Python Source Code Compilation

File: ./test_model.py

```
# test_model.py
from improved_depression_model import ImprovedDepressionModel
from audio_feature_extraction import AudioFeatureProcessor
import pandas as pd

# Load the trained model
print("Loading trained model...")
model = ImprovedDepressionModel()
model.load_model('models/depression_model_improved.pkl')

print("\n" + "="*60)
print("MODEL LOADED SUCCESSFULLY")
print("="*60)
print(f"Selected features: {len(model.selected_features)}")
print(f"Decision threshold: {model.threshold:.3f}")

# Test on a specific participant
def test_participant(participant_id):
    """Test depression detection on a specific participant"""

    processor = AudioFeatureProcessor(
        "data/covarep",
        "data/formant",
        "data/PHQ8_Labels.csv"
    )

    # Load features for this participant
    covarep_features = processor.load_covarep_features(str(participant_id))
    formant_features = processor.load_formant_features(str(participant_id))

    if covarep_features is None or formant_features is None:
        print(f" No audio data found for participant {participant_id}")
        return

    # Combine features
    all_features = {}
    all_features.update(covarep_features)
    all_features.update(formant_features)

    # Predict
    result = model.predict(all_features)

    # Load actual PHQ-8 score
    labels_df = pd.read_csv("data/phq8_labels.csv")
    actual_row = labels_df[labels_df['Participant_ID'] == int(participant_id)]

    print("\n" + "="*60)
    print(f"PREDICTION FOR PARTICIPANT {participant_id}")
    print("="*60)
```

Python Source Code Compilation

```
    if len(actual_row) > 0:
        actual_score = actual_row['PHQ_8Total'].values[0]
        actual_depression = actual_score >= 10
        print(f"\n Actual PHQ-8:")
        print(f"  Score: {actual_score}/24")
        print(f"  Depression: {'YES' if actual_depression else 'NO'}")

    print(f"\n Model Prediction:")
    print(f"  Depression Detected: {'YES' if result['depression_detected'] else 'NO'}")
    print(f"  Probability: {result['probability']:.2%}")
    print(f"  Risk Level: {result['risk_level']}")
    print(f"  Confidence: {result['confidence']:.2%}")

    if len(actual_row) > 0:
        correct = (result['depression_detected'] == actual_depression)
        print(f"\n Prediction: {'CORRECT' if correct else 'INCORRECT'}")

    print("="*60)

# Example: Test a few participants
print("\n" + "="*60)
print("TESTING MODEL ON SAMPLE PARTICIPANTS")
print("="*60)

# Test 5 random participants
test_participants = ['300', '301', '302', '303', '304']

for pid in test_participants:
    test_participant(pid)
    print()
```

Python Source Code Compilation

File: ./auth.py

```
import os
from datetime import datetime, timedelta

from fastapi import Depends, HTTPException, Header
from jose import jwt, JWTError
from passlib.context import CryptContext
from sqlalchemy.orm import Session
from dotenv import load_dotenv

from database import SessionLocal
from models import User

# ---------------- LOAD ENV ----------------
load_dotenv()

SECRET_KEY = os.getenv("SECRET_KEY")
if not SECRET_KEY:
    raise RuntimeError("SECRET_KEY not set in .env")

ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_HOURS = 12

# ---------------- PASSWORD HASHING ----------------
pwd_context = CryptContext(
    schemes=["bcrypt"],
    deprecated="auto"
)

def hash_password(password: str) -> str:
    """
    bcrypt has a hard 72-byte limit.
    We normalize + truncate safely.
    """
    password_bytes = password.encode("utf-8")[:72]
    return pwd_context.hash(password_bytes)

def verify_password(password: str, hashed: str) -> bool:
    password_bytes = password.encode("utf-8")[:72]
    return pwd_context.verify(password_bytes, hashed)

# ---------------- DATABASE DEP ----------------
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# ---------------- JWT ----------------
def create_token(user_id: int) -> str:
```

Python Source Code Compilation

```
    payload = {
        "sub": str(user_id),
        "exp": datetime.utcnow() + timedelta(hours=ACCESS_TOKEN_EXPIRE_HOURS)
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def get_current_user(
    authorization: str = Header(None),
    db: Session = Depends(get_db)
):
    if not authorization or not authorization.startswith("Bearer "):
        raise HTTPException(status_code=401, detail="Not authenticated")

    token = authorization.split(" ")[1]

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id = int(payload.get("sub"))
    except JWTError:
        raise HTTPException(status_code=401, detail="Invalid or expired token")

    user = db.query(User).filter(User.id == user_id).first()
    if not user:
        raise HTTPException(status_code=401, detail="User not found")

    return user
```

Python Source Code Compilation

File: ./models.py

```
from sqlalchemy import Column, Integer, String, ForeignKey, Text, DateTime, Float
from sqlalchemy.orm import relationship
from datetime import datetime
from database import Base

# ---------------- USER (UNCHANGED) ----------------
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    conversations = relationship("Conversation", back_populates="user")
    assessments = relationship("PHQ8Assessment", back_populates="user")  # NEW

# ---------------- CONVERSATION ----------------
class Conversation(Base):
    __tablename__ = "conversations"
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, default="New Conversation")
    user_id = Column(Integer, ForeignKey("users.id"))
    created_at = Column(DateTime, default=datetime.utcnow)
    user = relationship("User", back_populates="conversations")
    messages = relationship(
        "Message",
        back_populates="conversation",
        cascade="all, delete-orphan"
    )

# ---------------- MESSAGE ----------------
class Message(Base):
    __tablename__ = "messages"
    id = Column(Integer, primary_key=True, index=True)
    conversation_id = Column(Integer, ForeignKey("conversations.id"))
    role = Column(String)
    content = Column(Text)
    timestamp = Column(DateTime, default=datetime.utcnow)
    conversation = relationship("Conversation", back_populates="messages")

# ---------------- PHQ-8 ASSESSMENT (NEW) ----------------
class PHQ8Assessment(Base):
    __tablename__ = "phq8_assessments"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    conversation_id = Column(Integer, ForeignKey("conversations.id"), nullable=True)

    # Individual scores (0-3 each)
    no_interest = Column(Integer)
    depressed = Column(Integer)
    sleep = Column(Integer)
```

Python Source Code Compilation

```
    tired = Column(Integer)
    appetite = Column(Integer)
    failure = Column(Integer)
    concentrating = Column(Integer)
    moving = Column(Integer)

    # Calculated fields
    total_score = Column(Integer)
    severity = Column(String)
    binary = Column(Integer)  # 0 or 1

    created_at = Column(DateTime, default=datetime.utcnow)

    user = relationship("User", back_populates="assessments")
```

Python Source Code Compilation

File: ./train_model.py

```
# train_model.py
from audio_feature_extraction import AudioFeatureProcessor
from improved_depression_model import ImprovedDepressionModel  # Changed
import pandas as pd
import os

# ===== CONFIGURE THESE PATHS =====
COVAREP_DIR = "data/covarep"
FORMANT_DIR = "data/formant"
LABELS_FILE = "data/phq8_labels.csv"
# ================================

def check_paths():
    """Verify that all paths exist"""
    print("Checking paths...")

    if not os.path.exists(COVAREP_DIR):
        raise FileNotFoundError(f"COVAREP directory not found: {COVAREP_DIR}")

    if not os.path.exists(FORMANT_DIR):
        raise FileNotFoundError(f"Formant directory not found: {FORMANT_DIR}")

    if not os.path.exists(LABELS_FILE):
        raise FileNotFoundError(f"Labels file not found: {LABELS_FILE}")

    # Count files
    covarep_count = len([f for f in os.listdir(COVAREP_DIR) if f.endswith('.csv')])
    formant_count = len([f for f in os.listdir(FORMANT_DIR) if f.endswith('.csv')])

    print(f" COVAREP directory found: {covarep_count} CSV files")
    print(f" Formant directory found: {formant_count} CSV files")
    print(f" Labels file found")
    print()

def main():
    print("="*60)
    print("IMPROVED DEPRESSION DETECTION MODEL TRAINING")
    print("="*60)
    print()

    # Check paths
    check_paths()

    # Step 1: Extract and combine features
    print("STEP 1: Loading and processing audio features...")
    print("-"*60)
    processor = AudioFeatureProcessor(COVAREP_DIR, FORMANT_DIR, LABELS_FILE)
    feature_df = processor.create_feature_dataset()

    # Save feature dataset
    os.makedirs('data', exist_ok=True)
    feature_df.to_csv('data/audio_features.csv', index=False)
    print(f"\n Feature dataset saved to data/audio_features.csv")

    # Step 2: Train improved model
    print("\n" + "="*60)
    print("STEP 2: Training improved model with feature selection...")
    print("-"*60)

    model = ImprovedDepressionModel()

    try:
        # Train with 100 best features (reduced from 27,975)
        metrics = model.train(feature_df, n_features=100)
    except Exception as e:
        print(f"\n Training failed: {e}")
        import traceback
        traceback.print_exc()
        return

    # Step 3: Save model
    print("\n" + "="*60)
    print("STEP 3: Saving trained model...")
    print("-"*60)
    os.makedirs('models', exist_ok=True)
    model.save_model('models/depression_model_improved.pkl')

    print("\n" + "="*60)
    print(" TRAINING COMPLETE!")
    print("="*60)
    print(f"  ROC-AUC Score: {metrics['roc_auc']:.4f}")
    print(f"  Test Accuracy (standard): {metrics['test_score']:.4f}")
    print(f"  Test Accuracy (optimized): {metrics['test_score_optimal']:.4f}")
    print(f"  Optimal Threshold: {metrics['optimal_threshold']:.3f}")
    print(f"  Model: models/depression_model_improved.pkl")
    print("="*60)

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"\n Error: {e}")
        import traceback
        traceback.print_exc()
```

Python Source Code Compilation

File: ./database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "sqlite:///./users.db"

engine = create_engine(
    DATABASE_URL,
    connect_args={"check_same_thread": False}
)

SessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    bind=engine
)

Base = declarative_base()
```

Python Source Code Compilation

File: ./__init__.py

File: ./pdf.py

```
import os
from fpdf import FPDF

SOURCE_FOLDER = "."
OUTPUT_PDF = "all_python_files.pdf"

class PDF(FPDF):
    def header(self):
        self.set_font("Courier", "B", 10)
        self.cell(0, 8, "Python Source Code Compilation")
        self.ln(10)

def safe_text(text: str) -> str:
    """Remove unicode that FPDF can't handle"""
    return text.encode("latin-1", "ignore").decode("latin-1")

def add_file_to_pdf(pdf, filepath):
    pdf.add_page()
    pdf.set_font("Courier", size=8)

    # File header
    pdf.write(5, safe_text(f"File: {filepath}\n\n"))

    with open(filepath, "r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            safe_line = safe_text(line.rstrip())
            pdf.write(5, safe_line + "\n")

def main():
    pdf = PDF()
    pdf.set_auto_page_break(auto=True, margin=15)

    for root, dirs, files in os.walk(SOURCE_FOLDER):
        dirs[:] = [d for d in dirs if d not in {".venv", "venv", "__pycache__", ".git"}]

        for file in files:
            if file.endswith(".py"):
                add_file_to_pdf(pdf, os.path.join(root, file))

    pdf.output(OUTPUT_PDF)
    print(f" PDF created successfully: {OUTPUT_PDF}")

if __name__ == "__main__":
    main()
```

Python Source Code Compilation

File: ./phq8.py

```python
# phq8.py
from typing import Dict

def calculate_phq8(responses: Dict[str, int]) -> Dict:
    """
    Calculate PHQ-8 score and severity.

    Args:
        responses: Dict with keys matching PHQ-8 questions (values 0-3)

    Returns:
        Dict with score, severity, and binary classification
    """
    # Sum all responses
    total_score = sum([
        responses.get('no_interest', 0),
        responses.get('depressed', 0),
        responses.get('sleep', 0),
        responses.get('tired', 0),
        responses.get('appetite', 0),
        responses.get('failure', 0),
        responses.get('concentrating', 0),
        responses.get('moving', 0)
    ])

    # Determine severity
    if total_score <= 4:
        severity = "None/Minimal"
    elif total_score <= 9:
        severity = "Mild"
    elif total_score <= 14:
        severity = "Moderate"
    elif total_score <= 19:
        severity = "Moderately Severe"
    else:
        severity = "Severe"

    # Binary classification (10 = depressed)
    binary = 1 if total_score >= 10 else 0

    return {
        'total_score': total_score,
        'severity': severity,
        'binary': binary,
        'depressed': binary == 1
    }

# PHQ-8 Questions
PHQ8_QUESTIONS = [
    {
        'id': 'no_interest',
        'question': 'Little interest or pleasure in doing things?',
        'field': 'no_interest'
    },
    {
        'id': 'depressed',
        'question': 'Feeling down, depressed, or hopeless?',
        'field': 'depressed'
    },
    {
        'id': 'sleep',
        'question': 'Trouble falling or staying asleep, or sleeping too much?',
        'field': 'sleep'
    },
    {
        'id': 'tired',
        'question': 'Feeling tired or having little energy?',
        'field': 'tired'
    },
    {
        'id': 'appetite',
        'question': 'Poor appetite or overeating?',
        'field': 'appetite'
    },
    {
        'id': 'failure',
        'question': 'Feeling bad about yourself or that you are a failure?',
        'field': 'failure'
    },
    {
        'id': 'concentrating',
        'question': 'Trouble concentrating on things?',
        'field': 'concentrating'
    },
    {
        'id': 'moving',
        'question': 'Moving or speaking slowly, or being fidgety/restless?',
        'field': 'moving'
    }
]

PHQ8_OPTIONS = [
    {'value': 0, 'label': 'Not at all'},
    {'value': 1, 'label': 'Several days'},
    {'value': 2, 'label': 'More than half the days'},
    {'value': 3, 'label': 'Nearly every day'}
]
```

Python Source Code Compilation

File: ./depression_model.py

```
# depression_model.py
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
import joblib
import json

class DepressionDetectionModel:
    def __init__(self):
        self.model = None
        self.scaler = StandardScaler()
        self.feature_names = None
        self.threshold = 0.5

    def prepare_data(self, feature_df):
        """Prepare data for training with robust cleaning"""
        # Separate features and labels
        X = feature_df.drop(['participant_id', 'phq8_score', 'depression'], axis=1)
        y = feature_df['depression']

        # Handle infinity values
        X = X.replace([np.inf, -np.inf], np.nan)

        # Handle missing values by filling with column mean
        X = X.fillna(X.mean())

        # If any columns are still all NaN, fill with 0
        X = X.fillna(0)

        # Verify no infinite values remain
        if not np.isfinite(X.values).all():
            print("  Warning: Some features still contain non-finite values")
            print("Replacing remaining non-finite values with 0...")
            X = X.replace([np.inf, -np.inf, np.nan], 0)

        # Store feature names
        self.feature_names = X.columns.tolist()

        print(f"\n Prepared {len(X)} samples with {len(self.feature_names)} features")

        return X, y

    def train(self, feature_df, test_size=0.2, random_state=42):
        """Train the depression detection model"""
        X, y = self.prepare_data(feature_df)

        # Split data
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=test_size, random_state=random_state, stratify=y
        )

        # Scale features
        X_train_scaled = self.scaler.fit_transform(X_train)
        X_test_scaled = self.scaler.transform(X_test)

        # Train Random Forest
        print("Training Random Forest model...")
        self.model = RandomForestClassifier(
            n_estimators=200,
            max_depth=10,
            min_samples_split=5,
            min_samples_leaf=2,
            class_weight='balanced',
            random_state=random_state,
            n_jobs=-1
        )

        self.model.fit(X_train_scaled, y_train)

        # Evaluate
        y_pred = self.model.predict(X_test_scaled)
        y_pred_proba = self.model.predict_proba(X_test_scaled)[:, 1]

        print("\n=== Model Performance ===")
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred, target_names=['No Depression', 'Depression']))

        print("\nConfusion Matrix:")
        cm = confusion_matrix(y_test, y_pred)
        print(cm)
        print(f"\nTrue Negatives: {cm[0,0]} | False Positives: {cm[0,1]}")
        print(f"False Negatives: {cm[1,0]} | True Positives: {cm[1,1]}")

        print(f"\nROC-AUC Score: {roc_auc_score(y_test, y_pred_proba):.4f}")

        # Cross-validation
        print("\nCross-validation scores:")
        cv_scores = cross_val_score(
            self.model, X_train_scaled, y_train, cv=5, scoring='roc_auc'
        )
        print(f"CV ROC-AUC: {cv_scores.mean():.4f} (+/- {cv_scores.std():.4f})")

        # Feature importance
        self.print_feature_importance()

        return {
            'train_score': self.model.score(X_train_scaled, y_train),
            'test_score': self.model.score(X_test_scaled, y_test),
            'roc_auc': roc_auc_score(y_test, y_pred_proba),
            'cv_scores': cv_scores.tolist()
        }

    def print_feature_importance(self, top_n=20):
        """Print top important features"""
        if self.model is None:
            print("Model not trained yet")
            return

        importances = self.model.feature_importances_
        indices = np.argsort(importances)[::-1]

        print(f"\nTop {top_n} Most Important Features:")
        for i in range(min(top_n, len(indices))):
            idx = indices[i]
            print(f"{i+1}. {self.feature_names[idx]}: {importances[idx]:.4f}")

    def predict(self, features):
        """Predict depression from audio features"""
        if self.model is None:
            raise ValueError("Model not trained yet")

        # Prepare features
        feature_dict = {name: 0.0 for name in self.feature_names}
        feature_dict.update(features)

        X = pd.DataFrame([feature_dict])
        X = X[self.feature_names]  # Ensure correct order

        # Handle missing values
        X = X.fillna(0)

        # Scale
        X_scaled = self.scaler.transform(X)

        # Predict
        prediction = self.model.predict(X_scaled)[0]
        probability = self.model.predict_proba(X_scaled)[0]

        return {
            'depression_detected': bool(prediction),
            'probability': float(probability[1]),
            'confidence': float(max(probability)),
            'risk_level': self.get_risk_level(probability[1])
        }

    def get_risk_level(self, probability):
        """Convert probability to risk level"""
        if probability < 0.3:
            return "Low"
        elif probability < 0.6:
            return "Moderate"
        else:
            return "High"

    def save_model(self, model_path='models/depression_model.pkl'):
        """Save trained model"""
        import os
        os.makedirs(os.path.dirname(model_path), exist_ok=True)

        model_data = {
            'model': self.model,
            'scaler': self.scaler,
            'feature_names': self.feature_names,
            'threshold': self.threshold
        }

        joblib.dump(model_data, model_path)
        print(f"Model saved to {model_path}")

    def load_model(self, model_path='models/depression_model.pkl'):
        """Load trained model"""
        model_data = joblib.load(model_path)

        self.model = model_data['model']
        self.scaler = model_data['scaler']
        self.feature_names = model_data['feature_names']
        self.threshold = model_data.get('threshold', 0.5)

        print(f"Model loaded from {model_path}")
```

Python Source Code Compilation

File: ./rag.py

```
# rag.py
import os
from dotenv import load_dotenv
from sqlalchemy.orm import Session
from langchain_groq import ChatGroq
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_core.prompts import PromptTemplate
from models import Message, PHQ8Assessment

# ---------------- LOAD ENV ----------------
load_dotenv()
GROQ_API_KEY = os.getenv("GROQ_API_KEY")
CHROMA_DIR = "chroma_db"

# ---------------- EMBEDDINGS ----------------
embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)

# ---------------- VECTOR STORE (DAIC-WOZ) ----------------
vector_store = Chroma(
    persist_directory=CHROMA_DIR,
    embedding_function=embeddings
)

# ---------------- LLM ----------------
llm = ChatGroq(
    model_name="llama-3.1-8b-instant",
    temperature=0.35,
    groq_api_key=GROQ_API_KEY
)

# ---------------- PROMPT WITH PHQ-8 + VOICE AWARENESS ----------------
THERAPIST_PROMPT = PromptTemplate(
    template="""You are a skilled, empathetic mental health conversational assistant trained in active
listening and therapeutic communication.

{phq8_context}

{voice_context}

CORE THERAPEUTIC PRINCIPLES:
1. VALIDATE before you question - acknowledge emotions first
2. EXPLORE depth - move from surface to underlying feelings
3. REFLECT meaning - paraphrase to show understanding
4. AVOID repetition - never ask the same question twice
5. BE PRESENT - respond to what's said, not what you expect
6. REMEMBER context - use conversation history to build continuity
7. ADAPT to clinical context - if PHQ-8 scores or voice analysis are present, respond appropriately
```

Python Source Code Compilation

```
CRITICAL RULES - NEVER VIOLATE:
 NEVER simply echo short responses ("but"  "but")
 NEVER ask "What do you mean by X?" more than once per topic
 NEVER repeat the same question in different words
 NEVER give advice unless explicitly asked
 NEVER diagnose or label
 NEVER say "I understand" without demonstrating it
 NEVER ignore what was said earlier in the conversation
 NEVER minimize PHQ-8 scores or mental health concerns
 NEVER reveal or mention voice analysis to the user directly
 NEVER say "your voice sounds depressed" or reference audio analysis

THERAPEUTIC TECHNIQUES TO USE:

1. VALIDATION + EXPLORATION (for emotional statements):
   User: "People say I have attitude issues"
    Good: "That must feel frustrating to hear. Can you tell me about a specific time when someone said that?"
    Bad: "What do you mean by attitude issues?"

2. REFLECTION (show you're listening):
   User: "The way I speak, my expression and body language"
    Good: "So you're noticing that how you communicate - both verbally and non-verbally - might be coming across differently than you intend?"
    Bad: "What do you mean by body language?"

3. GENTLE PROBING (go deeper):
   User: "Should I change myself so people think I'm normal?"
    Good: "It sounds like you're caught between being yourself and meeting others' expectations. What would staying true to yourself look like?"
    Bad: "What do you mean by normal?"

4. HANDLING SHORT RESPONSES:
   User: "but"
    Good: "I sense there's more you want to say. What's on your mind?"
    Bad: "but"

   User: "yes indeed"
    Good: "Tell me more about that."
    Bad: "yes indeed"

5. CRISIS RESOURCES (when needed):
   If user mentions self-harm, suicide, or scores indicate severe depression:
   - Take it seriously and express concern
   - Provide: 104 Suicide & Crisis Lifeline (call or text)
   - Encourage professional help immediately
   - Don't leave them without resources

RESPONSE STRUCTURE:
1. First sentence: Validate/reflect what they said (use PHQ-8 and voice context if relevant)
2. Second sentence: Explore deeper OR ask ONE specific follow-up
3. Keep responses under 3 sentences unless they share a lot
```

Python Source Code Compilation

```
DAIC-WOZ STYLE EXAMPLES (STYLE REFERENCE ONLY - NOT CONTENT):
{style_examples}

FULL CONVERSATION HISTORY (THIS CONVERSATION ONLY):
{history}

CURRENT USER MESSAGE:
{question}

RESPOND AS A THERAPIST:
- Be warm but professional
- Validate first, explore second
- Reference PHQ-8 scores when relevant
- Use voice context to adjust warmth and urgency (never mention it directly)
- Adapt your approach based on severity level
- One clear direction per response
- Natural, conversational tone
- Show don't tell empathy
- Maintain continuity across the entire conversation
- Take mental health concerns seriously
Human:
""",
    input_variables=["phq8_context", "voice_context", "style_examples", "history", "question"]
)

chain = THERAPIST_PROMPT | llm

# ---------------- HELPER: GET LATEST PHQ-8 FOR THIS CONVERSATION ----------------
def get_latest_phq8_score(conversation_id: int, db: Session):
    """
    Get the most recent PHQ-8 score linked to this specific conversation.
    FIX: We now query strictly by conversation_id (which is always set correctly
    after the main.py fix), so scores never bleed between conversations.
    """
    latest_assessment = (
        db.query(PHQ8Assessment)
        .filter(PHQ8Assessment.conversation_id == conversation_id)
        .order_by(PHQ8Assessment.created_at.desc())
        .first()
    )

    if latest_assessment:
        return {
            'score': latest_assessment.total_score,
            'severity': latest_assessment.severity,
            'binary': latest_assessment.binary
        }
    return None

# ---------------- MAIN RESPONSE FUNCTION ----------------
def get_response(
    message: str,
    conversation_id: int,
    db: Session,
    voice_context: str = ""
) -> str:
    """
    Generates a therapist-style response.

    KEY FIXES:
    - History is fetched AFTER the user message is saved in main.py, so
      we exclude the very last message (the current one) to avoid it
      appearing twice in the prompt.
    - PHQ-8 lookup is strictly scoped to conversation_id.
    - Smart windowing: ALL messages if <=23, else First 3 + Last 20.
    """
    try:
        from phq8_therapeutic_responses import get_phq8_therapeutic_context

        cleaned = message.strip().lower()

        #  FIX: Fetch ALL saved messages for THIS conversation only.
        #    Because main.py now commits the user message before calling
        #    get_response, we exclude the last message (current user msg)
        #    from the history string to prevent duplication in the prompt.
        all_messages = (
            db.query(Message)
            .filter(Message.conversation_id == conversation_id)
            .order_by(Message.timestamp.asc())
            .all()
        )

        # Exclude the very last message  it IS the current user message
        # that was just saved. We pass it separately as {question}.
        history_messages = all_messages[:-1] if all_messages else []

        #  SMART MEMORY WINDOWING
        if len(history_messages) <= 23:
            past_messages = history_messages
        else:
            # Keep first 3 (conversation context) + last 20 (recent flow)
            past_messages = history_messages[:3] + history_messages[-20:]

        # Build conversation history string
        history = ""
        for m in past_messages:
            role = "User" if m.role == "user" else "Assistant"
            history += f"{role}: {m.content}\n"

        #  PHQ-8 CONTEXT  strictly scoped to this conversation
        phq8_data = get_latest_phq8_score(conversation_id, db)

        if phq8_data:
            phq8_context = get_phq8_therapeutic_context(
                phq8_data['score'],
                phq8_data['severity']
            )
        else:
            phq8_context = "No PHQ-8 assessment data available for this conversation."

        #  VOICE CONTEXT FALLBACK
        if not voice_context:
            voice_context = ""

        #  SHORT INPUT HANDLING (only when there's no history at all)
        if len(cleaned.split()) <= 2 and not history.strip():
            return "I'm here to listen. What's on your mind today?"

        #  CRISIS DETECTION
        crisis_keywords = {
            "suicide", "suicidal", "kill myself",
            "want to die", "end it all", "no point living"
        }
        has_crisis_content = any(phrase in cleaned for phrase in crisis_keywords)

        if has_crisis_content:
            return """I'm really concerned about what you're sharing. Your safety is the most important thing
right now.

Please reach out for immediate help:
- Call or text 104 (Suicide & Crisis Lifeline) - available 24/7
- Text "HELLO" to 104 (Crisis Text Line)
- Go to your nearest emergency room
- Call 100 if you're in immediate danger

You don't have to go through this alone. These feelings can get better with the right support. Will you reach
out to one of these resources right now?"""

        #  SMARTER RAG TRIGGER
        emotional_keywords = {
            "feel", "feeling", "felt", "depressed", "anxious", "sad",
            "worried", "scared", "angry", "lonely", "stressed", "overwhelmed",
            "hopeless", "tired", "sleep", "insomnia", "panic", "fear",
            "grief", "loss", "help", "struggle", "struggling", "hurt",
            "pain", "crying", "cry", "attitude", "people", "normal",
            "change", "myself", "behavior", "think", "thought", "thoughts",
            "mind", "issue", "issues", "problem", "suicide", "suicidal",
            "kill", "die", "death", "harm", "hurt myself"
        }

        has_emotional_content = any(
            word in cleaned.split() for word in emotional_keywords
        )
        use_rag = has_emotional_content or len(cleaned.split()) >= 5
```

Python Source Code Compilation

```
        #  DAIC-WOZ STYLE RETRIEVAL
        if use_rag:
            docs = vector_store.similarity_search(message, k=2)
            if docs:
                style_examples = "\n\n".join(d.page_content for d in docs)
            else:
                style_examples = "(No relevant style examples - use neutral supportive tone)"
        else:
            style_examples = "(Short response - use natural conversational tone)"

        #  LLM CALL WITH FULL CONTEXT
        result = chain.invoke({
            "phq8_context": phq8_context,
            "voice_context": voice_context,
            "style_examples": style_examples,
            "history": history.strip() if history.strip() else "(No prior messages in this conversation)",
            "question": message
        })

        return result.content.strip() if result else "I'm here with you."

    except Exception as e:
        import traceback
        print(f"RAG ERROR: {e}")
        print(traceback.format_exc())
        return "I'm here with you. Could you tell me a bit more about what's on your mind?"
```

Python Source Code Compilation

File: ./phq8_therapeutic_responses.py

```
# phq8_therapeutic_responses.py

def get_phq8_therapeutic_context(score: int, severity: str) -> str:
    """
    Generate therapeutic context based on PHQ-8 score.
    This will be added to the prompt to guide the therapist's response.
    """

    if score <= 4:  # None/Minimal
        return """
ASSESSMENT CONTEXT: User has minimal/no depression (PHQ-8: {score}/24)

THERAPEUTIC APPROACH:
- Acknowledge their self-awareness in taking the assessment
- Validate that low scores don't mean absence of struggles
- Encourage preventive self-care and emotional awareness
- Normalize checking in with mental health
- Be supportive but don't minimize if they express concerns
- Focus on maintaining wellbeing and building resilience

TONE: Encouraging, validating, preventive
AVOID: Dismissing concerns, implying "nothing is wrong"
""".format(score=score)

    elif score <= 9:  # Mild
        return """
ASSESSMENT CONTEXT: User has mild depression (PHQ-8: {score}/24)

THERAPEUTIC APPROACH:
- Validate that "mild" doesn't mean their experience isn't real or important
- Explore what specific symptoms are present
- Discuss self-care strategies and coping mechanisms
- Gently assess if symptoms are affecting daily functioning
- Monitor for worsening without being alarmist
- Encourage healthy habits (sleep, exercise, social connection)
- Be supportive of seeking help if desired, but don't pressure

TONE: Supportive, practical, normalizing
AVOID: Minimizing their experience, being overly clinical
""".format(score=score)

    elif score <= 14:  # Moderate
        return """
ASSESSMENT CONTEXT: User has moderate depression (PHQ-8: {score}/24)

THERAPEUTIC APPROACH:
- Validate the significance of their struggles
- Express genuine concern while avoiding alarm
- Gently recommend considering professional support
- Explore what's been most difficult lately
```

Python Source Code Compilation

```
- Ask about support systems and resources available
- Discuss both immediate coping and longer-term help
- Be present with their pain without rushing to "fix"
- Normalize therapy/counseling as a helpful step

TONE: Warm, concerned, supportive, gently directive
AVOID: Being overly clinical, creating panic, judging
""".format(score=score)

    elif score <= 19:  # Moderately Severe
        return """
ASSESSMENT CONTEXT: User has moderately severe depression (PHQ-8: {score}/24)

THERAPEUTIC APPROACH:
- Express clear concern while maintaining hope
- Strongly encourage professional help (therapist, counselor, doctor)
- Ask about safety and support systems
- Validate the courage it took to complete assessment
- Be direct about the importance of seeking help
- Offer to discuss barriers to getting help
- Provide resources if they're open to it
- Balance urgency with compassion

TONE: Caring, concerned, gently firm, hopeful
AVOID: Creating panic, being judgmental, minimizing severity
SAFETY CHECK: If user mentions self-harm thoughts, provide crisis resources
""".format(score=score)

    else:  # Severe (20-24)
        return """
ASSESSMENT CONTEXT: User has severe depression (PHQ-8: {score}/24) - HIGH CONCERN

THERAPEUTIC APPROACH:
- Express clear, compassionate concern
- STRONGLY recommend immediate professional help
- Assess safety - ask directly about self-harm thoughts if appropriate
- Emphasize that help is available and effective
- Validate how hard things must be right now
- Be directive while remaining compassionate
- Provide crisis resources
- Don't leave them alone - encourage reaching out to someone trusted

TONE: Deeply caring, urgent but not panicked, hopeful, firm
AVOID: Minimizing, being casual, overwhelming with information

CRITICAL: If ANY mention of self-harm/suicide:
- Take it seriously
- Encourage immediate professional contact
- Provide crisis hotline: 988 Suicide & Crisis Lifeline
- Suggest emergency room if in immediate danger

SAFETY IS PRIORITY
```

Python Source Code Compilation

```
""".format(score=score)



def get_phq8_follow_up_prompt(score: int, severity: str) -> str:
    """
    Generate the initial follow-up message after PHQ-8 completion.
    """

    if score <= 4:  # None/Minimal
        return f"""I just took a PHQ-8 assessment and my score is {score} out of 24, which indicates minimal or no depression. I'm not sure what to make of this result. Can you help me understand what this means?"""

    elif score <= 9:  # Mild
        return f"""I just took a PHQ-8 assessment and my score is {score} out of 24, which indicates mild depression. I'm feeling a bit concerned about this. What does this mean for me?"""

    elif score <= 14:  # Moderate
        return f"""I just took a PHQ-8 assessment and my score is {score} out of 24, which indicates moderate depression. I'm worried about what this means. Can you help me understand what I should do?"""

    elif score <= 19:  # Moderately Severe
        return f"""I just took a PHQ-8 assessment and my score is {score} out of 24, which indicates moderately severe depression. I'm really concerned about this result. What should I do?"""

    else:  # Severe
        return f"""I just took a PHQ-8 assessment and my score is {score} out of 24, which indicates severe depression. I'm very worried and don't know what to do. Can you help me?"""
```

Python Source Code Compilation

File: ./improved_depression_model.py

```
# improved_depression_model.py
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve
import joblib
import matplotlib.pyplot as plt

class ImprovedDepressionModel:
    def __init__(self):
        self.model = None
        self.scaler = StandardScaler()
        self.feature_selector = None
        self.feature_names = None
        self.selected_features = None
        self.threshold = 0.5

    def prepare_data(self, feature_df, n_features=100):
        """Prepare data with feature selection"""
        # Separate features and labels
        X = feature_df.drop(['participant_id', 'phq8_score', 'depression'], axis=1)
        y = feature_df['depression']

        # Clean data
        X = X.replace([np.inf, -np.inf], np.nan)
        X = X.fillna(X.mean())
        X = X.fillna(0)

        # Store original feature names
        self.feature_names = X.columns.tolist()

        print(f"\n Data Preparation:")
        print(f"   Original features: {len(self.feature_names)}")
        print(f"   Samples: {len(X)}")
        print(f"   Target distribution: {y.value_counts().to_dict()}")

        # Feature selection - keep top N most relevant features
        print(f"\n Selecting top {n_features} features...")
        self.feature_selector = SelectKBest(score_func=f_classif, k=n_features)
        X_selected = self.feature_selector.fit_transform(X, y)

        # Get selected feature names
        selected_indices = self.feature_selector.get_support(indices=True)
        self.selected_features = [self.feature_names[i] for i in selected_indices]

        print(f"   Reduced to {len(self.selected_features)} features")
        return pd.DataFrame(X_selected, columns=self.selected_features), y

    def train(self, feature_df, test_size=0.2, random_state=42, n_features=100):
        """Train improved model with feature selection"""

        X, y = self.prepare_data(feature_df, n_features=n_features)

        # Split data with stratification
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=test_size, random_state=random_state, stratify=y
        )

        print(f"\n Train/Test Split:")
        print(f"  Training: {len(X_train)} samples")
        print(f"  Testing: {len(X_test)} samples")

        # Scale features
        X_train_scaled = self.scaler.fit_transform(X_train)
        X_test_scaled = self.scaler.transform(X_test)

        # Train with better hyperparameters for small dataset
        print("\n Training Random Forest model...")
        self.model = RandomForestClassifier(
            n_estimators=100,
            max_depth=5,  # Reduced to prevent overfitting
            min_samples_split=10,
            min_samples_leaf=5,
            max_features='sqrt',
            class_weight='balanced',  # Handle class imbalance
            random_state=random_state,
            n_jobs=-1
        )

        self.model.fit(X_train_scaled, y_train)

        # Predictions
        y_pred = self.model.predict(X_test_scaled)
        y_pred_proba = self.model.predict_proba(X_test_scaled)[:, 1]

        # Adjust decision threshold for better recall
        # Find optimal threshold
        fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
        optimal_idx = np.argmax(tpr - fpr)
        optimal_threshold = thresholds[optimal_idx]

        print(f"\n  Optimal decision threshold: {optimal_threshold:.3f} (default: 0.5)")

        # Predict with optimal threshold
        y_pred_optimal = (y_pred_proba >= optimal_threshold).astype(int)

        # Evaluation
        print("\n" + "="*60)
```

Python Source Code Compilation

```
        print("MODEL PERFORMANCE - Standard Threshold (0.5)")
        print("="*60)
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred, target_names=['No Depression', 'Depression']))

        print("\nConfusion Matrix:")
        cm = confusion_matrix(y_test, y_pred)
        print(cm)
        print(f"True Negatives: {cm[0,0]} | False Positives: {cm[0,1]}")
        print(f"False Negatives: {cm[1,0]} | True Positives: {cm[1,1]}")

        # With optimal threshold
        print("\n" + "="*60)
        print(f"MODEL PERFORMANCE - Optimal Threshold ({optimal_threshold:.3f})")
        print("="*60)
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred_optimal, target_names=['No Depression', 'Depression']))

        print("\nConfusion Matrix:")
        cm_opt = confusion_matrix(y_test, y_pred_optimal)
        print(cm_opt)
        print(f"True Negatives: {cm_opt[0,0]} | False Positives: {cm_opt[0,1]}")
        print(f"False Negatives: {cm_opt[1,0]} | True Positives: {cm_opt[1,1]}")

        print(f"\n Metrics:")
        print(f"   ROC-AUC Score: {roc_auc_score(y_test, y_pred_proba):.4f}")
        print(f"   Accuracy (0.5): {self.model.score(X_test_scaled, y_test):.4f}")
        print(f"   Accuracy (optimal): {(y_pred_optimal == y_test).mean():.4f}")

        # Cross-validation with stratified folds
        print("\n Cross-validation (5-fold):")
        cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=random_state)
        cv_scores = cross_val_score(
            self.model, X_train_scaled, y_train, cv=cv, scoring='roc_auc'
        )
        print(f"  ROC-AUC: {cv_scores.mean():.4f} (+/- {cv_scores.std():.4f})")

        # Feature importance
        self.print_feature_importance()

        # Store optimal threshold
        self.threshold = optimal_threshold

        return {
            'train_score': self.model.score(X_train_scaled, y_train),
            'test_score': self.model.score(X_test_scaled, y_test),
            'test_score_optimal': (y_pred_optimal == y_test).mean(),
            'roc_auc': roc_auc_score(y_test, y_pred_proba),
            'cv_scores': cv_scores.tolist(),
            'optimal_threshold': optimal_threshold
        }
```

Python Source Code Compilation

```python
    def print_feature_importance(self, top_n=20):
        """Print top important features"""
        if self.model is None:
            print("Model not trained yet")
            return

        importances = self.model.feature_importances_
        indices = np.argsort(importances)[::-1]

        print(f"\n Top {top_n} Most Important Features:")
        for i in range(min(top_n, len(indices))):
            idx = indices[i]
            print(f"  {i+1}. {self.selected_features[idx]}: {importances[idx]:.4f}")

    def predict(self, features):
        """Predict depression from audio features"""
        if self.model is None:
            raise ValueError("Model not trained yet")

        # Prepare features - use selected features only
        feature_dict = {name: 0.0 for name in self.feature_names}
        feature_dict.update(features)

        X = pd.DataFrame([feature_dict])
        X = X[self.feature_names]

        # Apply feature selection
        X_selected = self.feature_selector.transform(X)
        X_selected = pd.DataFrame(X_selected, columns=self.selected_features)

        # Scale
        X_scaled = self.scaler.transform(X_selected)

        # Predict with optimal threshold
        probability = self.model.predict_proba(X_scaled)[0]
        prediction = (probability[1] >= self.threshold).astype(int)

        return {
            'depression_detected': bool(prediction),
            'probability': float(probability[1]),
            'confidence': float(max(probability)),
            'risk_level': self.get_risk_level(probability[1]),
            'threshold_used': float(self.threshold)
        }

    def get_risk_level(self, probability):
        """Convert probability to risk level"""
        if probability < 0.3:
            return "Low"
        elif probability < 0.6:
            return "Moderate"
        else:
            return "High"

    def save_model(self, model_path='models/depression_model_improved.pkl'):
        """Save trained model"""
        import os
        os.makedirs(os.path.dirname(model_path), exist_ok=True)

        model_data = {
            'model': self.model,
            'scaler': self.scaler,
            'feature_selector': self.feature_selector,
            'feature_names': self.feature_names,
            'selected_features': self.selected_features,
            'threshold': self.threshold
        }

        joblib.dump(model_data, model_path)
        print(f"\n Model saved to {model_path}")

    def load_model(self, model_path='models/depression_model_improved.pkl'):
        """Load trained model"""
        model_data = joblib.load(model_path)

        self.model = model_data['model']
        self.scaler = model_data['scaler']
        self.feature_selector = model_data['feature_selector']
        self.feature_names = model_data['feature_names']
        self.selected_features = model_data['selected_features']
        self.threshold = model_data.get('threshold', 0.5)

        print(f" Model loaded from {model_path}")
```

Python Source Code Compilation

File: ./voice_processor.py

```
# voice_processor.py
"""
Full Python voice pipeline:
- Receives raw audio bytes from frontend (webm/wav/ogg)
- Converts to WAV using ffmpeg via pydub
- Transcribes using OpenAI Whisper (runs locally, no API key needed)
- Returns clean transcript + audio metadata
"""

import os
import io
import tempfile
import traceback
import numpy as np

#  Whisper (local STT)
try:
    import whisper
    _WHISPER_MODEL = None  # lazy-loaded on first use

    def _get_whisper():
        global _WHISPER_MODEL
        if _WHISPER_MODEL is None:
            print("Loading Whisper model (base.en)")
            _WHISPER_MODEL = whisper.load_model("base.en")
            print(" Whisper model loaded")
        return _WHISPER_MODEL

    WHISPER_AVAILABLE = True
except ImportError:
    WHISPER_AVAILABLE = False
    print("  whisper not installed  run: pip install openai-whisper")

#  Audio conversion (pydub + ffmpeg)
try:
    from pydub import AudioSegment
    PYDUB_AVAILABLE = True
except ImportError:
    PYDUB_AVAILABLE = False
    print("  pydub not installed  run: pip install pydub")

#  soundfile fallback
try:
    import soundfile as sf
    SF_AVAILABLE = True
except ImportError:
    SF_AVAILABLE = False



class VoiceProcessor:
```

Python Source Code Compilation

```
    """
    Handles audio-to-text transcription entirely in Python.
    Supports: webm, ogg, wav, mp4, m4a from browser MediaRecorder.
    """

    # Minimum audio length to attempt transcription (seconds)
    MIN_DURATION_SEC = 0.5

    def __init__(self):
        self.whisper_available = WHISPER_AVAILABLE
        self.pydub_available   = PYDUB_AVAILABLE

    #  CONVERT ANY AUDIO FORMAT  16kHz MONO WAV
    def _to_wav_path(self, audio_bytes: bytes, src_suffix: str = ".webm") -> str | None:
        """
        Save audio_bytes to a temp file, convert to 16kHz mono WAV,
        return path to WAV file. Caller must delete the temp files.
        """
        # Write input to temp file
        with tempfile.NamedTemporaryFile(
            suffix=src_suffix, delete=False
        ) as src_f:
            src_f.write(audio_bytes)
            src_path = src_f.name

        wav_path = src_path.replace(src_suffix, "_converted.wav")

        try:
            if self.pydub_available:
                # pydub handles webm/ogg/mp4/wav automatically via ffmpeg
                audio = AudioSegment.from_file(src_path)
                audio = audio.set_channels(1).set_frame_rate(16000)
                audio.export(wav_path, format="wav")
                return wav_path
            else:
                # Fallback: assume it's already a WAV
                import shutil
                shutil.copy(src_path, wav_path)
                return wav_path
        except Exception as e:
            print(f"Audio conversion error: {e}")
            return None
        finally:
            # Clean up source temp file
            try:
                os.remove(src_path)
            except Exception:
                pass

    #  DETECT FORMAT FROM MAGIC BYTES
    @staticmethod
    def _detect_format(audio_bytes: bytes) -> str:
        """Detect audio container format from magic bytes."""
        if audio_bytes[:4] == b'RIFF':
            return ".wav"
        if audio_bytes[:4] == b'OggS':
            return ".ogg"
        if audio_bytes[:3] == b'ID3' or audio_bytes[:2] == b'\xff\xfb':
            return ".mp3"
        if audio_bytes[4:8] == b'ftyp':
            return ".mp4"
        # Default: webm (most browsers use MediaRecorder with webm)
        return ".webm"

    #  TRANSCRIBE
    def transcribe(self, audio_bytes: bytes) -> dict:
        """
        Transcribe audio bytes to text using Whisper.

        Returns:
            {
                "success": bool,
                "transcript": str,
                "language": str,
                "duration_sec": float,
                "error": str | None
            }
        """
        if not self.whisper_available:
            return {
                "success": False,
                "transcript": "",
                "language": "en",
                "duration_sec": 0.0,
                "error": "Whisper not installed. Run: pip install openai-whisper"
            }

        if not audio_bytes or len(audio_bytes) < 500:
            return {
                "success": False,
                "transcript": "",
                "language": "en",
                "duration_sec": 0.0,
                "error": "Audio too short or empty"
            }

        wav_path = None
        try:
            fmt      = self._detect_format(audio_bytes)
            wav_path = self._to_wav_path(audio_bytes, src_suffix=fmt)

            if not wav_path or not os.path.exists(wav_path):
                return {
                    "success": False,
```

```python
                    "transcript": "",
                    "language": "en",
                    "duration_sec": 0.0,
                    "error": "Audio conversion failed"
                }

            # Check duration
            duration = self._get_duration(wav_path)
            if duration < self.MIN_DURATION_SEC:
                return {
                    "success": False,
                    "transcript": "",
                    "language": "en",
                    "duration_sec": duration,
                    "error": f"Audio too short ({duration:.2f}s). Please speak for at least 0.5s."
                }

            # Transcribe with Whisper
            model  = _get_whisper()
            result = model.transcribe(
                wav_path,
                language="en",          # force English  change if multilingual needed
                task="transcribe",
                fp16=False,             # use fp32 for CPU compatibility
                verbose=False
            )

            transcript = result.get("text", "").strip()
            language   = result.get("language", "en")

            if not transcript:
                return {
                    "success": False,
                    "transcript": "",
                    "language": language,
                    "duration_sec": duration,
                    "error": "No speech detected in audio"
                }

            print(f" Whisper transcript ({duration:.1f}s): \"{transcript}\"")

            return {
                "success": True,
                "transcript": transcript,
                "language": language,
                "duration_sec": duration,
                "error": None
            }

        except Exception as e:
            print(f"Transcription error: {e}")
            print(traceback.format_exc())
            return {
                "success": False,
                "transcript": "",
                "language": "en",
                "duration_sec": 0.0,
                "error": str(e)
            }
        finally:
            # Always clean up WAV temp file
            if wav_path and os.path.exists(wav_path):
                try:
                    os.remove(wav_path)
                except Exception:
                    pass

    #  GET AUDIO DURATION
    def _get_duration(self, wav_path: str) -> float:
        """Get duration of WAV file in seconds."""
        try:
            if self.pydub_available:
                audio = AudioSegment.from_wav(wav_path)
                return len(audio) / 1000.0
            elif SF_AVAILABLE:
                info = sf.info(wav_path)
                return info.duration
            else:
                return 1.0  # assume ok if we can't check
        except Exception:
            return 1.0


#  Global instance
voice_processor = VoiceProcessor()
```

Python Source Code Compilation

File: ./main.py

```
# main.py
from fastapi import FastAPI, Depends, HTTPException
from pydantic import BaseModel
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session
from typing import Optional
import base64
import os

from database import Base, engine
from models import User, Conversation, Message, PHQ8Assessment
from auth import (
    get_db, hash_password, verify_password,
    create_token, get_current_user
)
from rag import get_response
from phq8 import calculate_phq8, PHQ8_QUESTIONS, PHQ8_OPTIONS
from phq8_therapeutic_responses import get_phq8_follow_up_prompt
from voice_depression_detector import voice_detector
from voice_processor import voice_processor          #  NEW: Python STT

# ---------------- APP ----------------
app = FastAPI()

# ---------------- DB INIT ----------------
Base.metadata.create_all(bind=engine)

# ---------------- CORS ----------------
app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        "http://localhost:5501", "http://127.0.0.1:5501",
        "http://localhost:5500", "http://127.0.0.1:5500",
        "http://localhost:3000", "http://127.0.0.1:3000",
        "null",
    ],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# ---------------- SCHEMAS ----------------
class Signup(BaseModel):
    email: str
    password: str

class Login(BaseModel):
    email: str
    password: str
```

Python Source Code Compilation

```
class Chat(BaseModel):
    message: str

class PHQ8Response(BaseModel):
    no_interest: int
    depressed: int
    sleep: int
    tired: int
    appetite: int
    failure: int
    concentrating: int
    moving: int

class ConversationUpdate(BaseModel):
    title: str

class VoiceAudio(BaseModel):
    audio_data: str                    # base64 encoded audio (webm/wav/ogg)
    conversation_id: int               # active conversation
    transcript: Optional[str] = None   # optional  Python STT used if omitted

# ---------------- ROUTES ----------------
@app.get("/")
def home():
    return {"status": "Mental Health Chatbot Running"}

# ==================== AUTH ====================

@app.post("/signup")
def signup(data: Signup, db: Session = Depends(get_db)):
    if db.query(User).filter(User.email == data.email).first():
        raise HTTPException(status_code=400, detail="User already exists")
    user = User(email=data.email, hashed_password=hash_password(data.password))
    db.add(user)
    db.commit()
    return {"message": "User created"}

@app.post("/login")
def login(data: Login, db: Session = Depends(get_db)):
    user = db.query(User).filter(User.email == data.email).first()
    if not user or not verify_password(data.password, user.hashed_password):
        raise HTTPException(status_code=401, detail="Invalid credentials")
    token = create_token(user.id)
    return {"access_token": token}

# ==================== CONVERSATIONS ====================

@app.post("/conversations")
def create_conversation(user=Depends(get_current_user), db: Session = Depends(get_db)):
    convo = Conversation(user_id=user.id)
    db.add(convo)
    db.commit()
    db.refresh(convo)
    return {"conversation_id": convo.id}

@app.get("/conversations")
def list_conversations(user=Depends(get_current_user), db: Session = Depends(get_db)):
    convos = (
        db.query(Conversation)
        .filter(Conversation.user_id == user.id)
        .order_by(Conversation.created_at.desc())
        .all()
    )
    return [{"id": c.id, "title": c.title, "created_at": c.created_at} for c in convos]

@app.get("/conversations/{conversation_id}/messages")
def get_messages(conversation_id: int, user=Depends(get_current_user), db: Session = Depends(get_db)):
    conversation = (
        db.query(Conversation)
        .filter(Conversation.id == conversation_id, Conversation.user_id == user.id)
        .first()
    )
    if not conversation:
        raise HTTPException(status_code=404, detail="Conversation not found")

    messages = (
        db.query(Message)
        .filter(Message.conversation_id == conversation_id)
        .order_by(Message.timestamp.asc())
        .all()
    )
    return [{"role": m.role, "content": m.content, "timestamp": m.timestamp} for m in messages]

@app.put("/conversations/{conversation_id}/title")
def update_conversation_title(
    conversation_id: int, data: ConversationUpdate,
    user=Depends(get_current_user), db: Session = Depends(get_db)
):
    conversation = (
        db.query(Conversation)
        .filter(Conversation.id == conversation_id, Conversation.user_id == user.id)
        .first()
    )
    if not conversation:
        raise HTTPException(status_code=404, detail="Conversation not found")
    conversation.title = data.title
    db.commit()
    return {"message": "Title updated", "title": data.title}

@app.delete("/conversations/{conversation_id}")
def delete_conversation(
    conversation_id: int,
    user=Depends(get_current_user), db: Session = Depends(get_db)
):
    conversation = (
        db.query(Conversation)
        .filter(Conversation.id == conversation_id, Conversation.user_id == user.id)
        .first()
    )
    if not conversation:
        raise HTTPException(status_code=404, detail="Conversation not found")
    db.delete(conversation)
    db.commit()
    return {"message": "Conversation deleted"}

# ==================== CHAT (TEXT) ====================

@app.post("/chat/{conversation_id}")
def chat(
    conversation_id: int, req: Chat,
    user=Depends(get_current_user), db: Session = Depends(get_db)
):
    conversation = (
        db.query(Conversation)
        .filter(Conversation.id == conversation_id, Conversation.user_id == user.id)
        .first()
    )
    if not conversation:
        raise HTTPException(status_code=404, detail="Conversation not found")

    # Save user message BEFORE get_response to fix double-message bug
    db.add(Message(conversation_id=conversation_id, role="user", content=req.message))
    db.commit()

    response = get_response(message=req.message, conversation_id=conversation_id, db=db)

    db.add(Message(conversation_id=conversation_id, role="assistant", content=response))
    db.commit()

    return {"response": response}

# ==================== VOICE CHAT (FULL PYTHON PIPELINE) ====================

@app.post("/voice/chat")
def voice_chat(
    req: VoiceAudio,
    user=Depends(get_current_user), db: Session = Depends(get_db)
):
    """
    Full Python voice pipeline:
    1. Decode base64 audio
    2. Transcribe with Whisper (Python STT  no browser Speech API needed)
    3. Analyze audio for depression markers
    4. RAG therapeutic response
    5. Return response + transcript + voice analysis
    """
    # Verify conversation belongs to user
    conversation = (
        db.query(Conversation)
        .filter(Conversation.id == req.conversation_id, Conversation.user_id == user.id)
        .first()
    )
    if not conversation:
        raise HTTPException(status_code=404, detail="Conversation not found")

    #  Step 1: Decode audio
    try:
        audio_bytes = base64.b64decode(req.audio_data)
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Invalid audio data: {e}")

    #  Step 2: Python STT via Whisper
    transcript = None
    stt_result = voice_processor.transcribe(audio_bytes)

    if stt_result["success"]:
        transcript = stt_result["transcript"]
        print(f" Whisper STT: \"{transcript}\" ({stt_result['duration_sec']:.1f}s)")
    else:
        # Fallback to client-provided transcript if Whisper fails
        if req.transcript and req.transcript.strip():
            transcript = req.transcript.strip()
            print(f" Using client transcript (Whisper error: {stt_result['error']}): \"{transcript}\"")
        else:
            return {
                "response": None,
                "transcript": None,
                "voice_analysis": None,
                "error": stt_result["error"] or "Could not transcribe audio. Please speak clearly and try
again."
            }

    #  Step 3: Depression analysis
    voice_analysis = None
    voice_context  = ""
    try:
        voice_analysis = voice_detector.analyze_audio_bytes(audio_bytes)
        voice_context  = voice_detector.get_therapeutic_context(voice_analysis)
        if voice_analysis and not voice_analysis.get("error"):
            print(f"Voice analysis  Risk: {voice_analysis.get('risk_level')} | "
                  f"Prob: {voice_analysis.get('probability', 0):.2%}")
    except Exception as e:
        print(f"Voice analysis error (non-critical): {e}")

    #  Step 4: Save user message BEFORE get_response
    db.add(Message(conversation_id=req.conversation_id, role="user", content=transcript))
    db.commit()
    #  Step 5: RAG response
    response = get_response(
        message=transcript,
        conversation_id=req.conversation_id,
        db=db,
        voice_context=voice_context
    )

    #  Step 6: Save assistant response
    db.add(Message(conversation_id=req.conversation_id, role="assistant", content=response))
    db.commit()

    #  Step 7: Build voice result
    voice_result = None
    if voice_analysis and not voice_analysis.get("error"):
        prob = voice_analysis.get("probability", 0)
        risk = voice_analysis.get("risk_level", "Low")
        voice_result = {
            "depression_detected": voice_analysis.get("depression_detected", False),
            "probability": round(prob * 100, 1),
            "risk_level": risk,
            "confidence": round(voice_analysis.get("confidence", 0) * 100, 1),
            "badge_color": (
                "#ef4444" if risk == "High"
                else "#f59e0b" if risk == "Moderate"
                else "#10b981"
            )
        }

    return {
        "response": response,
        "transcript": transcript,      # frontend displays this as the user message
        "voice_analysis": voice_result,
        "error": None
    }

# ==================== PHQ-8 ROUTES ====================

@app.get("/phq8/questions")
def get_phq8_questions():
    return {
        "questions": PHQ8_QUESTIONS,
        "options": PHQ8_OPTIONS,
        "instructions": "Over the last 2 weeks, how often have you been bothered by the following problems?"
    }

@app.post("/phq8/submit")
def submit_phq8(
    responses: PHQ8Response,
    conversation_id: int = None,
    user=Depends(get_current_user),
    db: Session = Depends(get_db)
):
    response_dict = responses.dict()
    result = calculate_phq8(response_dict)

    # Create/verify conversation BEFORE saving assessment
    if not conversation_id:
        new_convo = Conversation(user_id=user.id, title="PHQ-8 Assessment")
        db.add(new_convo)
        db.commit()
        db.refresh(new_convo)
        conversation_id = new_convo.id
    else:
        existing = (
            db.query(Conversation)
            .filter(Conversation.id == conversation_id, Conversation.user_id == user.id)
            .first()
        )
        if not existing:
            raise HTTPException(status_code=404, detail="Conversation not found")

    # Save assessment  conversation_id always set
    assessment = PHQ8Assessment(
        user_id=user.id,
        conversation_id=conversation_id,
        no_interest=response_dict['no_interest'],
        depressed=response_dict['depressed'],
        sleep=response_dict['sleep'],
        tired=response_dict['tired'],
        appetite=response_dict['appetite'],
        failure=response_dict['failure'],
        concentrating=response_dict['concentrating'],
        moving=response_dict['moving'],
        total_score=result['total_score'],
        severity=result['severity'],
        binary=result['binary']
    )
    db.add(assessment)
    db.commit()
    db.refresh(assessment)

    interpretation = get_interpretation(result['total_score'])
    result_message = f"""PHQ-8 Assessment Results:

Score: {result['total_score']} out of 24
Severity: {result['severity']}
Status: {'Depression Detected' if result['depressed'] else 'No Depression Detected'}

{interpretation}"""

    db.add(Message(conversation_id=conversation_id, role="user", content="I just completed a PHQ-8
assessment"))
```

```
    db.add(Message(conversation_id=conversation_id, role="assistant", content=result_message))
    db.commit()

    follow_up_prompt = get_phq8_follow_up_prompt(result['total_score'], result['severity'])
    therapeutic_response = get_response(message=follow_up_prompt, conversation_id=conversation_id, db=db)

    db.add(Message(conversation_id=conversation_id, role="assistant", content=therapeutic_response))
    db.commit()

    return {
        "assessment_id": assessment.id,
        "conversation_id": conversation_id,
        "score": result['total_score'],
        "severity": result['severity'],
        "binary": result['binary'],
        "depressed": result['depressed'],
        "interpretation": interpretation,
        "therapeutic_response": therapeutic_response
    }

@app.get("/phq8/history")
def get_phq8_history(user=Depends(get_current_user), db: Session = Depends(get_db)):
    assessments = (
        db.query(PHQ8Assessment)
        .filter(PHQ8Assessment.user_id == user.id)
        .order_by(PHQ8Assessment.created_at.desc())
        .all()
    )
    return [
        {"id": a.id, "score": a.total_score, "severity": a.severity, "binary": a.binary, "date": a.created_at}
        for a in assessments
    ]

@app.get("/phq8/latest")
def get_latest_phq8(user=Depends(get_current_user), db: Session = Depends(get_db)):
    latest = (
        db.query(PHQ8Assessment)
        .filter(PHQ8Assessment.user_id == user.id)
        .order_by(PHQ8Assessment.created_at.desc())
        .first()
    )
    if not latest:
        return {"message": "No assessments found"}
    return {"score": latest.total_score, "severity": latest.severity, "date": latest.created_at}

# ==================== HELPERS ====================

def get_interpretation(score: int) -> str:
    if score <= 4:
        return "Your responses suggest minimal or no depression. Keep taking care of yourself!"
    elif score <= 9:
        return "Your responses suggest mild depression. Consider monitoring your mood and practicing
```

```
self-care."
    elif score <= 14:
        return "Your responses suggest moderate depression. It may be helpful to speak with a mental health
professional."
    elif score <= 19:
        return "Your responses suggest moderately severe depression. I strongly recommend consulting a mental
health professional."
    else:
        return "Your responses suggest severe depression. Please seek professional help as soon as possible. If
you're in crisis, contact a crisis helpline immediately."
```

Python Source Code Compilation

File: ./audio_feature_extraction.py

```
# audio_feature_extraction.py
import pandas as pd
import numpy as np
from pathlib import Path
import os

class AudioFeatureProcessor:
    def __init__(self, covarep_dir, formant_dir, labels_file):
        """
        Initialize the audio feature processor.

        Args:
            covarep_dir: Directory containing COVAREP CSV files
            formant_dir: Directory containing Formant CSV files
            labels_file: Path to PHQ-8 labels CSV
        """
        self.covarep_dir = Path(covarep_dir)
        self.formant_dir = Path(formant_dir)
        self.labels_file = labels_file

    def load_labels(self):
        """Load PHQ-8 labels - Updated for your exact column format"""
        df = pd.read_csv(self.labels_file)

        print(f"Available columns: {df.columns.tolist()}")

        # Your column is PHQ_8Total, not PHQ8_Score
        if 'PHQ_8Total' not in df.columns:
            raise ValueError(f"PHQ_8Total column not found. Available: {df.columns.tolist()}")

        # Create depression label (PHQ-8 >= 10)
        df['depression'] = (df['PHQ_8Total'] >= 10).astype(int)

        # Rename for consistency in the rest of the code
        result = pd.DataFrame({
            'Participant_ID': df['Participant_ID'],
            'PHQ8_Score': df['PHQ_8Total'],  # Map PHQ_8Total to PHQ8_Score
            'depression': df['depression']
        })

        print(f"\nLoaded {len(result)} participants with PHQ-8 labels")
        print(f"Depression cases (PHQ-8 >= 10): {result['depression'].sum()}")
        print(f"Non-depression cases (PHQ-8 < 10): {(~result['depression'].astype(bool)).sum()}")

        return result

    def find_available_participants(self):
        """Find participants who have BOTH COVAREP and Formant data"""
        # Get all COVAREP files
        covarep_files = list(self.covarep_dir.glob("*.csv"))
        covarep_ids = set()

        for file in covarep_files:
            # Extract participant ID from filename
            # Try different patterns: XXX_COVAREP.csv or XXX.csv
            filename = file.stem
            if '_COVAREP' in filename:
                participant_id = filename.replace("_COVAREP", "")
            elif '_covarep' in filename:
                participant_id = filename.replace("_covarep", "")
            else:
                participant_id = filename

            covarep_ids.add(str(participant_id))

        # Get all Formant files
        formant_files = list(self.formant_dir.glob("*.csv"))
        formant_ids = set()

        for file in formant_files:
            # Extract participant ID from filename
            filename = file.stem
            if '_FORMANT' in filename:
                participant_id = filename.replace("_FORMANT", "")
            elif '_formant' in filename:
                participant_id = filename.replace("_formant", "")
            else:
                participant_id = filename

            formant_ids.add(str(participant_id))

        # Find intersection (participants with BOTH)
        available_ids = covarep_ids.intersection(formant_ids)

        print(f"\nAudio file summary:")
        print(f"  Total COVAREP files: {len(covarep_ids)}")
        print(f"  Total Formant files: {len(formant_ids)}")
        print(f"  Participants with BOTH: {len(available_ids)}")

        return available_ids

    def load_covarep_features(self, participant_id):
        """Load COVAREP features for a participant with safety checks"""
        # Try different possible file naming patterns
        possible_files = [
            self.covarep_dir / f"{participant_id}_COVAREP.csv",
            self.covarep_dir / f"{participant_id}.csv",
            self.covarep_dir / f"{participant_id}_covarep.csv",
        ]

        covarep_file = None
        for file_path in possible_files:
            if file_path.exists():
                covarep_file = file_path
                break

        if covarep_file is None:
            return None

        try:
            df = pd.read_csv(covarep_file)

            # Remove non-numeric columns and compute statistics
            numeric_cols = df.select_dtypes(include=[np.number]).columns

            if len(numeric_cols) == 0:
                return None

            # Compute statistical features with safety checks
            features = {}
            for col in numeric_cols:
                # Replace infinity with NaN and drop NaN values
                col_data = df[col].replace([np.inf, -np.inf], np.nan).dropna()

                if len(col_data) > 0:
                    mean_val = col_data.mean()
                    std_val = col_data.std()
                    min_val = col_data.min()
                    max_val = col_data.max()
                    median_val = col_data.median()

                    # Only add if values are finite
                    if np.isfinite(mean_val):
                        features[f'covarep_{col}_mean'] = mean_val
                    if np.isfinite(std_val):
                        features[f'covarep_{col}_std'] = std_val
                    if np.isfinite(min_val):
                        features[f'covarep_{col}_min'] = min_val
                    if np.isfinite(max_val):
                        features[f'covarep_{col}_max'] = max_val
                    if np.isfinite(median_val):
                        features[f'covarep_{col}_median'] = median_val

            return features
        except Exception as e:
            print(f"  Error loading COVAREP for {participant_id}: {e}")
            return None

    def load_formant_features(self, participant_id):
        """Load Formant features for a participant with safety checks"""
        # Try different possible file naming patterns
        possible_files = [
            self.formant_dir / f"{participant_id}_FORMANT.csv",
            self.formant_dir / f"{participant_id}.csv",
            self.formant_dir / f"{participant_id}_formant.csv",
        ]

        formant_file = None
        for file_path in possible_files:
            if file_path.exists():
                formant_file = file_path
                break

        if formant_file is None:
            return None

        try:
            df = pd.read_csv(formant_file)

            # Remove non-numeric columns and compute statistics
            numeric_cols = df.select_dtypes(include=[np.number]).columns

            if len(numeric_cols) == 0:
                return None

            # Compute statistical features with safety checks
            features = {}
            for col in numeric_cols:
                # Replace infinity with NaN and drop NaN values
                col_data = df[col].replace([np.inf, -np.inf], np.nan).dropna()

                if len(col_data) > 0:
                    mean_val = col_data.mean()
                    std_val = col_data.std()
                    min_val = col_data.min()
                    max_val = col_data.max()
                    median_val = col_data.median()

                    # Only add if values are finite
                    if np.isfinite(mean_val):
                        features[f'formant_{col}_mean'] = mean_val
                    if np.isfinite(std_val):
                        features[f'formant_{col}_std'] = std_val
                    if np.isfinite(min_val):
                        features[f'formant_{col}_min'] = min_val
                    if np.isfinite(max_val):
                        features[f'formant_{col}_max'] = max_val
                    if np.isfinite(median_val):
                        features[f'formant_{col}_median'] = median_val

            return features
        except Exception as e:
            print(f"  Error loading Formant for {participant_id}: {e}")
            return None

    def create_feature_dataset(self):
        """Create combined feature dataset"""
        print("="*70)
        print("STEP 1: Loading PHQ-8 labels...")
        print("="*70)
        labels_df = self.load_labels()
        total_labels = len(labels_df)

        print("\n" + "="*70)
        print("STEP 2: Finding participants with audio features...")
        print("="*70)
        available_audio_ids = self.find_available_participants()

        # Filter labels to only include participants with audio
        labels_df['Participant_ID'] = labels_df['Participant_ID'].astype(str)
        available_labels = labels_df[labels_df['Participant_ID'].isin(available_audio_ids)]

        print(f"\nMatching results:")
        print(f"  Participants with labels AND audio: {len(available_labels)}")
        print(f"  Participants excluded (no audio): {total_labels - len(available_labels)}")

        if len(available_labels) == 0:
            raise ValueError(" No participants found with both PHQ-8 labels and audio features!")

        print("\n" + "="*70)
        print("STEP 3: Extracting acoustic features...")
        print("="*70)

        all_features = []
        skipped = []
        processed_count = 0

        for idx, row in available_labels.iterrows():
            participant_id = str(row['Participant_ID'])

            # Progress indicator
            processed_count += 1
            if processed_count % 10 == 0 or processed_count == 1:
                print(f"Processing participant {processed_count}/{len(available_labels)}: {participant_id}")

            # Load features
            covarep_features = self.load_covarep_features(participant_id)
            formant_features = self.load_formant_features(participant_id)

            # Skip if features are missing or invalid
            if covarep_features is None or formant_features is None:
                skipped.append(participant_id)
                continue

            if len(covarep_features) == 0 or len(formant_features) == 0:
                skipped.append(participant_id)
                continue
            # Combine all features
            combined_features = {
                'participant_id': participant_id,
                'phq8_score': row['PHQ8_Score'],
                'depression': row['depression']
            }
            combined_features.update(covarep_features)
            combined_features.update(formant_features)

            all_features.append(combined_features)

        # Create DataFrame
        feature_df = pd.DataFrame(all_features)

        # Clean infinity and NaN values
        print("\nCleaning features...")
        feature_cols = [col for col in feature_df.columns if col not in ['participant_id', 'phq8_score',
'depression']]

        # Replace infinity with NaN
        feature_df[feature_cols] = feature_df[feature_cols].replace([np.inf, -np.inf], np.nan)

        # Fill NaN with column mean
        feature_df[feature_cols] = feature_df[feature_cols].fillna(feature_df[feature_cols].mean())

        # Fill any remaining NaN with 0
        feature_df[feature_cols] = feature_df[feature_cols].fillna(0)

        # Print comprehensive summary
        print("\n" + "="*70)
        print("FEATURE EXTRACTION COMPLETE - SUMMARY")
        print("="*70)
        print(f"\n Data Pipeline Results:")
        print(f"   Total PHQ-8 labels: {total_labels}")
        print(f"   Participants with audio files: {len(available_audio_ids)}")
        print(f"   Successfully processed: {len(feature_df)}")
        print(f"   Skipped (errors/missing): {len(skipped)}")

        print(f"\n Feature Statistics:")
        print(f"   Total features per participant: {len(feature_df.columns) - 3}")
        print(f"   COVAREP features: {sum(1 for col in feature_df.columns if 'covarep' in col)}")
        print(f"   Formant features: {sum(1 for col in feature_df.columns if 'formant' in col)}")

        print(f"\n Depression Distribution:")
        depression_count = feature_df['depression'].sum()
        no_depression_count = len(feature_df) - depression_count
        depression_pct = (depression_count / len(feature_df) * 100) if len(feature_df) > 0 else 0

        print(f"   Depression (PHQ-8 >= 10): {depression_count} ({depression_pct:.1f}%)")
        print(f"   No Depression (PHQ-8 < 10): {no_depression_count} ({100-depression_pct:.1f}%)")

        print(f"\n PHQ-8 Score Statistics:")
```

Python Source Code Compilation

```
        print(f"   Mean: {feature_df['phq8_score'].mean():.2f}")
        print(f"   Median: {feature_df['phq8_score'].median():.2f}")
        print(f"   Std Dev: {feature_df['phq8_score'].std():.2f}")
        print(f"   Range: {feature_df['phq8_score'].min():.0f} - {feature_df['phq8_score'].max():.0f}")

        print("="*70)

        # Warnings
        if len(skipped) > 0:
            print(f"\n  Skipped {len(skipped)} participants due to errors:")
            print(f"   {', '.join(skipped[:10])}")
            if len(skipped) > 10:
                print(f"   ... and {len(skipped) - 10} more")

        if len(feature_df) < 20:
            print("\n  WARNING: Very few samples! Model may not train well.")
            print("   Recommended minimum: 50+ samples")
            print(f"   Current samples: {len(feature_df)}")

        if depression_pct < 20 or depression_pct > 80:
            print(f"\n  WARNING: Class imbalance detected!")
            print(f"   Depression: {depression_pct:.1f}% | No Depression: {100-depression_pct:.1f}%")
            print("   Model may be biased toward majority class")

        print()

        return feature_df
```

Python Source Code Compilation

File: ./voice_depression_detector.py

```
# voice_depression_detector.py
import numpy as np
import librosa
import soundfile as sf
import tempfile
import os
from improved_depression_model import ImprovedDepressionModel

class VoiceDepressionDetector:
    def __init__(self, model_path='models/depression_model_improved.pkl'):
        """Initialize with trained model"""
        try:
            self.model = ImprovedDepressionModel()
            self.model.load_model(model_path)
            self.model_loaded = True
            print(" Voice depression detection model loaded")
        except Exception as e:
            print(f"  Could not load depression model: {e}")
            self.model_loaded = False

    def extract_features_from_audio(self, audio_path):
        """Extract acoustic features from audio file"""
        try:
            y, sr = librosa.load(audio_path, sr=16000)

            if len(y) < sr * 0.5:
                print("  Audio too short for analysis")
                return None

            features = {}

            # ---- PITCH FEATURES ----
            pitches, magnitudes = librosa.piptrack(y=y, sr=sr)
            pitch_values = []
            for t in range(pitches.shape[1]):
                index = magnitudes[:, t].argmax()
                pitch = pitches[index, t]
                if pitch > 0:
                    pitch_values.append(pitch)

            if len(pitch_values) > 0:
                features['covarep_pitch_mean']   = float(np.mean(pitch_values))
                features['covarep_pitch_std']    = float(np.std(pitch_values))
                features['covarep_pitch_min']    = float(np.min(pitch_values))
                features['covarep_pitch_max']    = float(np.max(pitch_values))
                features['covarep_pitch_median'] = float(np.median(pitch_values))

            # ---- ENERGY FEATURES ----
            rms = librosa.feature.rms(y=y)[0]
            features['covarep_energy_mean']   = float(np.mean(rms))
            features['covarep_energy_std']    = float(np.std(rms))
            features['covarep_energy_min']    = float(np.min(rms))
            features['covarep_energy_max']    = float(np.max(rms))
            features['covarep_energy_median'] = float(np.median(rms))

            # ---- SPECTRAL FEATURES ----
            spec_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)[0]
            features['covarep_spectral_centroid_mean']   = float(np.mean(spec_centroid))
            features['covarep_spectral_centroid_std']    = float(np.std(spec_centroid))
            features['covarep_spectral_centroid_min']    = float(np.min(spec_centroid))
            features['covarep_spectral_centroid_max']    = float(np.max(spec_centroid))
            features['covarep_spectral_centroid_median'] = float(np.median(spec_centroid))

            # ---- ZERO CROSSING RATE ----
            zcr = librosa.feature.zero_crossing_rate(y)[0]
            features['covarep_zcr_mean']   = float(np.mean(zcr))
            features['covarep_zcr_std']    = float(np.std(zcr))
            features['covarep_zcr_min']    = float(np.min(zcr))
            features['covarep_zcr_max']    = float(np.max(zcr))
            features['covarep_zcr_median'] = float(np.median(zcr))

            # ---- MFCCs ----
            mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
            for i in range(13):
                features[f'covarep_mfcc{i}_mean']   = float(np.mean(mfccs[i]))
                features[f'covarep_mfcc{i}_std']    = float(np.std(mfccs[i]))
                features[f'covarep_mfcc{i}_min']    = float(np.min(mfccs[i]))
                features[f'covarep_mfcc{i}_max']    = float(np.max(mfccs[i]))
                features[f'covarep_mfcc{i}_median'] = float(np.median(mfccs[i]))

            # ---- FORMANT-LIKE FEATURES ----
            spec_rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)[0]
            features['formant_rolloff_mean']   = float(np.mean(spec_rolloff))
            features['formant_rolloff_std']    = float(np.std(spec_rolloff))
            features['formant_rolloff_min']    = float(np.min(spec_rolloff))
            features['formant_rolloff_max']    = float(np.max(spec_rolloff))
            features['formant_rolloff_median'] = float(np.median(spec_rolloff))

            spec_bandwidth = librosa.feature.spectral_bandwidth(y=y, sr=sr)[0]
            features['formant_bandwidth_mean']   = float(np.mean(spec_bandwidth))
            features['formant_bandwidth_std']    = float(np.std(spec_bandwidth))
            features['formant_bandwidth_min']    = float(np.min(spec_bandwidth))
            features['formant_bandwidth_max']    = float(np.max(spec_bandwidth))
            features['formant_bandwidth_median'] = float(np.median(spec_bandwidth))

            # ---- TEMPO ----
            tempo, _ = librosa.beat.beat_track(y=y, sr=sr)
            features['covarep_tempo_mean'] = float(tempo)

            # ---- HARMONICS ----
            harmonic, percussive = librosa.effects.hpss(y)
            features['covarep_harmonic_mean'] = float(np.mean(np.abs(harmonic)))
```

```
            features['covarep_harmonic_std']  = float(np.std(np.abs(harmonic)))
            features['covarep_percussive_mean'] = float(np.mean(np.abs(percussive)))

            return features

        except Exception as e:
            print(f"Error extracting features: {e}")
            return None

    def analyze_audio_bytes(self, audio_bytes):
        """
        Analyze raw audio bytes and detect depression markers.
        Used by the FastAPI endpoint.
        """
        if not self.model_loaded:
            return {
                'error': 'Model not loaded',
                'depression_detected': None,
                'available': False
            }

        temp_path = None
        try:
            # Save audio bytes to temp file
            with tempfile.NamedTemporaryFile(
                suffix='.wav', delete=False
            ) as tmp:
                tmp.write(audio_bytes)
                temp_path = tmp.name

            # Extract features
            features = self.extract_features_from_audio(temp_path)

            if features is None:
                return {
                    'error': 'Could not extract features - audio too short',
                    'depression_detected': None,
                    'available': True
                }

            # Predict
            result = self.model.predict(features)
            result['available'] = True
            result['error'] = None
            return result

        except Exception as e:
            print(f"Analysis error: {e}")
            return {
                'error': str(e),
                'depression_detected': None,
                'available': True
            }
        finally:
            # Clean up temp file
            if temp_path and os.path.exists(temp_path):
                os.remove(temp_path)

    def get_therapeutic_context(self, result):
        """
        Get therapeutic context based on voice analysis result.
        Used to inform the chatbot response.
        """
        if not result or result.get('error') or result.get('depression_detected') is None:
            return ""

        probability = result.get('probability', 0)
        risk_level = result.get('risk_level', 'Low')

        if risk_level == 'Low':
            return f"""
VOICE ANALYSIS CONTEXT (for therapist awareness only):
- Voice analysis suggests low depression markers (probability: {probability:.1%})
- User appears to be communicating clearly
- No special adjustments needed based on voice
"""
        elif risk_level == 'Moderate':
            return f"""
VOICE ANALYSIS CONTEXT (for therapist awareness only):
- Voice analysis suggests moderate depression markers (probability: {probability:.1%})
- Pay extra attention to emotional undertones
- Gently explore how user has been feeling lately
- Be more supportive and validating in responses
"""
        else:  # High
            return f"""
VOICE ANALYSIS CONTEXT (HIGH CONCERN - for therapist awareness only):
- Voice analysis suggests significant depression markers (probability: {probability:.1%})
- User's voice patterns indicate potential distress
- Be especially warm, supportive, and present
- Gently assess current emotional state
- Consider recommending professional support
- Watch for any crisis indicators
"""

# Global instance
voice_detector = VoiceDepressionDetector()
```