import os

import time

import io

import base64

import json

import soundfile as sf

import librosa # Using librosa for consistency if advanced features are needed later

from dotenv import load\_dotenv

from typing import TypedDict, List, Dict, Any, Optional

from langchain\_openai import ChatOpenAI

from pydantic import BaseModel, Field

from langchain\_core.messages import SystemMessage, HumanMessage

from langgraph.graph import StateGraph, END

from langgraph.checkpoint.sqlite import SqliteSaver # For potential checkpointing if needed later

# --- Environment Setup ---

load\_dotenv()

# Phi-4 Configuration (from your existing code)

AZURE\_PHI4\_API\_KEY = os.getenv("PHI4\_API\_KEY", "your\_phi4\_api\_key") # Replace with actual or ensure .env

AZURE\_PHI4\_ENDPOINT\_FULL = os.getenv("PHI4\_ENDPOINT", "your\_phi4\_endpoint\_url/chat/completions?api-version=2024-05-01-preview") # Replace

AZURE\_PHI4\_MODEL\_NAME = "Phi-4-multimodal-instruct" # This is used in the payload for some Azure setups

# GPT-4o Configuration (newly provided)

AZURE\_GPT4O\_API\_KEY = os.getenv("AZURE\_GPT4O\_API\_KEY", "Cfjqwi4QrkAqdf5dwKOabIxAtEGh0Rgxi5LEM0EuwtfuNSj6MNtNJQQJ99BEACHYHv6XJ3w3AAAAACOGQbLu") # User provided

AZURE\_GPT4O\_ENDPOINT\_FULL = os.getenv("AZURE\_GPT4O\_ENDPOINT", "https://dm-ai-agents-openai-service.cognitiveservices.azure.com/openai/deployments/gpt-4o/chat/completions?api-version=2025-01-01-preview") # User provided

AZURE\_GPT4O\_DEPLOYMENT\_NAME = "gpt-4o" # This is often the model name for Azure OpenAI direct calls

# --- Pydantic Models for State Sub-structures ---

class UserMetadata(BaseModel):

domains\_of\_interest: Optional[List[str]] = Field(default\_factory=list)

redact\_pii: bool = False

class TranscriptionSegment(BaseModel):

text: str

start\_time: float # Placeholder, actual timing from ASR needed

end\_time: float # Placeholder

speaker\_label: Optional[str] = None # To be filled by DiarizerAgent

confidence: Optional[float] = None # From ASR model

class SpeakerTimestamp(BaseModel):

speaker\_label: str

start\_time: float

end\_time: float

class PIIEntity(BaseModel):

type: str

text: str

start\_char\_offset: Optional[int] = None # Character offset in full\_transcript\_text

end\_char\_offset: Optional[int] = None

confidence: Optional[float] = None

class DomainDataItem(BaseModel):

domain: str

topic: Optional[str] = None

keywords: Optional[List[str]] = Field(default\_factory=list)

sentiment: Optional[str] = None

confidence: Optional[float] = None

# --- LangGraph State Definition (as per your document) ---

class ComprehensiveGraphState(TypedDict):

original\_audio\_path: str

audio\_duration\_seconds: Optional[float]

user\_metadata: UserMetadata

# Fields populated by agents

base64\_audio: Optional[str] # Added for Phi-4 multimodal input if needed by transcriber

transcription\_segments: Optional[List[TranscriptionSegment]]

full\_transcript\_text: Optional[str]

speaker\_timestamps: Optional[List[SpeakerTimestamp]]

identified\_pii: Optional[List[PIIEntity]]

domain\_specific\_data: Optional[List[DomainDataItem]]

summary\_text: Optional[str]

# Control and status

error\_log: List[str]

final\_output\_json: Optional[Dict[str, Any]]

current\_task\_description: Optional[str]

# For conditional logic

transcription\_quality\_ok: bool # Default to True, set by Transcriber or QualityCheckNode

run\_pii\_detection: bool

run\_domain\_analysis: bool

# --- LLM Client Initialization ---

def get\_azure\_phi4\_llm(max\_tokens=2000, temperature=0.1):

"""Initializes the ChatOpenAI client for Azure Phi-4 multimodal endpoint."""

if not AZURE\_PHI4\_ENDPOINT\_FULL or not AZURE\_PHI4\_API\_KEY:

raise ValueError("Azure Phi-4 endpoint or API key is not configured.")

# The base URL is the endpoint URL without the specific path like /chat/completions

base\_url\_parts = AZURE\_PHI4\_ENDPOINT\_FULL.split('/')

azure\_phi4\_base\_url = "/".join(base\_url\_parts[:-2]) # e.g., https://resource.openai.azure.com/openai/deployments/MODEL\_NAME

return ChatOpenAI(

openai\_api\_key=AZURE\_PHI4\_API\_KEY,

azure\_endpoint=azure\_phi4\_base\_url, # Use azure\_endpoint for clarity with official SDK

api\_version=AZURE\_PHI4\_ENDPOINT\_FULL.split('api-version=')[-1], # Extract api-version

azure\_deployment=AZURE\_PHI4\_MODEL\_NAME, # This refers to your deployment name for Phi-4

temperature=temperature,

max\_tokens=max\_tokens,

default\_headers={"api-key": AZURE\_PHI4\_API\_KEY} # Some Azure setups need this explicitly

)

def get\_azure\_gpt4o\_llm(max\_tokens=2000, temperature=0.3):

"""Initializes the ChatOpenAI client for Azure GPT-4o endpoint."""

if not AZURE\_GPT4O\_ENDPOINT\_FULL or not AZURE\_GPT4O\_API\_KEY:

raise ValueError("Azure GPT-4o endpoint or API key is not configured.")

base\_url\_parts = AZURE\_GPT4O\_ENDPOINT\_FULL.split('/')

# Correctly form the base URL, typically up to /openai/deployments/DEPLOYMENT\_NAME

azure\_gpt4o\_base\_url = "/".join(base\_url\_parts[:-2])

return ChatOpenAI(

openai\_api\_key=AZURE\_GPT4O\_API\_KEY,

azure\_endpoint=azure\_gpt4o\_base\_url,

api\_version=AZURE\_GPT4O\_ENDPOINT\_FULL.split('api-version=')[-1],

azure\_deployment=AZURE\_GPT4O\_DEPLOYMENT\_NAME, # Deployment name for GPT-4o

temperature=temperature,

max\_tokens=max\_tokens,

default\_headers={"api-key": AZURE\_GPT4O\_API\_KEY}

)

# --- Agent Classes ---

class AudioLoaderPreparer:

"""Handles loading audio and initial state preparation."""

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: Audio Loader & Preparer ---")

state["current\_task\_description"] = f"Loading and preparing audio from: {state['original\_audio\_path']}"

try:

audio, sr = librosa.load(state["original\_audio\_path"], sr=16000, mono=True)

state["audio\_duration\_seconds"] = librosa.get\_duration(y=audio, sr=sr)

# Encode for Phi-4 if it strictly needs base64 and can't take path/bytes directly via a tool

# (Your previous code used base64 for Phi-4)

wav\_buffer = io.BytesIO()

sf.write(wav\_buffer, audio, 16000, format='WAV', subtype='PCM\_16')

wav\_buffer.seek(0)

audio\_bytes = wav\_buffer.read()

if not audio\_bytes:

state["error\_log"].append("Audio data is empty after processing.")

return state

state["base64\_audio"] = base64.b64encode(audio\_bytes).decode('utf-8')

print(f"Audio loaded. Duration: {state['audio\_duration\_seconds']:.2f}s. Encoded for multimodal.")

except Exception as e:

error\_msg = f"Error during audio loading/preparation: {str(e)}"

print(error\_msg)

state["error\_log"].append(error\_msg)

return state

class AudioTranscriberAgent:

def \_\_init\_\_(self):

# Assuming Phi-4 is preferred for transcription due to multimodal capabilities

self.llm = get\_azure\_phi4\_llm(max\_tokens=2500, temperature=0.0) # Low temp for accuracy

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: Audio Transcriber ---")

state["current\_task\_description"] = "Transcribing audio using Phi-4 Multimodal."

if state["error\_log"] or not state.get("base64\_audio"):

state["error\_log"].append("Skipping transcription due to previous error or no audio data.")

return state

# Simplified transcription prompt based on your previous one, focusing on content.

# Structured JSON output for segments might be challenging for some models without fine-tuning.

# For Phi-4, a direct transcription instruction might be more robust initially.

transcription\_prompt\_instructions = (

"You are an expert audio transcription service. Transcribe the provided audio content accurately. "

"Output ONLY the transcribed text. Do not add any speaker labels, summaries, or any other text. "

"Just provide the raw, continuous transcript of what is spoken in the audio."

)

# If Phi-4 can handle structured output for segments directly, the prompt would be more complex,

# similar to your original JSON request. For now, we aim for raw text and let Diarizer handle speakers.

messages = [

SystemMessage(content="You are an AI assistant that transcribes audio."),

HumanMessage(content=[

{"type": "text", "text": transcription\_prompt\_instructions},

{

"type": "audio\_url", # As per your working Phi-4 example

"audio\_url": {"url": f"data:audio/wav;base64,{state['base64\_audio']}"}

}

])

]

try:

print("Sending transcription request to Phi-4 endpoint...")

start\_time = time.time()

response = self.llm.invoke(messages)

print(f"Phi-4 transcription call completed in {time.time() - start\_time:.2f} seconds.")

full\_transcript = response.content.strip()

print(f"Raw transcript: {full\_transcript[:500]}...") # Print a snippet

state["full\_transcript\_text"] = full\_transcript

# Placeholder for segmenting and word-level timestamps (would require a dedicated ASR tool like WhisperX)

# For now, create a single segment. Diarizer will split this by speaker.

state["transcription\_segments"] = [

TranscriptionSegment(

text=full\_transcript,

start\_time=0.0, # Placeholder

end\_time=state.get("audio\_duration\_seconds", 0.0), # Placeholder

confidence=0.85 # Placeholder confidence

)

]

state["transcription\_quality\_ok"] = True # Assume OK for now

except Exception as e:

error\_msg = f"Error during Phi-4 transcription: {str(e)}"

print(error\_msg)

state["error\_log"].append(error\_msg)

state["transcription\_quality\_ok"] = False

return state

class SpeakerDiarizerAgent:

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: Speaker Diarizer ---")

state["current\_task\_description"] = "Performing speaker diarization."

if state["error\_log"] or not state.get("full\_transcript\_text"):

state["error\_log"].append("Skipping speaker diarization.")

return state

# --- Placeholder for actual diarization logic ---

# In a real implementation, you'd use a library like pyannote.audio here.

# Example:

# from pyannote.audio import Pipeline

# pipeline = Pipeline.from\_pretrained("pyannote/speaker-diarization-3.1", use\_auth\_token="YOUR\_HF\_TOKEN")

# diarization\_result = pipeline(state["original\_audio\_path"])

# speaker\_timestamps = []

# for turn, \_, speaker in diarization\_result.itertracks(yield\_label=True):

# speaker\_timestamps.append(SpeakerTimestamp(speaker\_label=speaker, start\_time=turn.start, end\_time=turn.end))

# state["speaker\_timestamps"] = speaker\_timestamps

# You would also map these timestamps back to transcription\_segments to assign speaker labels.

print("Simulating speaker diarization (placeholder)...")

# Create dummy speaker timestamps based on existing segments (if any) or make new ones

duration = state.get("audio\_duration\_seconds", 10.0) # default if no duration

if state.get("transcription\_segments") and len(state["transcription\_segments"]) > 1:

# If transcriber somehow gave multiple segments, try to assign alternating speakers

state["speaker\_timestamps"] = []

current\_speaker\_idx = 0

for i, seg in enumerate(state["transcription\_segments"]):

speaker\_label = f"Speaker { (i % 2) + 1 }"

seg.speaker\_label = speaker\_label # Update segment

state["speaker\_timestamps"].append(

SpeakerTimestamp(speaker\_label=speaker\_label, start\_time=seg.start\_time, end\_time=seg.end\_time)

)

elif duration > 0: # Create some dummy timestamps for demo

state["speaker\_timestamps"] = [

SpeakerTimestamp(speaker\_label="Speaker 1", start\_time=0.0, end\_time=duration / 2),

SpeakerTimestamp(speaker\_label="Speaker 2", start\_time=duration / 2, end\_time=duration)

]

# Update the single transcription\_segment if it exists

if state.get("transcription\_segments"):

state["transcription\_segments"][0].speaker\_label = "Speaker 1 (partially assigned)"

else:

state["speaker\_timestamps"] = []

print(f"Generated speaker timestamps (placeholder): {state['speaker\_timestamps']}")

return state

class PIIDetectorAgent:

def \_\_init\_\_(self):

self.llm = get\_azure\_gpt4o\_llm(max\_tokens=500, temperature=0.1) # GPT-4o for better detection

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: PII Detector ---")

state["current\_task\_description"] = "Detecting PII/PHI entities."

if state["error\_log"] or not state.get("full\_transcript\_text"):

state["error\_log"].append("Skipping PII detection.")

return state

transcript = state["full\_transcript\_text"]

# redact\_pii = state["user\_metadata"].redact\_pii # Use this if redaction logic is built here

prompt = (

"You are an expert in identifying Personal Identifiable Information (PII) and Protected Health Information (PHI) in text. "

"Analyze the following transcript and identify any PII/PHI entities. For each entity, specify its type and the exact text. "

"Common PII types include: PERSON\_NAME, PHONE\_NUMBER, EMAIL\_ADDRESS, ADDRESS, DATE\_OF\_BIRTH, SSN, CREDIT\_CARD\_NUMBER, BANK\_ACCOUNT\_NUMBER. "

"Common PHI types include: MEDICAL\_RECORD\_NUMBER, DIAGNOSIS, TREATMENT, MEDICATION. \n"

"Respond ONLY with a JSON list of identified entities, where each item is an object with 'type' and 'text' fields. "

"If no PII/PHI is found, return an empty list []. Do not add any explanations or extra text.\n\n"

"Example format: [{'type': 'PERSON\_NAME', 'text': 'John Doe'}, {'type': 'PHONE\_NUMBER', 'text': '555-123-4567'}]\n\n"

f"Transcript to analyze:\n```\n{transcript}\n```"

)

messages = [HumanMessage(content=prompt)]

try:

print("Sending PII detection request to GPT-4o endpoint...")

response\_content = self.llm.invoke(messages).content

print(f"Raw PII response: {response\_content}")

# Robust JSON extraction

json\_start = response\_content.find('[')

json\_end = response\_content.rfind(']') + 1

if json\_start != -1 and json\_end > json\_start:

json\_str = response\_content[json\_start:json\_end]

identified\_pii\_raw = json.loads(json\_str)

state["identified\_pii"] = [PIIEntity(\*\*item) for item in identified\_pii\_raw if isinstance(item, dict) and "type" in item and "text" in item]

else: # If no list found, assume no PII or malformed response

state["identified\_pii"] = []

if response\_content.strip() and response\_content.strip() != "[]": # Log if there was content but not list

state["error\_log"].append(f"PII response was not a valid list: {response\_content}")

print(f"Identified PII: {state['identified\_pii']}")

except Exception as e:

error\_msg = f"Error during PII detection: {str(e)}"

print(error\_msg)

state["error\_log"].append(error\_msg)

state["identified\_pii"] = [] # Ensure it's an empty list on error

return state

class DomainAnalyzerAgent:

def \_\_init\_\_(self):

self.llm = get\_azure\_gpt4o\_llm(max\_tokens=1000, temperature=0.2)

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: Domain Analyzer ---")

state["current\_task\_description"] = "Analyzing transcript for domain-specific data."

if state["error\_log"] or not state.get("full\_transcript\_text"):

state["error\_log"].append("Skipping domain analysis.")

return state

transcript = state["full\_transcript\_text"]

domains\_of\_interest = state["user\_metadata"].domains\_of\_interest

domain\_prompt\_str = f"Domains of interest: {', '.join(domains\_of\_interest) if domains\_of\_interest else 'General (auto-detect if possible)'}."

prompt = (

"You are an expert in domain-specific analysis of conversations. "

f"Analyze the following transcript. {domain\_prompt\_str}\n"

"For each relevant domain, identify the main topic discussed, key entities or keywords related to that topic within the domain, and overall sentiment for that domain segment if discernible. "

"Respond ONLY with a JSON list, where each item is an object with 'domain', 'topic', 'keywords' (list of strings), and 'sentiment' (e.g., Positive, Negative, Neutral) fields. "

"If no specific domain data is found, return an empty list []. Do not add any explanations or extra text.\n\n"

"Example: [{'domain': 'Finance', 'topic': 'Loan Application Query', 'keywords': ['mortgage', 'interest rate', 'application status'], 'sentiment': 'Neutral'}]\n\n"

f"Transcript to analyze:\n```\n{transcript}\n```"

)

messages = [HumanMessage(content=prompt)]

try:

print("Sending domain analysis request to GPT-4o endpoint...")

response\_content = self.llm.invoke(messages).content

print(f"Raw domain analysis response: {response\_content}")

json\_start = response\_content.find('[')

json\_end = response\_content.rfind(']') + 1

if json\_start != -1 and json\_end > json\_start:

json\_str = response\_content[json\_start:json\_end]

domain\_data\_raw = json.loads(json\_str)

state["domain\_specific\_data"] = [DomainDataItem(\*\*item) for item in domain\_data\_raw if isinstance(item, dict) and "domain" in item]

else: # If no list found, assume no domain data or malformed response

state["domain\_specific\_data"] = []

if response\_content.strip() and response\_content.strip() != "[]":

state["error\_log"].append(f"Domain analysis response was not a valid list: {response\_content}")

print(f"Identified domain data: {state['domain\_specific\_data']}")

except Exception as e:

error\_msg = f"Error during domain analysis: {str(e)}"

print(error\_msg)

state["error\_log"].append(error\_msg)

state["domain\_specific\_data"] = []

return state

class SummarizationAgent:

def \_\_init\_\_(self):

# GPT-4o is good for summarization as well

self.llm = get\_azure\_gpt4o\_llm(max\_tokens=500, temperature=0.4)

def execute(self, state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Agent: Summarization ---")

state["current\_task\_description"] = "Generating summary of the conversation."

if state["error\_log"] or not state.get("full\_transcript\_text"):

state["error\_log"].append("Skipping summarization.")

return state

transcript = state["full\_transcript\_text"]

prompt = (

"You are an expert in summarizing conversations. Provide a concise, abstractive summary of the following transcript. "

"Capture the main points, decisions, and outcomes. The summary should be a single paragraph of about 3-5 sentences. "

"Respond ONLY with the summary text. Do not add any preamble like 'Here is the summary:' or any other extra text.\n\n"

f"Transcript to summarize:\n```\n{transcript}\n```"

)

messages = [HumanMessage(content=prompt)]

try:

print("Sending summarization request to GPT-4o endpoint...")

response = self.llm.invoke(messages)

summary = response.content.strip()

state["summary\_text"] = summary

print(f"Generated summary: {summary}")

except Exception as e:

error\_msg = f"Error during summarization: {str(e)}"

print(error\_msg)

state["error\_log"].append(error\_msg)

return state

# --- Node for Final Output Formatting ---

def format\_final\_output(state: ComprehensiveGraphState) -> ComprehensiveGraphState:

print("\n--- Node: Format Final Output ---")

state["current\_task\_description"] = "Formatting final JSON output."

# Using model\_dump for Pydantic models to get dict representations

final\_json = {

"original\_audio\_path": state.get("original\_audio\_path"),

"audio\_duration\_seconds": state.get("audio\_duration\_seconds"),

"user\_metadata": state.get("user\_metadata").model\_dump() if state.get("user\_metadata") else None,

"full\_transcript\_text": state.get("full\_transcript\_text"),

"transcription\_segments": [seg.model\_dump() for seg in state.get("transcription\_segments", [])] if state.get("transcription\_segments") else None,

"speaker\_timestamps": [st.model\_dump() for st in state.get("speaker\_timestamps", [])] if state.get("speaker\_timestamps") else None,

"identified\_pii": [pii.model\_dump() for pii in state.get("identified\_pii", [])] if state.get("identified\_pii") else None,

"domain\_specific\_data": [dd.model\_dump() for dd in state.get("domain\_specific\_data", [])] if state.get("domain\_specific\_data") else None,

"summary\_text": state.get("summary\_text"),

"processing\_status": "Completed with errors" if state.get("error\_log") else "Completed successfully",

"errors": state.get("error\_log") if state.get("error\_log") else None

}

state["final\_output\_json"] = final\_json

print("Final output JSON compiled.")

return state

# --- Conditional Logic / Routers ---

def route\_after\_transcription(state: ComprehensiveGraphState) -> str:

"""Decides if transcription quality is OK to proceed."""

if not state.get("transcription\_quality\_ok", False) or not state.get("full\_transcript\_text"):

print("Transcription quality low or no transcript, routing to output formatting.")

return "format\_output" # Go directly to formatting if transcription failed badly

return "speaker\_diarizer"

def route\_after\_diarization(state: ComprehensiveGraphState) -> str:

"""Sets flags for PII and Domain analysis based on user\_metadata."""

state["run\_pii\_detection"] = state["user\_metadata"].redact\_pii # or any other logic to run PII

state["run\_domain\_analysis"] = bool(state["user\_metadata"].domains\_of\_interest) or True # Example: always run domain or if specified

if state["run\_pii\_detection"]:

return "pii\_detector"

elif state["run\_domain\_analysis"]:

return "domain\_analyzer"

else:

return "summarizer" # Skip to summarizer if neither PII nor Domain is needed

def route\_after\_pii(state: ComprehensiveGraphState) -> str:

if state["run\_domain\_analysis"]:

return "domain\_analyzer"

return "summarizer"

# --- Build the Graph ---

# Instantiate agents

audio\_loader = AudioLoaderPreparer()

transcriber\_agent = AudioTranscriberAgent()

diarizer\_agent = SpeakerDiarizerAgent()

pii\_agent = PIIDetectorAgent()

domain\_agent = DomainAnalyzerAgent()

summarizer\_agent = SummarizationAgent()

# Define the graph

workflow = StateGraph(ComprehensiveGraphState)

workflow.add\_node("audio\_loader", audio\_loader.execute)

workflow.add\_node("transcriber", transcriber\_agent.execute)

# Placeholder for QualityCheckNode - for now, decision is in route\_after\_transcription

workflow.add\_node("speaker\_diarizer", diarizer\_agent.execute)

workflow.add\_node("pii\_detector", pii\_agent.execute)

workflow.add\_node("domain\_analyzer", domain\_agent.execute)

workflow.add\_node("summarizer", summarizer\_agent.execute)

workflow.add\_node("format\_output", format\_final\_output)

# Set entry point

workflow.set\_entry\_point("audio\_loader")

# Define edges based on the flow

workflow.add\_edge("audio\_loader", "transcriber")

# Conditional edge after transcription

workflow.add\_conditional\_edges(

"transcriber",

route\_after\_transcription,

{

"speaker\_diarizer": "speaker\_diarizer",

"format\_output": "format\_output" # If transcription failed

}

)

workflow.add\_conditional\_edges(

"speaker\_diarizer",

route\_after\_diarization,

{

"pii\_detector": "pii\_detector",

"domain\_analyzer": "domain\_analyzer", # If PII is skipped but domain is not

"summarizer": "summarizer" # If both PII and Domain are skipped

}

)

workflow.add\_conditional\_edges(

"pii\_detector",

route\_after\_pii, # Decides if domain analysis runs next or skip to summarizer

{

"domain\_analyzer": "domain\_analyzer",

"summarizer": "summarizer"

}

)

workflow.add\_edge("domain\_analyzer", "summarizer") # Domain always goes to Summarizer

workflow.add\_edge("summarizer", "format\_output")

workflow.add\_edge("format\_output", END)

# Compile the graph (add checkpointing if needed for resilience)

# memory = SqliteSaver.from\_conn\_string(":memory:") # Example for in-memory checkpointing

# app = workflow.compile(checkpointer=memory)

app = workflow.compile()

# --- Main Execution ---

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

audio\_file\_path = r"C:\doc\_audio\_project\audio\_files\Banking Example Audio.mp3" # Replace with your actual path

# audio\_file\_path = r"C:\doc\_audio\_project\audio\_files\short\_example.mp3" # Use a short audio for faster testing

if not os.path.exists(audio\_file\_path):

print(f"ERROR: Audio file not found at {audio\_file\_path}")

exit()

initial\_user\_metadata = UserMetadata(

domains\_of\_interest=["finance", "customer\_service"],

redact\_pii=True # Set to True to trigger PII agent if routing allows

)

initial\_state = ComprehensiveGraphState(

original\_audio\_path=audio\_file\_path,

audio\_duration\_seconds=None,

user\_metadata=initial\_user\_metadata,

base64\_audio=None,

transcription\_segments=None,

full\_transcript\_text=None,

speaker\_timestamps=None,

identified\_pii=None,

domain\_specific\_data=None,

summary\_text=None,

error\_log=[],

final\_output\_json=None,

current\_task\_description="Starting workflow",

transcription\_quality\_ok=True, # Initial assumption

run\_pii\_detection=False, # Will be set by router

run\_domain\_analysis=False # Will be set by router

)

print("Starting LangGraph Audio Processing Workflow...")

config = {"recursion\_limit": 25} # Standard config

final\_state\_result = None

for event in app.stream(initial\_state, config=config):

for node\_name, state\_update in event.items():

print(f"\nOutput from node: '{node\_name}'")

# print(f"Current State after {node\_name}:")

# print(json.dumps(state\_update, indent=2, default=str)) # Can be very verbose

print(f" Task: {state\_update.get('current\_task\_description', 'N/A')}")

if state\_update.get('error\_log'):

print(f" Errors: {state\_update['error\_log']}")

final\_state\_result = state\_update # Keep the latest state

print("\n\n--- Workflow Execution Complete ---")

if final\_state\_result and final\_state\_result.get("final\_output\_json"):

print("\n--- Final Compiled Output JSON ---")

print(json.dumps(final\_state\_result["final\_output\_json"], indent=2, default=str))

elif final\_state\_result and final\_state\_result.get("error\_log"):

print("\n--- Workflow completed with errors: ---")

for err in final\_state\_result["error\_log"]:

print(f"- {err}")

print("\n--- Partial State at Finish ---")

# Print a summary of key fields from the state if an error occurred before final formatting

print(f" Transcript (snippet): {str(final\_state\_result.get('full\_transcript\_text'))[:200]}...")

print(f" Summary: {final\_state\_result.get('summary\_text')}")

else:

print("\n--- Workflow completed, but no explicit final\_output\_json or error\_log found in the last state. ---")