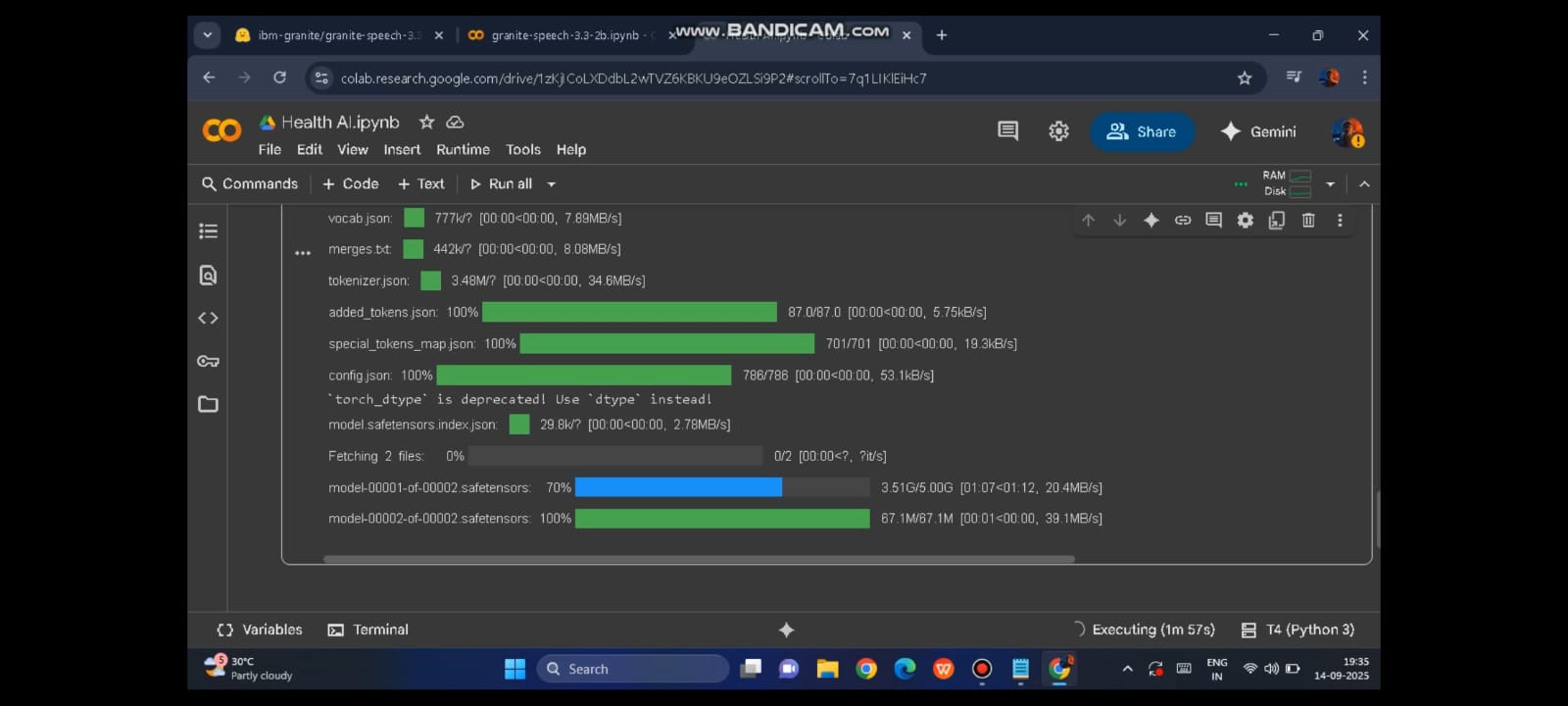
Use pytest --html=report.html for HTML reports.

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

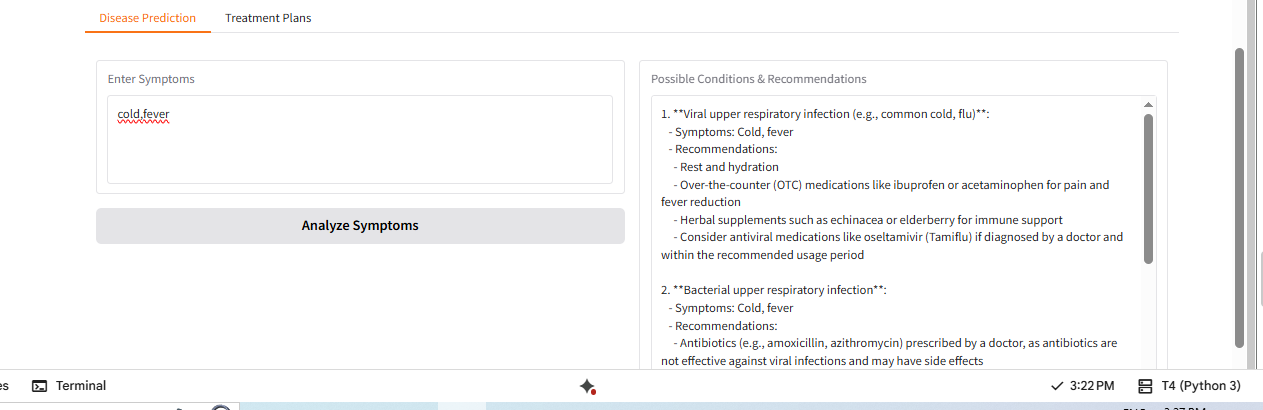
**11. Screenshot:**

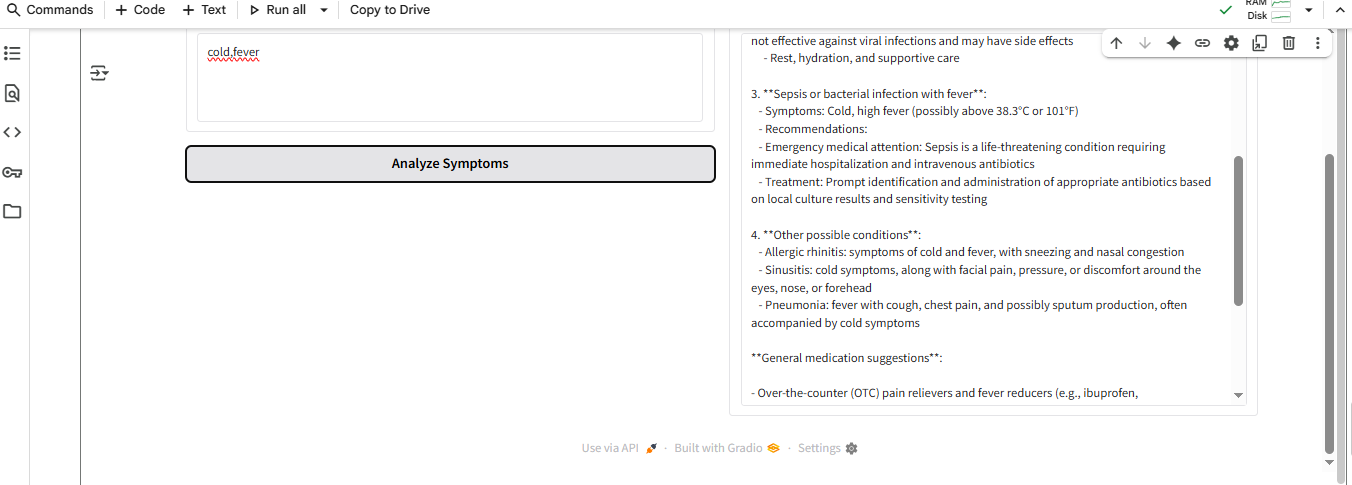
**Program:**

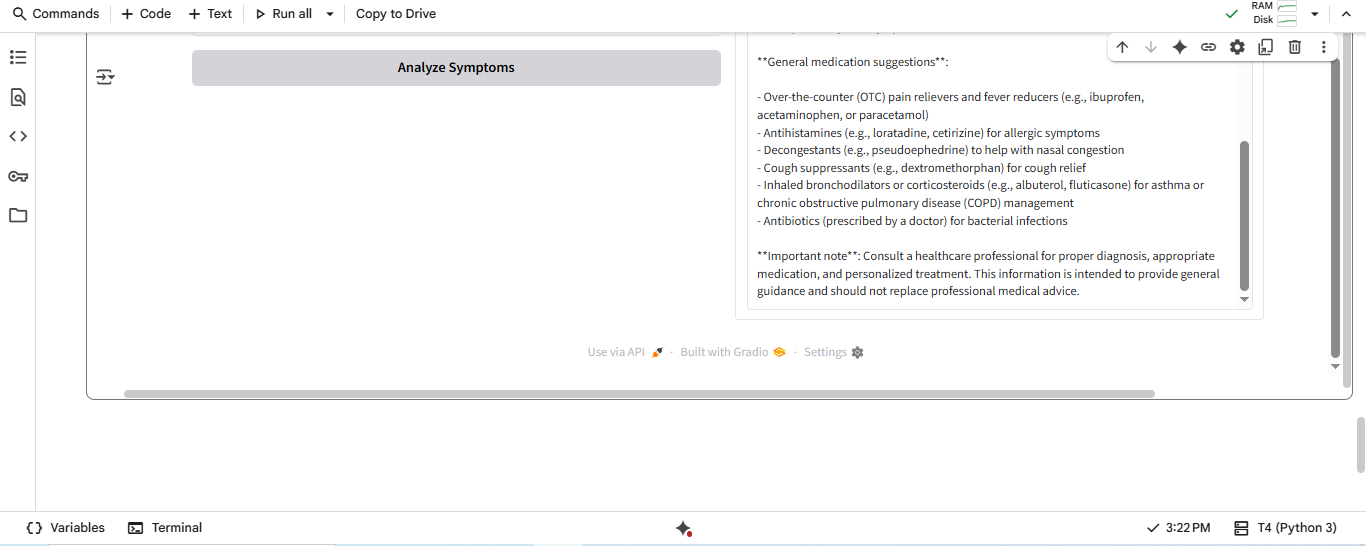
****

****

**Output:**







**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.