```
import streamlit as st
import plotly.express as px
import plotly.graph_objects as go
from datetime import datetime
import pandas as pd
import openai  # Ensure openai is imported
from utils.ai_helper import get_emotional_support, generate_schedule, get_daily_affirmation,
get_task_details, get_daily_quote
from utils.data_manager import DataManager
from utils.scheduler import ScheduleManager

# Initialize session state
if 'chat_history' not in st.session_state:
    st.session_state.chat_history = []
if 'current_tasks' not in st.session_state:
    st.session_state.current_tasks = []
if 'task_completion' not in st.session_state:
    st.session_state.task_completion = {}
if 'task_details' not in st.session_state:
    st.session_state.task_details = {}
if 'task_times' not in st.session_state:
    st.session_state.task_times = {}
if 'friend_type' not in st.session_state:
    st.session_state.friend_type = None
if 'selected_tab' not in st.session_state:
    st.session_state.selected_tab = None
if 'daily_quote' not in st.session_state:
    st.session_state.daily_quote = get_daily_quote()
if 'conversation_context' not in st.session_state:
    st.session_state.conversation_context = []
# Update the conversation state initialization
if 'conversation_history' not in st.session_state:
    st.session_state.conversation_history = []

# Initialize managers
data_manager = DataManager()
schedule_manager = ScheduleManager()

# Load custom CSS
with open('assets/style.css') as f:
    st.markdown(f'<style>{f.read()}</style>', unsafe_allow_html=True)

def main():
    # Create a container for the sticky navigation
```

```
with st.sidebar:
    st.markdown('<div class="sticky-nav">', unsafe_allow_html=True)

    if st.button("Mood Tracker", use_container_width=True):
        st.session_state.selected_tab = "mood"
        st.rerun()

    if st.button("Talk to a Friend", use_container_width=True):
        st.session_state.selected_tab = "chat"
        st.rerun()

    if st.button("Daily Planner", use_container_width=True):
        st.session_state.selected_tab = "planner"
        st.rerun()

    if st.button("Breathing Exercise", use_container_width=True):
        st.session_state.selected_tab = "breathing"
        st.rerun()

    st.markdown('</div>', unsafe_allow_html=True)

# Create a container for the main content
st.markdown('<div class="main-content">', unsafe_allow_html=True)

# Display daily quote
if st.session_state.selected_tab is None:
    st.markdown("---")
    st.markdown("### Quote of the Day")
    st.markdown(f"*{st.session_state.daily_quote}*")
    st.markdown("---")

    # Display logo
    st.image("attached_assets/Elevate U (3).png", use_container_width=True)

    # Daily Affirmation on home page
    with st.container():
        st.subheader("Today's Affirmation")
        if 'daily_affirmation' not in st.session_state:
            st.session_state.daily_affirmation = get_daily_affirmation()
        st.info(st.session_state.daily_affirmation)
else:
    # Show back button
    if st.button("← Back to Home"):
        st.session_state.selected_tab = None
```

```
            st.rerun()

        # Display selected section
        if st.session_state.selected_tab == "mood":
            display_mood_tracker()
        elif st.session_state.selected_tab == "chat":
            display_support_chat()
        elif st.session_state.selected_tab == "planner":
            display_daily_planner()
        elif st.session_state.selected_tab == "breathing":
            display_breathing_exercise()

    st.markdown('</div>', unsafe_allow_html=True)

def display_breathing_exercise():
    st.subheader("Guided Breathing Exercise")
    st.markdown("Take a moment to breathe and center yourself.")

    # Add breathing circle animation
    breathing_html = """
        <div class="breathing-container">
            <div class="breathing-circle">
                <div class="breathing-text">Breathe</div>
            </div>
        </div>
    """
    st.markdown(breathing_html, unsafe_allow_html=True)

    # Add some guidance text
    st.markdown("""
        ### How to Practice:
        1. Inhale as the circle expands
        2. Hold briefly at full expansion
        3. Exhale as the circle contracts
        4. Repeat for 5-10 cycles

        Remember: Breathe at your own comfortable pace. The animation is just a guide.
    """)

def display_mood_tracker():
    st.subheader("Track Your Mood")

    col1, col2 = st.columns(2)
```

```
with col1:
    mood_score = st.slider("How are you feeling today?", 1, 10, 5,
                    help="1 = Very Low, 5 = Neutral, 10 = Excellent")
    positive_thoughts = st.number_input("Number of positive thoughts today", 0, 100, 0)
    notes = st.text_area("Journal your thoughts")

    if st.button("Save Mood Entry"):
        data_manager.save_mood_entry(mood_score, positive_thoughts, notes)
        st.success("Mood entry saved!")

with col2:
    streak = data_manager.calculate_streak()
    st.metric("Current Streak", f"{streak} days")

    # Display mood history chart
    mood_history = data_manager.get_mood_history()
    if not mood_history.empty:
        # Convert timestamp to datetime if it's not already
        mood_history['timestamp'] = pd.to_datetime(mood_history['timestamp'])

        # Calculate rolling average
        mood_history['rolling_avg'] = mood_history['mood_score'].rolling(window=3).mean()

        # Create the main line chart with gradient colors
        fig = go.Figure()

        # Add the mood score line
        fig.add_trace(go.Scatter(
            x=mood_history['timestamp'],
            y=mood_history['mood_score'],
            name='Mood Score',
            line=dict(color='#90CAF9', width=3),
            mode='lines+markers',
            marker=dict(
                size=8,
                color=mood_history['mood_score'],
                colorscale=[
                    [0, '#ff6b6b'],    # Red for low scores
                    [0.5, '#ffd93d'],  # Yellow for middle scores
                    [1, '#6bcb77']     # Green for high scores
                ],
                colorbar=dict(title="Mood Level"),
                showscale=True
            ),
            hovertemplate="<b>Date:</b> %{x|%Y-%m-%d %H:%M}<br>" +
                         "<b>Mood Score:</b> %{y}<br>" +
                         "<extra></extra>"
        ))

        # Add the rolling average line
        fig.add_trace(go.Scatter(
            x=mood_history['timestamp'],
            y=mood_history['rolling_avg'],
            name='3-Day Average',
            line=dict(color='rgba(144, 202, 249, 0.5)', dash='dash'),
            hovertemplate="<b>3-Day Average:</b> %{y:.1f}<br>" +
                         "<extra></extra>"
        ))

        # Update layout with better styling
        fig.update_layout(
            title='Your Mood History',
            xaxis_title="Date",
            yaxis_title="Mood Score",
            yaxis=dict(
                ticktext=['Very Low', 'Low', 'Neutral', 'Good', 'Excellent'],
                tickvals=[2, 4, 5, 7, 9],
                range=[1, 10]
            ),
            hovermode='x unified',
            showlegend=True,
            legend=dict(
                yanchor="top",
                y=0.99,
                xanchor="left",
                x=0.01
            ),
            plot_bgcolor='rgba(255,255,255,0.9)',
            paper_bgcolor='rgba(255,255,255,0)'
        )

        # Format x-axis to show time in 12-hour format
        fig.update_xaxes(
            tickformat="%I:%M %p\n%b %d",  # Shows time as HH:MM AM/PM and date as
Month Day
            showgrid=True,
            gridwidth=1,
            gridcolor='rgba(128, 128, 128, 0.2)'
        )
        fig.update_yaxes(showgrid=True, gridwidth=1, gridcolor='rgba(128, 128, 128, 0.2)')

        st.plotly_chart(fig, use_container_width=True)

        # Add a helpful description
        st.info("""
        📊 **Understanding Your Mood Chart:**
        - Each point shows your mood score for that day
        - Colors indicate mood levels (red=low, yellow=neutral, green=high)
        - Dotted line shows your 3-day average trend
        - Hover over points to see detailed information
        """)

def display_support_chat():
    st.subheader("Chat with Your Friend")

    # Initialize chat history if not exists
    if 'conversation_history' not in st.session_state:
        st.session_state.conversation_history = []

    # Friend type selector
    if st.session_state.friend_type is None:
        st.write("Choose your friend type! 💫")
        col1, col2, col3 = st.columns(3)

        with col1:
        if st.button("Normal Teenager 🎮"):
            st.session_state.friend_type = "teen"
            st.rerun()

        with col2:
        if st.button("Girl Best Friend 💅"):
            st.session_state.friend_type = "bestie"
            st.rerun()

        with col3:
        if st.button("Cool Bro 🤙"):
            st.session_state.friend_type = "bro"
            st.rerun()
    else:
        # Container for chat history
        chat_container = st.container()
        # Create a container for the input at the bottom
        input_container = st.container()

        # Use the input container for the chat input
        with input_container:
            user_input = st.chat_input("Share your thoughts with me...")

            if user_input:
                try:
                    # Get AI response
                    response = get_emotional_support(
                        user_input,
                        st.session_state.friend_type,
                        st.session_state.conversation_history
                    )

                    # Add messages to conversation history
                    st.session_state.conversation_history.append({
                        "role": "user",
                        "content": user_input
                    })
                    st.session_state.conversation_history.append({
                        "role": "assistant",
                        "content": response
                    })

                except Exception as e:
                    st.error("Sorry, I'm having trouble connecting. Please try again.")
                    return

        # Display chat history in the chat container
        with chat_container:
            for message in st.session_state.conversation_history:
                with st.chat_message(message["role"]):
                    st.write(message["content"])

        # Add option to change friend type
        if st.button("Change Friend Type"):
            st.session_state.friend_type = None
            st.session_state.conversation_history = []
            st.rerun()

    def add_task():
        task = st.session_state.new_task
    if task and task not in st.session_state.current_tasks:
        st.session_state.current_tasks.append(task)
        st.session_state.task_completion[task] = False
        # Get AI-generated details for the task
        details = get_task_details(task)
        st.session_state.task_details[task] = details
        # Initialize default time (9 AM + number of existing tasks)
        default_hour = 9 + len(st.session_state.current_tasks) - 1
        if default_hour < 17:  # Cap at 5 PM
            st.session_state.task_times[task] = f"{default_hour:02d}:00"
    # Clear the input
    st.session_state.new_task = ""

def display_daily_planner():
    st.subheader("Daily Planner")

    # Task input using a form
    st.text_input("Add a task to your day",
                  key="new_task",
                  on_change=add_task,
                  value=st.session_state.get("new_task", ""))

    # Display current tasks with checkboxes and details
    if st.session_state.current_tasks:
        st.write("Your Tasks:")
        for i, task in enumerate(st.session_state.current_tasks):
            col1, col2, col3 = st.columns([1, 2, 4])

            with col1:
                # Update task completion status
                completed = st.checkbox("Done", key=f"task_{i}",
                                        value=st.session_state.task_completion.get(task, False))
                st.session_state.task_completion[task] = completed

            with col2:
                # Time selector for each task
                times = [f"{h:02d}:00" for h in range(9, 18)]  # 9 AM to 5 PM
                selected_time = st.selectbox(
                    "Time",
                    times,
                    key=f"time_{i}",
                    index=times.index(st.session_state.task_times.get(task, "09:00"))
                )
                st.session_state.task_times[task] = selected_time
            with col3:
                st.write(f"{task}")
                if task in st.session_state.task_details:
                    with st.expander("View affirmation and tip"):
                        details = st.session_state.task_details[task]
                        st.info(f"🌟 Affirmation: {details['affirmation']}")
                        st.success(f"💡 Tip: {details['tip']}")

        if st.button("Update Schedule"):
            # Generate new schedule based on uncompleted tasks and their selected times
            schedule = {}
            active_tasks = [task for task in st.session_state.current_tasks
                              if not st.session_state.task_completion.get(task, False)]

            for task in active_tasks:
                time_slot = st.session_state.task_times.get(task, "09:00")
                schedule[time_slot] = task

            st.session_state.current_schedule = schedule
            data_manager.save_daily_tasks(active_tasks, schedule)

    # Display current schedule if it exists
    if hasattr(st.session_state, 'current_schedule') and st.session_state.current_schedule:
        display_schedule(st.session_state.current_schedule)

def display_schedule(schedule):
    st.subheader("Your Daily Schedule")
    formatted_schedule = schedule_manager.format_schedule(schedule)

    # Create a timeline visualization
    fig = go.Figure()

    for item in formatted_schedule:
        fig.add_trace(go.Scatter(
            x=[item["time"], item["time"]],
            y=[0, 1],
            mode="lines",
            name=item["activity"],
            text=item["activity"],
            hoverinfo="text"
        ))

    fig.update_layout(
        title="Daily Timeline",
        xaxis_title="Time",
        showlegend=False,
        height=200
    )

    st.plotly_chart(fig)

    # Display schedule as a list
    for item in formatted_schedule:
        st.write(f"{item['time']}: {item['activity']}")

def analyze_mood_score(user_input):
    # Placeholder:  Replace with actual mood analysis logic
    # This is a dummy function.  You'll need to implement real mood analysis here.
    # For example, you could use a sentiment analysis library or a more sophisticated NLP
technique.
    positive_words = ["good", "great", "happy", "excited", "wonderful"]
    negative_words = ["bad", "sad", "angry", "depressed", "terrible"]

    positive_count = sum(1 for word in positive_words if word in user_input.lower())
    negative_count = sum(1 for word in negative_words if word in user_input.lower())

    score = max(1, min(10, 5 + positive_count - negative_count)) #Keep score between 1 and 10
    return score

if __name__ == "__main__":
    main()
from openai import OpenAI
import os
import json
import time

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

def get_emotional_support(user_input, friend_type=None, conversation_context=None):
    """Direct emotional support conversation."""
    try:
        # Simple but effective personality prompts
        personality = {
            "teen": "You're a teenage friend who understands young people. Use casual language,
emojis, and share relatable experiences. Address the user's specific situation.",
            "bestie": "You're a caring best friend. Be warm, supportive, and understanding.
Reference specific details from what they share.",
            "bro": "You're a chill friend. Keep it real while being supportive. Use relaxed language
and show you're really listening."
        }

        # Build messages with minimal context
        messages = [
            {
                "role": "system",
                "content": f"{personality.get(friend_type, personality['bestie'])} Focus on responding
directly to what they say."
            }
        ]

        # Add last message for minimal context
        if conversation_context and len(conversation_context) > 0:
            last_msg = conversation_context[-1]
            if last_msg["role"] == "user":
                messages.append(last_msg)

        # Add current message with instruction for focused response
        messages.append({
            "role": "user",
            "content": f"Respond to this, referencing specific details they mention: {user_input}"
        })

        # Get response with optimal parameters
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=messages,
            temperature=0.7,
            max_tokens=60
        )

        return response.choices[0].message.content.strip()

    except Exception as e:
        # Smart fallback responses based on context
        input_lower = user_input.lower()
        if any(word in input_lower for word in ["sad", "hurt", "depressed", "angry"]):
            return f"I can see that's really affecting you. What specifically about
{input_lower.split()[1:4]} is troubling you the most?"
        elif any(word in input_lower for word in ["happy", "great", "excited"]):
            return f"That's fantastic! Tell me more about what's making you feel so good!"
        elif "?" in input_lower:
```

```
        return "That's a great question. What made you think about that?"
    else:
        return "I'm really interested in hearing more about that. Could you tell me what
happened?"

def get_daily_quote():
    """Get an inspiring quote."""
    quotes = [
        "Your strength grows with every challenge you face.",
        "Small steps today lead to big changes tomorrow.",
        "You have the power to create positive change.",
        "Every moment is a chance to start fresh.",
        "Your journey is uniquely yours - embrace it."
    ]
    return quotes[int(time.time()) % len(quotes)]

def get_daily_affirmation():
    """Get a daily affirmation."""
    affirmations = [
        "I grow stronger with each passing day.",
        "I choose to embrace my confidence.",
        "I create my own path to happiness.",
        "I am worthy of wonderful things.",
        "I can overcome any obstacle."
    ]
    return affirmations[int(time.time()) % len(affirmations)]

def get_task_details(task):
    """Get task details with better focus."""
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{
                "role": "user",
                "content": f"For task '{task}', provide a motivating affirmation and practical tip in JSON
format"
            }],
            max_tokens=40,
            response_format={"type": "json_object"}
        )
        return json.loads(response.choices[0].message.content)
    except Exception:
        return {
            "affirmation": f"I will accomplish {task} with focus and determination!",
            "tip": "Break this down into smaller, manageable steps."
        }

def generate_schedule(tasks):
    """Generate simple schedule."""
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{
                "role": "user",
                "content": f"Create a simple, practical schedule for: {', '.join(tasks)}"
            }],
            max_tokens=60
        )
        return response.choices[0].message.content.strip()
    except Exception:
        return "Distribute these tasks evenly throughout your day for best results."
import pandas as pd
import json
from datetime import datetime
import os

class DataManager:
    def __init__(self):
        self.mood_file = "data/mood_tracker.csv"
        self.tasks_file = "data/tasks.csv"
        self._initialize_data_files()

    def _initialize_data_files(self):
        """Initialize data files if they don't exist."""
        os.makedirs("data", exist_ok=True)

        if not os.path.exists(self.mood_file):
            pd.DataFrame(columns=[
                'timestamp', 'mood_score', 'positive_thoughts', 'notes'
            ]).to_csv(self.mood_file, index=False)

        if not os.path.exists(self.tasks_file):
            pd.DataFrame(columns=[
                'date', 'tasks', 'schedule'
            ]).to_csv(self.tasks_file, index=False)

    def save_mood_entry(self, mood_score, positive_thoughts, notes=""):
        """Save a new mood entry."""
        new_entry = pd.DataFrame([{
            'timestamp': datetime.now(),
            'mood_score': mood_score,
            'positive_thoughts': positive_thoughts,
            'notes': notes
        }])

        if os.path.exists(self.mood_file):
            df = pd.read_csv(self.mood_file)
            df = pd.concat([df, new_entry], ignore_index=True)
        else:
            df = new_entry

        df.to_csv(self.mood_file, index=False)

    def get_mood_history(self, days=30):
        """Get mood history for the specified number of days."""
        if os.path.exists(self.mood_file):
            df = pd.read_csv(self.mood_file)
            df['timestamp'] = pd.to_datetime(df['timestamp'])
            recent_data = df.sort_values('timestamp').tail(days)
            return recent_data
        return pd.DataFrame()

    def save_daily_tasks(self, tasks, schedule):
        """Save daily tasks and generated schedule."""
        new_entry = pd.DataFrame([{
            'date': datetime.now().date(),
            'tasks': json.dumps(tasks),
            'schedule': json.dumps(schedule)
        }])

        if os.path.exists(self.tasks_file):
            df = pd.read_csv(self.tasks_file)
            df = pd.concat([df, new_entry], ignore_index=True)
        else:
            df = new_entry

        df.to_csv(self.tasks_file, index=False)

    def get_current_day_tasks(self):
        """Get tasks and schedule for current day."""
        if os.path.exists(self.tasks_file):
            df = pd.read_csv(self.tasks_file)
            today = datetime.now().date().isoformat()
            today_data = df[df['date'] == today]

            if not today_data.empty:
                tasks = json.loads(today_data.iloc[0]['tasks'])
                schedule = json.loads(today_data.iloc[0]['schedule'])
                return tasks, schedule
        return [], {}

    def calculate_streak(self):
        """Calculate current streak of positive thoughts."""
        if os.path.exists(self.mood_file):
            df = pd.read_csv(self.mood_file)
            df['timestamp'] = pd.to_datetime(df['timestamp'])
            df = df.sort_values('timestamp')

            if df.empty:
                return 0

            streak = 0
            current_date = datetime.now().date()

            for _, row in df.iloc[::-1].iterrows():
                entry_date = row['timestamp'].date()
                if (current_date - entry_date).days > 1:
                    break
                if row['positive_thoughts'] > 0:
                    streak += 1
                current_date = entry_date

            return streak
        return 0

from datetime import datetime, timedelta

class ScheduleManager:
    def __init__(self):
        self.time_slots = self._generate_time_slots()

    def _generate_time_slots(self):
        """Generate available time slots for scheduling."""
        slots = []
        start_time = datetime.now().replace(hour=8, minute=0, second=0, microsecond=0)
        end_time = start_time.replace(hour=22, minute=0)
        current_time = start_time
        while current_time < end_time:
            slots.append(current_time.strftime("%H:%M"))
            current_time += timedelta(hours=1)  # Changed from minutes=30 to hours=1

        return slots

    def _convert_to_12hr_format(self, time_24hr):
        """Convert 24-hour time format to 12-hour format."""
        time_obj = datetime.strptime(time_24hr, "%H:%M")
        return time_obj.strftime("%I:%M %p").lstrip("0")

    def format_schedule(self, schedule_data):
        """Format the AI-generated schedule for display."""
        formatted_schedule = []
        for time_slot in self.time_slots:
            activity = schedule_data.get(time_slot, "Free time")
            formatted_schedule.append({
                "time": self._convert_to_12hr_format(time_slot),
                "activity": activity
            })
        return formatted_schedule

    def validate_schedule(self, schedule_data):
        """Validate the schedule format and time slots."""
        valid_schedule = {}
        for time_slot in self.time_slots:
            if time_slot in schedule_data:
                valid_schedule[time_slot] = schedule_data[time_slot]
            else:
                valid_schedule[time_slot] = "Free time"
        return valid_schedule

    def get_current_activity(self):
        """Get the current activity based on time."""
        current_time = datetime.now()
        current_slot = current_time.strftime("%H:%M")

        # Find the closest time slot
        for i, slot in enumerate(self.time_slots):
            if slot > current_slot:
                if i > 0:
                    return self._convert_to_12hr_format(self.time_slots[i-1])
        return self._convert_to_12hr_format(self.time_slots[0])
return self._convert_to_12hr_format(self.time_slots[-1])
```