**1ST STEP  
  
import os**

**import subprocess**

**# ✅ Dependency List (Final Version with TensorFlow, CUDA, and GPU Support)**

**dependencies = [**

**# Core Libraries**

**"transformers torch torchvision torchaudio accelerate librosa sentencepiece datasets numpy scipy requests",**

**# Speech Processing & Recognition (Multilingual Whisper)**

**"openai-whisper ffmpeg python-multipart",**

**# Text Processing & NLP**

**"nltk spacy pydantic",**

**# Named Entity Recognition (NER) Model (SpaCy)**

**"speechbrain",**

**# Retrieval-Augmented Generation (RAG) & Vector Search**

**"faiss-gpu || faiss-cpu",  # Install GPU version first, fallback to CPU**

**# Text-to-Speech (TTS) Models**

**"TTS soundfile",**

**# FastAPI & WebSockets for Real-Time API Deployment**

**"fastapi uvicorn websockets",**

**# Voice Activity Detection (VAD) & Speaker Diarization**

**"pyannote-audio silero-vad",**

**# Performance Optimization (ONNX for Faster Inference)**

**"onnx onnxruntime-gpu torch-onnx",**

**# TensorFlow with CUDA Support**

**"tensorflow tensorflow-gpu",**

**# CUDA & GPU Acceleration (PyTorch + TensorFlow with CUDA 11.8)**

**"--index-url https://download.pytorch.org/whl/cu118 torch torchvision torchaudio",**

**# API Performance Monitoring (Prometheus)**

**"prometheus\_client",**

**# Error Logging & Debugging (Sentry)**

**"sentry-sdk",**

**# Cloud Deployment (AWS Lambda, S3)**

**"awscli boto3",**

**# Containerization & Scaling (Docker, Kubernetes)**

**"docker-compose",**

**# Security (OAuth 2.0 Authentication for API)**

**"authlib",**

**# Public API Exposure (Secure Tunnels via Ngrok)**

**"pyngrok"**

**]**

**# ✅ Install Dependencies**

**for dep in dependencies:**

**print(f"\n🔹 Installing: {dep}")**

**subprocess.run(f"pip install {dep}", shell=True, check=True)**

**# ✅ Download SpaCy Language Model**

**print("\n🔹 Downloading SpaCy Model...")**

**subprocess.run("python -m spacy download en\_core\_web\_sm", shell=True, check=True)**

**print("\n✅ All dependencies installed successfully!")**

2nd step

**import torch**

**import tensorflow as tf**

**import whisper**

**import spacy**

**import faiss**

**import numpy as np**

**import os**

**print("\n🔹 Verifying System Setup...")**

**# ✅ 1. Check CUDA & GPU Availability for PyTorch**

**print("\n🔹 Checking CUDA Availability for PyTorch...")**

**cuda\_available = torch.cuda.is\_available()**

**print(f"CUDA Available: {cuda\_available}")**

**if cuda\_available:**

**print(f"✅ Using GPU: {torch.cuda.get\_device\_name(0)}")**

**print(f"✅ PyTorch Version: {torch.\_\_version\_\_}")**

**# Check CUDA Tensor Operations**

**try:**

**x = torch.rand(2, 2).to("cuda")**

**print("✅ PyTorch CUDA Tensor Test Passed!")**

**except Exception as e:**

**print(f"❌ PyTorch CUDA Test Failed! Error: {e}")**

**else:**

**print("⚠️ Warning: Running on CPU (Expect Slower Performance)")**

**# ✅ 2. Check CUDA & GPU Availability for TensorFlow**

**print("\n🔹 Checking CUDA Availability for TensorFlow...")**

**tf\_gpu\_available = tf.config.list\_physical\_devices("GPU")**

**if tf\_gpu\_available:**

**print(f"✅ TensorFlow is using GPU: {tf\_gpu\_available}")**

**print(f"✅ TensorFlow Version: {tf.\_\_version\_\_}")**

**else:**

**print("⚠️ Warning: TensorFlow is not using GPU.")**

**# ✅ 3. Verify Whisper Model & Perform Sample Transcription**

**try:**

**print("\n🔹 Loading Whisper Model...")**

**whisper\_model = whisper.load\_model("medium") # Change to "large-v2" if needed**

**print("✅ Whisper Model Loaded Successfully!")**

**# Check if Whisper Model File Exists**

**model\_path = os.path.expanduser("~/.cache/whisper/medium.pt")**

**if os.path.exists(model\_path):**

**print(f"✅ Whisper Model File Exists: {model\_path}")**

**else:**

**print("⚠️ Warning: Whisper model file not found. It may not have downloaded correctly.")**

**# Test Transcription on a Silent Audio File**

**dummy\_audio = np.zeros((16000,), dtype=np.float32) # 1 sec of silence**

**whisper\_output = whisper\_model.transcribe(dummy\_audio)**

**print(f"✅ Whisper Test Output: {whisper\_output['text']} (This should be empty)")**

**except Exception as e:**

**print(f"❌ Whisper Model Load Failed! Error: {e}")**

**# ✅ 4. Verify FAISS Installation & Perform Indexing Test**

**try:**

**print("\n🔹 Checking FAISS Installation & Functionality...")**

**index = faiss.IndexFlatL2(768) # Create FAISS index for vector search**

**test\_vectors = np.random.rand(10, 768).astype("float32") # Generate 10 random embeddings**

**index.add(test\_vectors) # Add them to the FAISS index**

**# Check FAISS Index Size**

**index\_size = index.ntotal**

**print(f"✅ FAISS Index Size: {index\_size} vectors")**

**if index\_size == 10:**

**print("✅ FAISS Vector Indexing Successful!")**

**else:**

**print("⚠️ Warning: FAISS Index size mismatch!")**

**# Perform a FAISS search**

**query\_vector = np.random.rand(1, 768).astype("float32") # Generate a random query**

**D, I = index.search(query\_vector, 3) # Retrieve top 3 results**

**print(f"✅ FAISS Installed Successfully! Test search results: {I}")**

**except Exception as e:**

**print(f"❌ FAISS Installation Failed! Error: {e}")**

**# ✅ 5. Verify SpaCy Model & Perform Named Entity Recognition (NER) Test**

**try:**

**print("\n🔹 Checking SpaCy Model & NER Functionality...")**

**nlp = spacy.load("en\_core\_web\_sm")**

**print("✅ SpaCy Model Loaded Successfully!")**

**# Test Named Entity Recognition (NER)**

**test\_text = "Google was founded in 1998 in California by Larry Page and Sergey Brin."**

**doc = nlp(test\_text)**

**entities = [(ent.text, ent.label\_) for ent in doc.ents]**

**print(f"✅ SpaCy NER Test Passed! Extracted Entities: {entities}")**

**except Exception as e:**

**print(f"❌ SpaCy Model Not Found! Run: python -m spacy download en\_core\_web\_sm. Error: {e}")**

**print("\n✅ System Verification Complete! 🚀")  
  
STEP 3**

Speech Preprocessing (Noise Reduction, VAD, Speaker Diarization)

🔹 Steps:

1️⃣ Convert audio to 16kHz (required for Whisper & Wav2Vec2).  
2️⃣ Remove Silence & Background Noise using Silero VAD.  
3️⃣ Speaker Diarization using pyannote-audio (optional).

**import torchaudio**

**import librosa**

**import numpy as np**

**import torch**

**from silero\_vad import SileroVAD**

**from pyannote.audio.pipelines import SpeakerDiarization**

**from pyannote.core import Segment**

**import torchaudio.transforms as T**

**# Load VAD model**

**vad\_model = SileroVAD("silero\_vad")**

**# Load Speaker Diarization model (optional)**

**diarization\_pipeline = SpeakerDiarization.from\_pretrained("pyannote/speaker-diarization")**

**# Function to process speech**

**def preprocess\_speech(audio\_path):**

**# Load audio**

**waveform, sample\_rate = librosa.load(audio\_path, sr=16000)**

**# Apply VAD for noise removal**

**speech\_timestamps = vad\_model.get\_speech\_timestamps(waveform, sample\_rate)**

**clean\_waveform = np.concatenate([waveform[start:end] for start, end in speech\_timestamps])**

**# Optional: Speaker Diarization**

**diarization = diarization\_pipeline({"uri": "sample", "waveform": waveform})**

**return clean\_waveform, diarization**

**# Example usage**

**audio\_file = "sample\_audio.wav"**

**processed\_audio, speakers = preprocess\_speech(audio\_file)**

**print(f"Processed Audio Length: {len(processed\_audio)}, Speakers Detected: {speakers}")**

Text Preprocessing (Normalization & Named Entity Recognition)

🔹 Steps:

1️⃣ Lowercase, remove punctuation & contractions  
2️⃣ Expand numbers (e.g., "100" → "one hundred")  
3️⃣ Extract Named Entities for better context-aware RAG retrieval

import spacy

import re

import nltk

from nltk.corpus import wordnet

# Load English NLP model for Named Entity Recognition (NER)

nlp = spacy.load("en\_core\_web\_sm")

# Function to expand contractions

contractions = {"I'm": "I am", "you're": "you are", "it's": "it is"}

def expand\_contractions(text):

return " ".join([contractions[word] if word in contractions else word for word in text.split()])

# Function to normalize text

def preprocess\_text(text):

text = text.lower() # Convert to lowercase

text = re.sub(r'\d+', lambda x: num2words(x.group()), text) # Expand numbers

text = re.sub(r'[^\w\s]', '', text) # Remove punctuation

text = expand\_contractions(text) # Expand contractions

# Named Entity Recognition (NER)

doc = nlp(text)

entities = [(ent.text, ent.label\_) for ent in doc.ents]

return text, entities

# Example usage

text\_input = "I'm going to New York tomorrow for a Google event with 100 attendees."

clean\_text, named\_entities = preprocess\_text(text\_input)

print(f"Normalized Text: {clean\_text}")

print(f"Named Entities: {named\_entities}")

Text-Based Sentiment & Emotion Detection

🔹 Steps:

1️⃣ Tokenize text input (using XLM-RoBERTa for multilingual support).  
2️⃣ Predict Sentiment (Positive, Neutral, Negative)  
3️⃣ Detect Emotions (Happy, Angry, Sad, Fear, etc.)

from transformers import pipeline

# Load multilingual sentiment analysis model

sentiment\_pipeline = pipeline("text-classification", model="cardiffnlp/twitter-xlm-roberta-base-sentiment")

# Load emotion classification model

emotion\_pipeline = pipeline("text-classification", model="j-hartmann/emotion-english-distilroberta-base", return\_all\_scores=True)

# Function to classify sentiment & emotion

def analyze\_text\_sentiment(text):

sentiment = sentiment\_pipeline(text)

emotions = emotion\_pipeline(text)

# Extract sentiment & dominant emotion

sentiment\_label = sentiment[0]["label"]

dominant\_emotion = max(emotions[0], key=lambda x: x["score"])["label"]

return sentiment\_label, dominant\_emotion

# Example usage

text\_input = "I am feeling extremely happy today!"

sentiment, emotion = analyze\_text\_sentiment(text\_input)

print(f"Sentiment: {sentiment}, Emotion: {emotion}")

**Speech-Based Emotion Detection**

**🔹 Steps:**

1️⃣ **Extract speech features (MFCCs, Pitch, Loudness)**  
2️⃣ **Pass through Wav2Vec2 model fine-tuned for emotion classification**

import librosa

import torch

from transformers import Wav2Vec2Processor, Wav2Vec2ForSequenceClassification

# Load pre-trained Wav2Vec2 model for emotion classification

speech\_model = Wav2Vec2ForSequenceClassification.from\_pretrained("superb/wav2vec2-large-superb-er")

processor = Wav2Vec2Processor.from\_pretrained("superb/wav2vec2-large-superb-er")

# Function to process speech & classify emotion

def analyze\_speech\_emotion(audio\_path):

# Load and preprocess audio

waveform, sample\_rate = librosa.load(audio\_path, sr=16000)

inputs = processor(waveform, sampling\_rate=16000, return\_tensors="pt")

# Predict emotion

with torch.no\_grad():

logits = speech\_model(\*\*inputs).logits

predicted\_class = torch.argmax(logits, dim=-1).item()

# Emotion mapping

emotions = ["Neutral", "Happy", "Sad", "Angry", "Fear"]

return emotions[predicted\_class]

# Example usage

audio\_file = "sample\_audio.wav"

speech\_emotion = analyze\_speech\_emotion(audio\_file)

print(f"Speech Emotion: {speech\_emotion}")

Multi-Sentiment & Multi-Emotion Alignment

🔹 Steps:

1️⃣ Combine text & speech emotions  
2️⃣ Align conflicting emotions using Graph Neural Networks (GNNs) or Self-Attention Fusion

def fuse\_emotions(speech\_emotion, text\_emotion):

emotion\_scores = {"Happy": 1, "Neutral": 0, "Sad": -1, "Angry": -2, "Fear": -3}

# Weighted fusion based on confidence

final\_emotion\_score = (emotion\_scores[speech\_emotion] + emotion\_scores[text\_emotion]) / 2

final\_emotion = max(emotion\_scores, key=lambda k: abs(emotion\_scores[k] - final\_emotion\_score))

return final\_emotion

# Example usage

final\_emotion = fuse\_emotions(speech\_emotion, emotion)

print(f"Final Fused Emotion: {final\_emotion}")

RAG Fine-Tuning & Multilingual Retrieval

This module will:  
✅ Fine-tune RAG (Retrieval-Augmented Generation) for multilingual emotion-aware retrieval  
✅ Use FAISS for fast knowledge retrieval  
✅ Enhance retrieval with emotion embeddings (contrastive learning)

Load & Fine-Tune RAG Model

🔹 Steps:

1️⃣ Load the base RAG model (facebook/rag-token-base)  
2️⃣ Fine-tune it with multilingual datasets (MLQA, XGLUE, OSCAR)  
3️⃣ Enhance retrieval using FAISS for dense passage search

from transformers import RagTokenizer, RagRetriever, RagSequenceForGeneration, Trainer, TrainingArguments

from datasets import load\_dataset

import faiss

# Load pre-trained RAG model

tokenizer = RagTokenizer.from\_pretrained("facebook/rag-token-base")

retriever = RagRetriever.from\_pretrained("facebook/rag-token-base", index\_name="exact")

model = RagSequenceForGeneration.from\_pretrained("facebook/rag-token-base", retriever=retriever)

# Load multilingual dataset (MLQA)

dataset = load\_dataset("mlqa", "mlqa-translate-train")

# Fine-tuning RAG on multilingual QA dataset

def preprocess\_rag(example):

query = f"{example['question']} [CONTEXT] {example['context']}"

return tokenizer(query, return\_tensors="pt")

tokenized\_dataset = dataset["train"].map(preprocess\_rag)

training\_args = TrainingArguments(

output\_dir="./rag-finetuned",

evaluation\_strategy="epoch",

save\_strategy="epoch",

learning\_rate=2e-5,

per\_device\_train\_batch\_size=4,

num\_train\_epochs=3

)

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=tokenized\_dataset

)

# Train RAG model

trainer.train()

Integrate FAISS for Fast Retrieval

🔹 Steps:

1️⃣ Convert knowledge base into dense vectors  
2️⃣ Index documents in FAISS  
3️⃣ Use emotion embeddings to improve retrieval quality

import numpy as np

# Load FAISS

index = faiss.IndexFlatL2(768) # Assuming embeddings are 768-dimensional

# Generate random embeddings (Replace this with real document vectors)

doc\_embeddings = np.random.rand(1000, 768).astype('float32')

index.add(doc\_embeddings) # Add embeddings to FAISS index

# Example query vector

query\_embedding = np.random.rand(1, 768).astype('float32')

D, I = index.search(query\_embedding, k=5) # Retrieve top 5 results

print(f"Top retrieved document IDs: {I}")

**Emotion-Aware RAG Query Expansion**

**🔹 Steps:**

1️⃣ **Modify RAG query based on detected emotions**  
2️⃣ **Use Contrastive Learning (InfoNCE loss) to improve retrieval**

def modify\_query\_with\_emotion(query, emotion):

emotion\_context = {

"Happy": "positive reinforcement",

"Sad": "supportive and encouraging",

"Angry": "calm and neutral response",

"Fear": "reassurance and guidance"

}

return f"{query} [Emotion: {emotion} - Provide a {emotion\_context[emotion]} response]"

# Example usage

query = "Tell me about climate change"

modified\_query = modify\_query\_with\_emotion(query, "Sad")

print(f"Modified Query: {modified\_query}")

Step 4: Response Generation (Emotion-Aware LLMs & TTS)

This module will:  
✅ Generate responses using fine-tuned LLMs (Mistral, Llama 3, OpenChat)  
✅ Adjust responses based on detected emotions  
✅ Convert text responses to speech using emotion-aware TTS (VITS, Tacotron2, FastSpeech2)

Load & Generate Responses Using Fine-Tuned LLMs

🔹 Steps:

1️⃣ Load pre-trained multilingual LLM (Mistral-7B, Llama 3, OpenChat)  
2️⃣ Modify response style based on detected emotion  
3️⃣ Fine-tune LLM to align responses with emotional tone

from transformers import AutoModelForCausalLM, AutoTokenizer

# Load pre-trained LLM

model\_name = "mistralai/Mistral-7B-Instruct-v0.2"

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

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

# Function to generate emotion-aware response

def generate\_response(prompt, emotion):

emotion\_tone = {

"Happy": "Use an optimistic and cheerful tone.",

"Sad": "Provide a supportive and comforting response.",

"Angry": "Maintain a calm and neutral response.",

"Fear": "Offer reassurance and guidance."

}

prompt = f"{prompt} [Emotion: {emotion} - {emotion\_tone[emotion]}]"

inputs = tokenizer(prompt, return\_tensors="pt")

output = model.generate(\*\*inputs, max\_length=100)

return tokenizer.decode(output[0], skip\_special\_tokens=True)

# Example usage

query = "What are the benefits of meditation?"

emotion = "Sad"

response = generate\_response(query, emotion)

print(f"AI Response: {response}")

Convert AI Response to Emotion-Aware Speech (TTS)

🔹 Steps:

1️⃣ Use VITS, Tacotron2, or FastSpeech2 to synthesize speech  
2️⃣ Apply emotion-based voice modulation  
3️⃣ Save & Play the generated audio

from TTS.api import TTS

import soundfile as sf

# Load pre-trained TTS model (VITS)

tts\_model = TTS("tts\_models/en/ljspeech/vits").to("cuda")

# Function to generate speech with emotion

def text\_to\_speech(text, emotion):

emotion\_tone = {

"Happy": "excited",

"Sad": "sad",

"Angry": "angry",

"Fear": "narrator"

}

output\_path = "output\_audio.wav"

# Generate speech with emotion-based modulation

tts\_model.tts\_to\_file(text=text, speaker\_wav=None, file\_path=output\_path, emotion=emotion\_tone[emotion])

return output\_path

# Example usage

tts\_file = text\_to\_speech(response, emotion)

print(f"Generated Speech File: {tts\_file}")

Real-Time API Deployment (FastAPI + WebSockets + Streaming Whisper)

This module will:  
✅ Deploy the full system as a real-time API using FastAPI  
✅ Enable speech streaming with WebSockets for low-latency processing  
✅ Integrate all modules (Preprocessing → Sentiment/Emotion → RAG → Response)

Implement FastAPI Backend

🔹 Steps:

1️⃣ Create API endpoints for speech & text processing  
2️⃣ Use WebSockets for real-time speech streaming  
3️⃣ Deploy using Uvicorn

from fastapi import FastAPI, WebSocket

from transformers import pipeline

import whisper

import torch

import uvicorn

app = FastAPI()

# Load Models

asr\_model = whisper.load\_model("medium") # Whisper for speech-to-text

sentiment\_pipeline = pipeline("text-classification", model="cardiffnlp/twitter-xlm-roberta-base-sentiment")

emotion\_pipeline = pipeline("text-classification", model="j-hartmann/emotion-english-distilroberta-base")

tts\_model = TTS("tts\_models/en/ljspeech/vits").to("cuda")

# Speech-to-Text API

@app.post("/speech-to-text/")

async def speech\_to\_text(audio\_file: bytes):

with open("temp.wav", "wb") as f:

f.write(audio\_file)

text = asr\_model.transcribe("temp.wav")["text"]

return {"transcription": text}

# Sentiment & Emotion API

@app.post("/analyze-text/")

async def analyze\_text(text: str):

sentiment = sentiment\_pipeline(text)[0]["label"]

emotion = max(emotion\_pipeline(text)[0], key=lambda x: x["score"])["label"]

return {"sentiment": sentiment, "emotion": emotion}

# Text-to-Speech API

@app.post("/text-to-speech/")

async def text\_to\_speech(text: str, emotion: str):

output\_path = f"{emotion}\_response.wav"

tts\_model.tts\_to\_file(text=text, file\_path=output\_path, emotion=emotion)

return {"audio\_file": output\_path}

# WebSocket for Real-Time Speech Streaming

@app.websocket("/ws/")

async def websocket\_endpoint(websocket: WebSocket):

await websocket.accept()

while True:

audio\_data = await websocket.receive\_bytes()

with open("streaming\_audio.wav", "wb") as f:

f.write(audio\_data)

text = asr\_model.transcribe("streaming\_audio.wav")["text"]

sentiment = sentiment\_pipeline(text)[0]["label"]

emotion = max(emotion\_pipeline(text)[0], key=lambda x: x["score"])["label"]

response = f"Detected Emotion: {emotion}, Sentiment: {sentiment}"

await websocket.send\_text(response)

# Run FastAPI Server

if \_\_name\_\_ == "\_\_main\_\_":

uvicorn.run(app, host="0.0.0.0", port=8000)

**Deploy & Test the API**

**Run FastAPI Server**

Run this command in your terminal to **start the API**:

**uvicorn filename:app --host 0.0.0.0 --port 8000 --reload**

Replace filename with the Python script name.

**Test API Endpoints**

1️⃣ **Speech-to-Text** (Upload a .wav file)

**curl -X POST "http://localhost:8000/speech-to-text/" -F "audio\_file=@sample.wav"**

2️⃣ **Analyze Sentiment & Emotion**

curl -X POST "http://localhost:8000/analyze-text/" -d '{"text": "I am very happy today!"}' -H "Content-Type: application/json"

3️⃣ **Convert Text to Speech (Emotion-Aware)**

**curl -X POST "http://localhost:8000/text-to-speech/" -d '{"text": "Hello, how are you?", "emotion": "happy"}' -H "Content-Type: application/json"**

4️⃣ **Test Real-Time WebSocket Streaming**  
Use a WebSocket client like wscat:

**wscat -c ws://localhost:8000/ws/**

Send audio **bytes**, and the server **processes & returns sentiment & emotion in real-time**.

**🚀 Step 6: Post-Deployment Enhancements & Monitoring**

Now that the system is fully deployed, we will optimize, monitor, and scale it for **real-world production** use.

**1. Performance Optimization**

This step improves **inference speed, memory efficiency, and API response time**.

import torch

import onnx

# Convert Whisper Model to ONNX

whisper\_model = torch.jit.load("whisper\_model.pth") # Load pre-trained model

dummy\_input = torch.randn(1, 80, 3000) # Example input

torch.onnx.export(whisper\_model, dummy\_input, "whisper\_model.onnx")

print("ONNX conversion successful!")

**next code part**

from torch.quantization import quantize\_dynamic

quantized\_model = quantize\_dynamic(whisper\_model, {torch.nn.Linear}, dtype=torch.qint8)

torch.save(quantized\_model, "whisper\_model\_quantized.pth")

print("Model quantization complete!")

**System Monitoring & Logging**

This step tracks **API usage, errors, and system health**.

from prometheus\_client import start\_http\_server, Counter

# Define API request counter

REQUEST\_COUNT = Counter("api\_requests\_total", "Total number of API requests")

# Start Prometheus HTTP Server

start\_http\_server(8001)

@app.middleware("http")

async def add\_process\_time\_header(request, call\_next):

REQUEST\_COUNT.inc()

response = await call\_next(request)

return response

**NEXT PART OF CODE**

import sentry\_sdk

sentry\_sdk.init("YOUR\_SENTRY\_PROJECT\_URL")

@app.exception\_handler(Exception)

async def handle\_exception(request, exc):

sentry\_sdk.capture\_exception(exc)

return JSONResponse(content={"error": "Internal Server Error"}, status\_code=500)

**Scaling & Cloud Deployment**

This step **moves the system to cloud** for **high availability**.

FROM python:3.9

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["uvicorn", "multimodal\_rag:app", "--host", "0.0.0.0", "--port", "8000"]

**Next part of the code**

docker build -t multimodal\_rag .

docker run -p 8000:8000 multimodal\_rag

**Next part of the code**

pip install awscli

aws lambda create-function --function-name multimodal\_rag --runtime python3.9 --handler multimodal\_rag.handler --memory-size 512

**Security & API Exposure**

**This step protects the API from unauthorized access.**

from fastapi.security import OAuth2PasswordBearer

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.get("/secure-endpoint/")

async def secure\_data(token: str = Depends(oauth2\_scheme)):

return {"message": "Access granted"}

**next part of the code**

pip install pyngrok

ngrok http 8000