🧠 Agent #1: Individualized Financial Advisory Agent

📝 Overview

Build an AI agent that provides personalized financial advice to users based on their income, spending habits, financial goals, and risk profile. The agent will interact via a chat interface, summarize user financial data, and suggest savings, investment, or budget strategies.

🧪 Lab Objectives

By the end of this lab, you will:

- Build a chat-based financial advisor using OpenAI's GPT + LangChain
- Connect user input to mock financial data
- Generate personalized recommendations
- Output summaries using structured templates

🧰 Tech Stack

- **Python**
- **OpenAI GPT-4 / GPT-3.5**
- **LangChain**
- **Streamlit** (for UI)
- **Pandas** (mock financial data)

🧭 Step-by-Step Instructions

✅ Step 1: Set up the environment

```
mkdir financial_advisor_agent
cd financial_advisor_agent
python -m venv venv
source venv/bin/activate   # or venv\Scripts\activate on Windows

pip install openai langchain streamlit pandas
```

✅ Step 2: Create your mock financial dataset (`mock_data.py`)

```
import pandas as pd

def get_user_financial_data(user_id="user123"):
    return {
        "income": 70000,
        "monthly_expenses": {
            "housing": 1500,
            "food": 600,
            "transport": 300,
            "entertainment": 200,
            "others": 150,
        },
        "financial_goals": ["save for a house", "retire at 60"],
        "risk_tolerance": "moderate"
    }
```

✅ Step 3: Build your prompt template (`advisor_prompt.py`)

```
from langchain.prompts import PromptTemplate

advisor_template = PromptTemplate.from_template("""
You are a financial advisor AI. Given the user's financial profile
        below, provide 3 personalized suggestions:
- Summarize their financial status
- Suggest a monthly savings goal
- Recommend an investment strategy based on their risk profile

User Profile:
Income: ${income}
Expenses: ${expenses}
Goals: ${goals}
Risk: ${risk}

Respond clearly and concisely.
""")
```

✅ Step 4: Create your LangChain agent logic (`advisor_agent.py`)

```
import openai
from langchain.chat_models import ChatOpenAI
from advisor_prompt import advisor_template
from mock_data import get_user_financial_data

def run_financial_advisor():
    user_data = get_user_financial_data()
    input_str = advisor_template.format(
```

```
        income=user_data["income"],
        expenses=user_data["monthly_expenses"],
        goals=", ".join(user_data["financial_goals"]),
        risk=user_data["risk_tolerance"]
    )

    llm = ChatOpenAI(temperature=0.4)
    response = llm.predict(input_str)
    return response
```

✅ Step 5: Build the UI with Streamlit (`app.py`)

```
import streamlit as st
from advisor_agent import run_financial_advisor

st.title("Individualized Financial Advisory Agent 💰")
if st.button("Get Advice"):
    result = run_financial_advisor()
    st.markdown("### Your Financial Plan")
    st.write(result)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Your income is $70,000 per year with moderate expenses. You could save
$1,200/month.
To meet your goal of buying a house, start a high-yield savings
account.
For retirement, consider a 60% stock / 40% bond portfolio with
automated contributions.
```

📈 Agent #10: AI Portfolio Manager

📝 Overview

This AI agent assists in managing investment portfolios by evaluating asset performance, balancing risk, and recommending reallocation strategies. In this lab, you'll simulate an investment portfolio and use GPT to assess allocation health, risk diversification, and next-step actions.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a multi-asset investment portfolio
- Analyze allocation vs. expected returns and risk
- Use GPT to provide a portfolio health summary and rebalancing strategy
- Visualize portfolio distribution and performance

🧰 Tech Stack

- **Python**
- **Pandas + Matplotlib**
- **LangChain + GPT-4 / GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

If continuing from earlier labs, skip this. Otherwise:

```
mkdir portfolio_manager_agent
cd portfolio_manager_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas matplotlib streamlit
```

✅ Step 2: Simulate portfolio data (`portfolio_data.py`)

```
import pandas as pd

def load_portfolio():
    data = {
        "Asset": ["US Stocks", "International Stocks", "Bonds", "Real
        Estate", "Crypto"],
        "Allocation (%)": [40, 25, 20, 10, 5],
        "Annual Return (%)": [8.0, 6.5, 3.5, 5.0, 20.0],
        "Risk Score (1–10)": [6, 7, 2, 5, 9]
    }
    return pd.DataFrame(data)
```

✅ Step 3: Create GPT prompt template (`portfolio_prompt.py`)

```
from langchain.prompts import PromptTemplate

portfolio_template = PromptTemplate.from_template("""
You are an AI investment advisor.

Given the portfolio below:

{portfolio_table}

Please:
1. Evaluate the portfolio's risk vs. return balance.
2. Suggest any rebalancing if overexposed to high-risk or low-return
        assets.
3. Recommend a strategy for the next quarter (e.g., shift, diversify,
        hedge).

Respond in a concise, professional tone.
""")
```

✅ Step 4: GPT analysis logic (`portfolio_agent.py`)

```
from portfolio_data import load_portfolio
from portfolio_prompt import portfolio_template
from langchain.chat_models import ChatOpenAI

def analyze_portfolio():
    df = load_portfolio()
    table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = portfolio_template.format(portfolio_table=table)
    response = llm.predict(prompt)

    return df, response
```

✅ Step 5: Streamlit app (`app.py`)

```
import streamlit as st
import matplotlib.pyplot as plt
from portfolio_agent import analyze_portfolio

st.title("📈 AI Portfolio Manager")

if st.button("Analyze My Portfolio"):
    df, summary = analyze_portfolio()
```

```
    st.subheader("💼 Portfolio Breakdown")
    st.dataframe(df)

    st.subheader("📊 Allocation Pie Chart")
    fig, ax = plt.subplots()
    ax.pie(df["Allocation (%)"], labels=df["Asset"],
        autopct="%1.1f%%", startangle=90)
    ax.axis("equal")
    st.pyplot(fig)

    st.subheader("🧠 AI Investment Guidance")
    st.write(summary)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
The current portfolio is moderately aggressive with a heavy tilt
toward equities and crypto. While US stocks provide stable returns,
the 20% exposure to high-volatility crypto increases downside risk.

Recommendations:
- Consider reducing crypto to 2% and shifting 3% into bonds to improve
diversification.
- Real estate remains underweight given inflation hedging benefits —
consider increasing it to 15%.
- Maintain equities but review international market exposure
quarterly.

Outlook:
Adopt a “balanced growth” strategy with quarterly rebalancing and tax-
loss harvesting.
```

📝 Agent #100: Executive Summary Generator Agent

🧠 Overview

The Executive Summary Generator Agent takes long-form content—such as reports, meeting transcripts, research papers, or business plans—and produces clear, concise executive summaries. These summaries highlight key decisions, insights, and next steps. In this lab, you'll build an agent that ingests a document and generates a bullet or paragraph summary designed for busy decision-makers.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload or paste long-form text
- Use GPT to summarize the content in executive style
- Choose between paragraph and bullet formats
- Export the result as downloadable text

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add text chunking and document type classification)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir executive_summary_agent
cd executive_summary_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Summarization Logic (`summarizer.py`)

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0.3)

def summarize_text(text, mode="bullet"):
    prompt_type = {
        "bullet": "Generate an executive summary in bullet points (max
        7 items)",
        "paragraph": "Write a 150-word executive summary paragraph"
    }
    prompt = f"""
    {prompt_type[mode]}
    Content:
    {text[:5000]}  # Trim for token efficiency
    """
    return llm.predict(prompt)
```

✅ Step 3: Streamlit Interface (`app.py`)

```
import streamlit as st
from summarizer import summarize_text

st.title("📝 Executive Summary Generator Agent")
st.caption("Upload or paste content and get instant executive
        summaries")

input_text = st.text_area("Paste your document content:", height=300)
summary_mode = st.radio("Choose summary style:", ["bullet",
        "paragraph"])

if st.button("Generate Summary") and input_text:
    with st.spinner("Generating summary..."):
        result = summarize_text(input_text, summary_mode)
        st.subheader("📄 Executive Summary")
        st.text_area("Output", result, height=250)
        st.download_button("📩 Download Summary", data=result,
        file_name="executive_summary.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Input: A 7-page quarterly operations report.

Output (Bullet Format):

- Revenue increased by 12% YoY, driven by APAC expansion
- Customer churn reduced from 9% to 6% via onboarding improvements
- Cloud costs were optimized, saving $1.3M in Q2
- Product roadmap delay expected due to hiring freeze
- Net Promoter Score improved from 48 to 56
- Key risk: potential regulatory delays in EU launch
- Next review scheduled for October 1, 2025

Output (Paragraph Format): “In Q2, the company saw a 12% revenue increase, improved customer retention, and significant cloud cost savings. While the NPS score rose, hiring delays could impact product delivery. The EU launch faces regulatory hurdles, and the leadership team plans to re-evaluate timelines in the next quarterly review.”

👔 Agent #11: Employee Hiring Advisor

📝 Overview

This AI agent assists HR teams in screening candidates by analyzing resumes, matching them with job descriptions, and ranking applicants based on fit. In this lab, you'll simulate a candidate pool, job requirements, and use GPT to evaluate candidate-job fit and generate interview shortlists.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate candidate profiles and a sample job description
- Use GPT to analyze and rank candidate fit
- Generate shortlisting justifications and interview recommendations
- Display everything in a clean Streamlit interface

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If you haven't already:

```
mkdir hiring_advisor_agent
cd hiring_advisor_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate job and candidate data (`hiring_data.py`)

```
import pandas as pd

def get_job_description():
    return """We are hiring a Data Analyst with:
```

```
- 2+ years experience in Python and SQL
- Familiarity with BI tools like Tableau or Power BI
- Understanding of statistics and A/B testing
- Strong communication and collaboration skills
"""

def get_candidates():
    return pd.DataFrame({
        "Name": ["Alice", "Bob", "Clara", "David", "Elena"],
        "Experience (yrs)": [3, 1, 5, 2, 4],
        "Skills": [
            "Python, SQL, Tableau, Excel",
            "Excel, SQL, Google Sheets",
            "Python, R, Power BI, Statistics, A/B Testing",
            "Python, SQL, Tableau, Statistics",
            "Power BI, SQL, Excel, Communication"
        ]
    })
```

✅ Step 3: Create GPT prompt template (`hiring_prompt.py`)

```
from langchain.prompts import PromptTemplate

hiring_template = PromptTemplate.from_template("""
You are an AI hiring advisor. Given this job description:

{job_description}

And the following candidates:

{candidate_table}

Please:
1. Rank the candidates from best to least fit (1 to 5).
2. Justify your rankings based on skills and experience match.
3. Recommend top 2 candidates for interview with reasons.
""")
```

✅ Step 4: Generate evaluation using GPT (`hiring_agent.py`)

```
from hiring_data import get_job_description, get_candidates
from hiring_prompt import hiring_template
from langchain.chat_models import ChatOpenAI

def evaluate_candidates():
    jd = get_job_description()
    df = get_candidates()
    table = df.to_string(index=False)
```

```
    llm = ChatOpenAI(temperature=0.2)
    prompt = hiring_template.format(job_description=jd,
        candidate_table=table)
    result = llm.predict(prompt)

    return jd, df, result
```

✅ Step 5: Build the Streamlit interface (`app.py`)

```
import streamlit as st
from hiring_agent import evaluate_candidates

st.title("👔 Employee Hiring Advisor")

if st.button("Evaluate Candidates"):
    jd, df, summary = evaluate_candidates()

    st.subheader("📄 Job Description")
    st.code(jd)

    st.subheader("👥 Candidate Pool")
    st.dataframe(df)

    st.subheader("🧠 AI Shortlisting & Justification")
    st.write(summary)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Candidate Ranking:
1. Clara – Excellent skills match with Python, Power BI, Statistics,
and A/B Testing.
2. Alice – Strong fit with Python, SQL, and Tableau.
3. David – Good match, lacks business tools like Power BI.
4. Elena – Moderate fit, missing statistical background.
5. Bob – Lacks Python and BI tools.

Recommend interviewing Clara and Alice for strong alignment with job
requirements.
```

👋 Agent #12: Employee Onboarding Agent

📝 Overview

This AI agent automates and personalizes the onboarding process for new hires by delivering checklists, policy docs, welcome messages, and role-specific guidance. In this lab, you'll simulate a new employee onboarding scenario and build a chatbot-like experience that guides them through tasks.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate an onboarding checklist and employee profile
- Use GPT to provide a dynamic, personalized onboarding assistant
- Display onboarding steps in a conversational Streamlit interface
- Track completion and generate a welcome summary

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If continuing from earlier labs, skip this. Otherwise:

```
mkdir onboarding_agent
cd onboarding_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate onboarding checklist and employee profile (`onboarding_data.py`)

```
def get_employee_profile():
    return {
        "name": "Jordan Lee",
```

```
        "role": "Data Analyst",
        "department": "Business Intelligence",
        "start_date": "2025-08-01"
    }

def get_onboarding_tasks():
    return [
        "Complete HR paperwork",
        "Review data security policy",
        "Attend team introduction call",
        "Set up company email & tools",
        "Schedule 1:1 with manager",
        "Complete Tableau training module"
    ]
```

✅ Step 3: GPT prompt for personalized onboarding guide (`onboarding_prompt.py`)

```
from langchain.prompts import PromptTemplate

onboarding_template = PromptTemplate.from_template("""
You are an AI onboarding assistant. A new employee is starting soon:

Name: {name}
Role: {role}
Department: {department}
Start Date: {start_date}

Onboarding Tasks:
{tasks_list}

Create a welcome message with:
1. Personalized greeting
2. Overview of first-week priorities
3. Encouraging tone to build excitement
""")
```

✅ Step 4: Generate onboarding plan (`onboarding_agent.py`)

```
from onboarding_data import get_employee_profile, get_onboarding_tasks
from onboarding_prompt import onboarding_template
from langchain.chat_models import ChatOpenAI

def generate_onboarding_message():
    profile = get_employee_profile()
    tasks = get_onboarding_tasks()
    tasks_str = "\n- " + "\n- ".join(tasks)
```

```
    llm = ChatOpenAI(temperature=0.4)
    prompt = onboarding_template.format(
        name=profile["name"],
        role=profile["role"],
        department=profile["department"],
        start_date=profile["start_date"],
        tasks_list=tasks_str
    )
    message = llm.predict(prompt)
    return profile, tasks, message
```

✅ Step 5: Streamlit interface (`app.py`)

```
import streamlit as st
from onboarding_agent import generate_onboarding_message

st.title("👋 AI Employee Onboarding Agent")

if st.button("Generate Welcome Plan"):
    profile, tasks, message = generate_onboarding_message()

    st.subheader("👨‍💼 New Hire Profile")
    st.write(profile)

    st.subheader("📋 Onboarding Checklist")
    for task in tasks:
        st.checkbox(task)

    st.subheader("💬 Personalized Welcome Message")
    st.write(message)
```

Run it:

```
streamlit run app.py
```

🧪 Example Output:

```
Welcome, Jordan!

We're thrilled to have you join the Business Intelligence team as a
Data Analyst starting August 1st. Your first week will focus on
essential setup, meeting your team, and diving into core tools like
Tableau.

Priorities:
- HR paperwork and policy reviews
- Getting your systems and accounts ready
- Meeting your manager and colleagues
```

```
We're here to support you every step of the way. Let's make your
onboarding experience great!
```

👨‍💼 Agent #13: HR Companion Agent

📝 Overview

This AI agent acts as an always-on virtual assistant for employees, answering HR-related questions about benefits, leave policy, payroll, and more. In this lab, you'll build a chatbot-style assistant that uses GPT to respond to natural language queries based on internal HR policies.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate HR policy documents
- Create a chat interface where employees ask HR questions
- Use GPT to retrieve and answer from the HR knowledge base
- Display responses in Streamlit as an interactive assistant

🧰 Tech Stack

- **Python**
- **LangChain + GPT-3.5 or GPT-4**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

If you haven't already:

```bash
mkdir hr_companion_agent
cd hr_companion_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Simulate HR policy FAQ knowledge base (`hr_policies.py`)

```
def get_hr_knowledge_base():
    return """
1. PTO Policy: Employees receive 15 paid vacation days per year and 10
        paid holidays. PTO must be approved by your manager.

2. Parental Leave: New parents are eligible for 12 weeks of paid
        leave.

3. Health Benefits: Employees are enrolled in Aetna PPO plans. Dental
        and vision are optional add-ons.

4. Remote Work: Employees can work remotely up to 3 days per week.

5. Payroll: Salaries are paid on the 15th and last day of each month.
        Direct deposit is required.

6. Training Budget: Every employee receives $1000/year for approved
        training.
"""
```

✅ Step 3: Create prompt template for GPT (`hr_prompt.py`)

```
from langchain.prompts import PromptTemplate

hr_prompt = PromptTemplate.from_template("""
You are an HR Companion AI. Use the company policies below to answer
        the employee's question:

Company HR Policies:
{policies}

Employee's Question:
{question}

Answer as clearly and politely as possible.
""")
```

✅ Step 4: Define GPT-powered HR agent logic (`hr_agent.py`)

```
from hr_policies import get_hr_knowledge_base
from hr_prompt import hr_prompt
from langchain.chat_models import ChatOpenAI

def get_hr_response(question: str):
    policies = get_hr_knowledge_base()
    llm = ChatOpenAI(temperature=0.3)
    prompt = hr_prompt.format(policies=policies, question=question)
```

```
    response = llm.predict(prompt)
    return response
```

✅ Step 5: Build Streamlit interface (`app.py`)

```
import streamlit as st
from hr_agent import get_hr_response

st.title("👨‍💼 HR Companion Agent")

user_question = st.text_input("Ask an HR question (e.g., 'How many
        vacation days do I get?')")

if user_question:
    response = get_hr_response(user_question)
    st.subheader("💬 AI Response")
    st.write(response)
```

Run it:

```
streamlit run app.py
```

🧪 Example Output:

User input: *"Am I allowed to work from home on Fridays?"*

AI Response: "Yes, employees can work remotely up to 3 days per week. If Friday is within your remote schedule, you're allowed to work from home. Please coordinate with your manager."

🧾 Agent #14: HR Support Agent

📝 Overview

This AI agent supports HR operations by handling repetitive tasks like answering employee questions, generating HR reports, sending reminders, and logging leave requests. In this lab, you'll simulate an HR helpdesk agent that routes queries and automates typical HR tasks using GPT.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate HR task categories (e.g., leave requests, reporting, reminders)

- Use GPT to classify and route incoming queries
- Automate basic HR task responses
- Provide a simple form-driven Streamlit interface

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 / GPT-3.5**
- **Streamlit**
- **Pandas** (for mock reporting)

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If not already done:

```
mkdir hr_support_agent
cd hr_support_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate query types (`hr_tasks.py`)

```
TASKS = {
    "Leave Request": "Record PTO request and notify manager",
    "Policy Inquiry": "Respond to question using the HR policy
        database",
    "Payroll Issue": "Log issue with payroll and forward to HR ops",
    "Report Generation": "Generate headcount or leave balance
        reports",
    "Reminder Request": "Send reminder email for training or
        compliance",
}
```

✅ Step 3: GPT prompt to classify and respond (`hr_support_prompt.py`)

```
from langchain.prompts import PromptTemplate

hr_support_template = PromptTemplate.from_template("""
You are an AI assistant in an HR support team.

Here are known categories of requests:
{task_list}
```

```
Given the employee's message:
"{message}"

1. Classify the request into one of the task categories.
2. Generate an appropriate response.
""")
```

✅ Step 4: Core HR agent logic (`hr_support_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from hr_support_prompt import hr_support_template
from hr_tasks import TASKS

def handle_hr_query(user_message):
    task_list = "\n".join(f"- {key}: {value}" for key, value in
        TASKS.items())

    llm = ChatOpenAI(temperature=0.3)
    prompt = hr_support_template.format(
        task_list=task_list,
        message=user_message
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit interface (`app.py`)

```python
import streamlit as st
from hr_support_agent import handle_hr_query

st.title("🤖 HR Support Agent")

user_msg = st.text_area("Enter your request (e.g., 'I need to take off
        next Friday')")

if st.button("Submit"):
    if user_msg:
        response = handle_hr_query(user_msg)
        st.subheader("📋 AI Response")
        st.write(response)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

User Input: “Can I take PTO from Sept 10 to Sept 12?”

AI Output: Classification: Leave Request **Response:** “Your PTO request from Sept 10 to Sept 12 has been recorded. Please ensure your manager is notified for final approval. Let me know if you’d like a calendar block created.”

🤖 Agent #15: AI Recruitment Coordinator

📝 Overview

This AI agent automates coordination tasks in the recruitment process: scheduling interviews, sending reminders, updating candidate statuses, and responding to common questions. In this lab, you’ll simulate an end-to-end coordination flow and use GPT to generate professional messages and reminders.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate candidate pipeline stages and interview slots
- Use GPT to auto-generate scheduling emails and reminders
- Display an interactive interface to manage candidates
- Prepare communication outputs for HR and hiring managers

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

```
mkdir recruitment_coordinator_agent
cd recruitment_coordinator_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Create mock candidate data (`candidate_data.py`)

```
import pandas as pd

def get_candidates():
    return pd.DataFrame({
        "Candidate": ["Alice", "Bob", "Clara"],
        "Role": ["Data Analyst", "UX Designer", "ML Engineer"],
        "Stage": ["Phone Screen", "Technical Interview", "HR
        Interview"],
        "Preferred Time": ["2025-08-01 10:00", "2025-08-01 14:00",
        "2025-08-02 09:00"]
    })
```

✅ Step 3: Create GPT prompt for scheduling (`recruitment_prompt.py`)

```
from langchain.prompts import PromptTemplate

recruitment_template = PromptTemplate.from_template("""
You are an AI recruitment coordinator.

For the following candidate:
- Name: {name}
- Role: {role}
- Interview Stage: {stage}
- Preferred Time: {time}

Please:
1. Write a professional email to the candidate confirming the
        interview.
2. Suggest 2 alternate times in case of conflict.
3. Include a friendly tone and a reminder to bring any required
        materials.
""")
```

✅ Step 4: Generate GPT output (`recruitment_agent.py`)

```
from candidate_data import get_candidates
from recruitment_prompt import recruitment_template
from langchain.chat_models import ChatOpenAI

def generate_emails():
    llm = ChatOpenAI(temperature=0.4)
    df = get_candidates()
    responses = []

    for _, row in df.iterrows():
        prompt = recruitment_template.format(
```

```
            name=row["Candidate"],
            role=row["Role"],
            stage=row["Stage"],
            time=row["Preferred Time"]
        )
        response = llm.predict(prompt)
        responses.append((row["Candidate"], response))

    return df, responses
```

✅ Step 5: Streamlit interface (`app.py`)

```
import streamlit as st
from recruitment_agent import generate_emails

st.title("📅 AI Recruitment Coordinator")

if st.button("Generate Interview Emails"):
    df, responses = generate_emails()

    st.subheader("📋 Candidate Pipeline")
    st.dataframe(df)

    for name, email_text in responses:
        st.subheader(f"📧 Email for {name}")
        st.write(email_text)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Email to Bob: Hi Bob,

Thank you for your continued interest in the UX Designer position. We've scheduled your technical interview for **August 1st at 2:00 PM** via Zoom.

If this time does not work for you, we can offer:

- August 1st at 3:30 PM
- August 2nd at 11:00 AM

Please bring your design portfolio and be ready to walk through a case study. We're looking forward to connecting!

Best, Recruitment Team

💼 Agent #16: Benefits Analyst Agent

📝 Overview

This AI agent helps employees compare and choose from available benefit plans (health, dental, vision, retirement, etc.) based on eligibility, family situation, and preferences. In this lab, you'll simulate benefit options and employee profiles, and use GPT to recommend the most suitable package.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate employee benefit options and eligibility factors
- Build a form to collect employee inputs (e.g., family size, health priorities)
- Use GPT to analyze and recommend plans
- Provide a benefits summary in a Streamlit interface

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

If you haven't yet:

```
mkdir benefits_analyst_agent
cd benefits_analyst_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Define benefits options (`benefit_data.py`)

```
def get_benefits_catalog():
    return """
1. Health Plan A: Low premium, high deductible. Best for healthy
        individuals.
```

```
2. Health Plan B: Higher premium, low deductible. Covers most routine
       care.
3. Dental Plan A: Covers preventive and basic care.
4. Vision Plan A: Covers eye exams, basic lenses.
5. Retirement 401(k): Company match up to 5%.
6. HSA Account: Available with Health Plan A for tax-free savings.
"""
```

✅ Step 3: GPT Prompt Template (`benefit_prompt.py`)

```python
from langchain.prompts import PromptTemplate

benefit_prompt = PromptTemplate.from_template("""
You are a benefits advisor AI.

Given the benefits catalog:
{catalog}

And the employee details:
- Age: {age}
- Family Status: {family}
- Health Priorities: {priority}
- Risk Tolerance: {risk}

Suggest:
1. The best combination of health, dental, and vision plans.
2. Whether to use an HSA or 401(k).
3. One-paragraph explanation tailored to the profile.
""")
```

✅ Step 4: Build GPT Logic (`benefit_agent.py`)

```python
from benefit_data import get_benefits_catalog
from benefit_prompt import benefit_prompt
from langchain.chat_models import ChatOpenAI

def recommend_benefits(age, family, priority, risk):
    catalog = get_benefits_catalog()
    llm = ChatOpenAI(temperature=0.4)

    prompt = benefit_prompt.format(
        catalog=catalog,
        age=age,
        family=family,
        priority=priority,
        risk=risk
    )
```

```
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from benefit_agent import recommend_benefits

st.title("💼 Benefits Analyst Agent")

with st.form("benefits_form"):
    age = st.number_input("Your Age", min_value=18, max_value=70,
        value=30)
    family = st.selectbox("Family Status", ["Single", "Married",
        "Married with Kids"])
    priority = st.selectbox("Primary Health Concern", ["Routine Care",
        "Major Illness Coverage", "Low Cost"])
    risk = st.selectbox("Financial Risk Tolerance", ["Low",
        "Moderate", "High"])
    submitted = st.form_submit_button("Get Recommendation")

if submitted:
    response = recommend_benefits(age, family, priority, risk)
    st.subheader("🧠 Personalized Benefits Recommendation")
    st.write(response)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Recommended Plans:
- Health Plan B for low deductible and routine coverage
- Dental Plan A
- Vision Plan A
- Enroll in 401(k) for employer match

Since you're 30, married with kids, and value predictable healthcare
costs, Plan B will cover your needs with less out-of-pocket risk. The
401(k) match ensures long-term savings. Avoid the HSA since Plan B
doesn't qualify.
```

😊 Agent #17: Employee Sentiment Analysis Agent

📝 Overview

This AI agent helps HR teams monitor employee morale by analyzing text feedback (from surveys, Slack messages, or emails) for emotion, tone, and satisfaction. In this lab, you'll simulate anonymous feedback, analyze it using GPT for sentiment and themes, and generate a summary report.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate open-ended employee feedback data
- Use GPT to extract sentiment (positive, negative, neutral) and key topics
- Generate an HR-friendly summary with recommended actions
- Display analysis results in a clean Streamlit dashboard

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

If you haven't already:

```
mkdir sentiment_analysis_agent
cd sentiment_analysis_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate feedback dataset (`feedback_data.py`)

```
import pandas as pd

def get_feedback():
    return pd.DataFrame({
```

```
        "Employee ID": [101, 102, 103, 104],
        "Feedback": [
            "I feel very supported by my team and manager.",
            "The workload has been overwhelming lately and
        communication is lacking.",
            "Appreciate the new remote work policy, it's a big help.",
            "I'm confused about our goals and wish leadership
        communicated more clearly."
        ]
    })
```

✅ Step 3: GPT Prompt Template (`sentiment_prompt.py`)

```
from langchain.prompts import PromptTemplate

sentiment_template = PromptTemplate.from_template("""
You are an AI sentiment analyst.

Given this employee feedback:
"{feedback}"

1. Classify sentiment as Positive, Negative, or Neutral.
2. Identify the main theme (e.g., management, workload, culture).
3. Suggest one HR action if needed.

Format:
- Sentiment: ...
- Theme: ...
- Action: ...
""")
```

✅ Step 4: Analyze sentiment (`sentiment_agent.py`)

```
from feedback_data import get_feedback
from sentiment_prompt import sentiment_template
from langchain.chat_models import ChatOpenAI

def analyze_feedback():
    df = get_feedback()
    results = []
    llm = ChatOpenAI(temperature=0.3)

    for _, row in df.iterrows():
        prompt = sentiment_template.format(feedback=row["Feedback"])
        result = llm.predict(prompt)
        results.append(result)
```

```
    df["Analysis"] = results
    return df
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from sentiment_agent import analyze_feedback

st.title("😊 Employee Sentiment Analysis Agent")

if st.button("Analyze Feedback"):
    df = analyze_feedback()

    st.subheader("🗣️ Raw Feedback + AI Analysis")
    for i, row in df.iterrows():
        st.markdown(f"**Employee {row['Employee ID']}**")
        st.text(f"Feedback: {row['Feedback']}")
        st.text(f"AI Analysis:\n{row['Analysis']}")
        st.markdown("---")
```

Run it:

```
streamlit run app.py
```

🧪 Example Output:

```
Employee 102

Feedback: "The workload has been overwhelming lately and communication
is lacking."

- Sentiment: Negative
- Theme: Workload & Communication
- Action: HR should schedule a check-in and assess workload
distribution with the team.
```

🔥 Agent #18: Burnout Prediction Agent

📝 Overview

This AI agent identifies early signs of employee burnout by analyzing behavioral signals such as working hours, PTO usage, communication tone, and productivity metrics. In this lab, you'll simulate an employee dataset and use GPT to assess burnout risk and suggest interventions.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate behavioral and productivity data for employees
- Use GPT to evaluate burnout risk
- Generate a summary with suggested HR actions
- Visualize risk levels in a Streamlit interface

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir burnout_agent
cd burnout_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate employee behavioral data (`burnout_data.py`)

```
import pandas as pd

def load_employee_metrics():
    return pd.DataFrame({
        "Employee ID": [201, 202, 203, 204],
        "Avg Weekly Hours": [42, 60, 37, 55],
        "PTO Taken (days last 90d)": [5, 0, 8, 1],
        "Slack Message Sentiment": ["Neutral", "Negative", "Positive",
        "Negative"],
        "Late-Night Emails/week": [2, 7, 0, 6]
    })
```

✅ Step 3: GPT Prompt Template (`burnout_prompt.py`)

```
from langchain.prompts import PromptTemplate
```

```
burnout_template = PromptTemplate.from_template("""
You are a workplace wellness AI.

Based on the data below, assess this employee's burnout risk:

- Weekly Hours: {hours}
- PTO Taken: {pto} days in 90 days
- Message Sentiment: {sentiment}
- Late-Night Emails/Week: {emails}

Please:
1. Classify burnout risk as Low, Moderate, or High.
2. Justify your reasoning.
3. Recommend one specific HR intervention.
""")
```

✅ Step 4: GPT Agent Logic (`burnout_agent.py`)

```python
from burnout_data import load_employee_metrics
from burnout_prompt import burnout_template
from langchain.chat_models import ChatOpenAI

def analyze_burnout():
    df = load_employee_metrics()
    results = []
    llm = ChatOpenAI(temperature=0.3)

    for _, row in df.iterrows():
        prompt = burnout_template.format(
            hours=row["Avg Weekly Hours"],
            pto=row["PTO Taken (days last 90d)"],
            sentiment=row["Slack Message Sentiment"],
            emails=row["Late-Night Emails/week"]
        )
        result = llm.predict(prompt)
        results.append(result)

    df["Burnout Analysis"] = results
    return df
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from burnout_agent import analyze_burnout

st.title("🔥 Burnout Prediction Agent")

if st.button("Run Burnout Scan"):
```

```
    df = analyze_burnout()

    for _, row in df.iterrows():
        st.subheader(f"👴 Employee {row['Employee ID']}")
        st.markdown(f"**Avg Weekly Hours:** {row['Avg Weekly
        Hours']}")
        st.markdown(f"**PTO Taken (90d):** {row['PTO Taken (days last
        90d)']} days")
        st.markdown(f"**Sentiment:** {row['Slack Message
        Sentiment']}")
        st.markdown(f"**Late-Night Emails:** {row['Late-Night
        Emails/week']}/week")
        st.text_area("🧠 Burnout Risk Assessment", row["Burnout
        Analysis"], height=180)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Burnout Risk: High

This employee is working 60 hours per week, has not taken any PTO in
the past 90 days, and frequently sends emails late at night. Their
Slack sentiment is also negative.

Recommendation:
Schedule a well-being check-in with their manager and encourage
immediate PTO usage. Explore workload redistribution options.
```

📈 Agent #19: Performance Review Agent

📝 Overview

This AI agent streamlines performance reviews by generating summaries based on employee goals, feedback, peer reviews, and manager notes. In this lab, you'll simulate performance input data and use GPT to produce well-structured performance review drafts.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate employee performance data (KPIs, feedback, goals)

- Use GPT to draft personalized review narratives
- Summarize strengths, areas of growth, and ratings
- Display in a review-ready Streamlit dashboard

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir performance_review_agent
cd performance_review_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate performance data (`review_data.py`)

```
import pandas as pd

def get_employee_reviews():
    return pd.DataFrame({
        "Employee": ["Jordan", "Maya", "Alex"],
        "Role": ["Data Analyst", "UX Designer", "Product Manager"],
        "KPI Summary": [
            "Completed 5 dashboards, reduced report time by 40%",
            "Led 2 usability studies, improved UX metrics by 15%",
            "Launched 2 products, managed cross-functional team of 12"
        ],
        "Peer Feedback": [
            "Great attention to detail and collaboration",
            "Very empathetic and user-focused",
            "Strong leadership and great communicator"
        ],
        "Manager Comments": [
            "Reliable and consistently meets deadlines",
            "Creative thinker with strong execution",
            "Excellent at driving outcomes under pressure"
        ]
    })
```

✅ Step 3: Create GPT Prompt Template (`review_prompt.py`)

```
from langchain.prompts import PromptTemplate

review_template = PromptTemplate.from_template("""
You are an AI HR assistant drafting employee performance reviews.

Given this employee data:
- Name: {name}
- Role: {role}
- KPI Summary: {kpi}
- Peer Feedback: {peer}
- Manager Comments: {manager}

Generate a performance review with:
1. Strengths summary
2. Development areas (if any)
3. Overall performance rating (Excellent / Good / Needs Improvement)
4. A friendly, professional tone
""")
```

✅ Step 4: Generate GPT Review Drafts (`review_agent.py`)

```
from review_data import get_employee_reviews
from review_prompt import review_template
from langchain.chat_models import ChatOpenAI

def draft_reviews():
    df = get_employee_reviews()
    llm = ChatOpenAI(temperature=0.3)
    outputs = []

    for _, row in df.iterrows():
        prompt = review_template.format(
            name=row["Employee"],
            role=row["Role"],
            kpi=row["KPI Summary"],
            peer=row["Peer Feedback"],
            manager=row["Manager Comments"]
        )
        result = llm.predict(prompt)
        outputs.append((row["Employee"], result))

    return df, outputs
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from review_agent import draft_reviews

st.title("📈 Performance Review Agent")

if st.button("Generate Reviews"):
    df, outputs = draft_reviews()

    st.subheader("👴 Employee Performance Drafts")
    for name, review in outputs:
        st.markdown(f"### {name}")
        st.text_area("Review", review, height=300)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Jordan – Data Analyst Jordan has consistently delivered high-quality dashboards and streamlined reporting processes, reducing time to insight by 40%. Peer and manager feedback highlight his reliability and attention to detail.

Strengths: Analytical rigor, dependable delivery, team collaboration **Development Areas:** Consider leading initiatives for broader impact **Rating:** Excellent

---

🧮 Agent #2: Ledger Agent

📝 Overview

This AI agent monitors, classifies, and reconciles ledger entries (debits/credits) to help finance teams ensure accuracy, detect anomalies, and generate summaries. In this lab, we'll simulate ledger data and build a GPT-powered agent to explain inconsistencies or balance summaries conversationally.

🧪 Lab Objectives

By the end of this lab, you will:

- Ingest a sample ledger as a Pandas DataFrame
- Use GPT to summarize transactions, detect imbalances
- Explain discrepancies and generate reconciliation notes

- Build a UI to chat with the ledger

🧰 Tech Stack

- **Python**
- **OpenAI GPT-4 / GPT-3.5**
- **LangChain**
- **Pandas**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If continuing from the previous lab, stay in the same virtual environment. Otherwise:

```
mkdir ledger_agent
cd ledger_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit pandas
```

✅ Step 2: Create a mock ledger dataset (`ledger_data.py`)

```
import pandas as pd

def get_ledger():
    data = {
        "Date": ["2025-07-01", "2025-07-01", "2025-07-02", "2025-07-
        02", "2025-07-03"],
        "Description": ["Customer Payment", "Revenue Recorded",
        "Office Supplies", "Cash Paid", "Consulting Income"],
        "Account": ["Accounts Receivable", "Revenue", "Office
        Supplies", "Cash", "Revenue"],
        "Debit": [0, 500, 100, 100, 0],
        "Credit": [500, 0, 0, 0, 1500],
    }
    df = pd.DataFrame(data)
    return df
```

✅ Step 3: Create your prompt template (`ledger_prompt.py`)

```
from langchain.prompts import PromptTemplate

ledger_template = PromptTemplate.from_template("""
```

```
You are a financial ledger analyst AI. Analyze the following
        transactions:

{ledger_table}

Tasks:
1. Identify if the debits equal credits.
2. Highlight any imbalances or potential errors.
3. Provide a one-paragraph explanation of the ledger status.
""")
```

✅ Step 4: Format ledger and run GPT analysis (`ledger_agent.py`)

```python
import openai
import pandas as pd
from langchain.chat_models import ChatOpenAI
from ledger_prompt import ledger_template
from ledger_data import get_ledger

def run_ledger_analysis():
    df = get_ledger()
    ledger_table = df.to_string(index=False)

    prompt = ledger_template.format(ledger_table=ledger_table)
    llm = ChatOpenAI(temperature=0.2)
    response = llm.predict(prompt)

    return df, response
```

✅ Step 5: Build Streamlit interface (`app.py`)

```python
import streamlit as st
from ledger_agent import run_ledger_analysis

st.title("Ledger Analysis Agent 📒")
if st.button("Analyze Ledger"):
    df, result = run_ledger_analysis()
    st.subheader("🧾 Ledger Data")
    st.dataframe(df)
    st.subheader("🔍 Analysis Result")
    st.write(result)
```

Launch it:

```
streamlit run app.py
```

🧪 Example Output:

```
Debits and credits are not balanced. The total debit is $200 while the
total credit is $2000.

There appears to be a mismatch—likely due to a missing revenue entry
or a duplicated credit. Please verify the entries for Consulting
Income and Office Supplies. This could cause discrepancies in monthly
close and must be resolved before reporting.
```

🎓 Agent #20: Learning & Development Recommendation Agent

📝 Overview

This AI agent recommends personalized training resources, courses, or skill-building paths based on an employee's current role, career goals, and skill gaps. In this lab, you'll simulate employee profiles and use GPT to generate customized L&D suggestions with justifications.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate employee roles, current skills, and future goals
- Use GPT to generate personalized training plans
- Categorize resources by technical, soft skills, and leadership
- Display a custom L&D dashboard via Streamlit

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir learning_agent
cd learning_agent
```

```
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate employee L&D profile (`employee_ld_data.py`)

```
import pandas as pd

def get_ld_profiles():
    return pd.DataFrame({
        "Employee": ["Jordan", "Mei", "Raj"],
        "Role": ["Data Analyst", "UX Designer", "Engineering
        Manager"],
        "Current Skills": [
            "Excel, SQL, basic Python",
            "Figma, User Research, Wireframing",
            "Team management, Agile, system architecture"
        ],
        "Career Goal": [
            "Become a Machine Learning Engineer",
            "Lead a UX research team",
            "Move into Director of Engineering role"
        ]
    })
```

✅ Step 3: GPT Prompt Template (`ld_prompt.py`)

```
from langchain.prompts import PromptTemplate

ld_template = PromptTemplate.from_template("""
You are an AI Learning & Development advisor.

Given this employee:
- Role: {role}
- Current Skills: {skills}
- Career Goal: {goal}

Recommend a personalized learning plan that includes:
1. Technical skills to acquire
2. Soft skills to strengthen
3. Suggested learning resources (courses or topics)
4. Short summary of how this supports their career growth
""")
```

✅ Step 4: GPT-Powered L&D Generator (`ld_agent.py`)

```python
from employee_ld_data import get_ld_profiles
from ld_prompt import ld_template
from langchain.chat_models import ChatOpenAI

def recommend_ld_plans():
    df = get_ld_profiles()
    llm = ChatOpenAI(temperature=0.4)
    results = []

    for _, row in df.iterrows():
        prompt = ld_template.format(
            role=row["Role"],
            skills=row["Current Skills"],
            goal=row["Career Goal"]
        )
        response = llm.predict(prompt)
        results.append((row["Employee"], response))

    return df, results
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from ld_agent import recommend_ld_plans

st.title("🎓 L&D Recommendation Agent")

if st.button("Generate Learning Plans"):
    df, results = recommend_ld_plans()

    for name, plan in results:
        st.subheader(f"📘 Learning Plan for {name}")
        st.text_area("Recommendations", plan, height=350)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Jordan – Data Analyst Goal: Become a Machine Learning Engineer

- **Technical:** Learn Python for ML, scikit-learn, TensorFlow
- **Soft Skills:** Critical thinking, storytelling with data

- **Resources:** Coursera's "Machine Learning" by Andrew Ng, "DataCamp Python Track", Google ML crash course **Summary:** These skills will help Jordan transition into ML roles by layering on core AI capabilities to their strong data foundation.

🧾 Agent #21: Supplier Risk Assessment Agent

📝 Overview

This AI agent helps procurement teams evaluate and monitor supplier risk by analyzing data such as financials, delivery history, ESG scores, news sentiment, and compliance records. In this lab, you'll simulate supplier profiles, use GPT to generate risk assessments, and provide action-oriented summaries.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate supplier profiles with risk-related data
- Use GPT to classify suppliers as Low, Medium, or High risk
- Extract reasons for risk and recommend mitigation actions
- Display results in a Streamlit dashboard

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir supplier_risk_agent
cd supplier_risk_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Supplier Profile Simulation (`supplier_data.py`)

```
import pandas as pd

def get_suppliers():
    return pd.DataFrame({
        "Supplier": ["AlphaTech", "GreenSource", "LogiXpress",
        "SteelCore"],
        "Delivery Delays (last 6 mo)": [2, 8, 1, 5],
        "Compliance Issues": ["None", "Minor violations", "None",
        "Pending litigation"],
        "ESG Score": [85, 68, 90, 50],
        "Financial Stability": ["Strong", "Average", "Strong",
        "Weak"],
        "Recent News Sentiment": ["Positive", "Neutral", "Positive",
        "Negative"]
    })
```

✅ Step 3: GPT Prompt Template (`risk_prompt.py`)

```
from langchain.prompts import PromptTemplate

risk_template = PromptTemplate.from_template("""
You are a Supplier Risk Analyst AI.

Given this supplier data:
- Delivery Delays: {delays}
- Compliance: {compliance}
- ESG Score: {esg}
- Financial Status: {finance}
- News Sentiment: {sentiment}

Please:
1. Classify risk level (Low / Medium / High)
2. Justify your assessment with 2 key reasons
3. Recommend one action for procurement to take
""")
```

✅ Step 4: Risk Evaluation Logic (`risk_agent.py`)

```
from supplier_data import get_suppliers
from risk_prompt import risk_template
from langchain.chat_models import ChatOpenAI

def assess_supplier_risks():
    df = get_suppliers()
    llm = ChatOpenAI(temperature=0.3)
    results = []
```

```
    for _, row in df.iterrows():
        prompt = risk_template.format(
            delays=row["Delivery Delays (last 6 mo)"],
            compliance=row["Compliance Issues"],
            esg=row["ESG Score"],
            finance=row["Financial Stability"],
            sentiment=row["Recent News Sentiment"]
        )
        result = llm.predict(prompt)
        results.append(result)

    df["AI Risk Assessment"] = results
    return df
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from risk_agent import assess_supplier_risks

st.title("📉 Supplier Risk Assessment Agent")

if st.button("Run Risk Evaluation"):
    df = assess_supplier_risks()

    for _, row in df.iterrows():
        st.subheader(f"🏭 {row['Supplier']}")
        st.markdown(f"- **Delivery Delays:** {row['Delivery Delays
        (last 6 mo)']}")
        st.markdown(f"- **Compliance Issues:** {row['Compliance
        Issues']}")
        st.markdown(f"- **ESG Score:** {row['ESG Score']}")
        st.markdown(f"- **Financial Stability:** {row['Financial
        Stability']}")
        st.markdown(f"- **News Sentiment:** {row['Recent News
        Sentiment']}")
        st.text_area("📋 Risk Summary", row["AI Risk Assessment"],
        height=200)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Supplier: SteelCore

Risk Level: High **Reasoning:**

- Weak financial stability and ongoing litigation signal operational risk.
- Negative news sentiment may indicate reputational challenges.

Action: Avoid long-term contracts until legal clarity is obtained and consider alternative sourcing.

📜 Agent #22: Contract Analysis Agent

📝 Overview

This AI agent reads and analyzes legal and procurement contracts to identify key terms, risks, renewal dates, and compliance clauses. In this lab, you'll simulate contract text inputs and use GPT to extract structured summaries for decision-makers.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate sample contract excerpts
- Use GPT to identify key clauses (e.g., termination, renewal, liability)
- Classify potential risks or missing terms
- Display contract summaries and red flags in a Streamlit dashboard

---

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir contract_analysis_agent
cd contract_analysis_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Create sample contracts (`contract_data.py`)

```
contracts = {
    "Contract A": """
This agreement is valid for a period of 12 months from the date of
        signing. Either party may terminate the agreement with a 30-
        day written notice. Payment terms are Net 60. There is no
        indemnification clause specified.
""",
    "Contract B": """
The vendor shall deliver services per the agreed SLA. Contract renews
        automatically unless cancelled 60 days prior to the end date.
        Payment is due within 30 days of invoice. Liability is capped
        at the total contract value.
""",
    "Contract C": """
This contract is perpetual unless terminated due to breach of terms.
        The client reserves the right to audit vendor data. No
        specified dispute resolution mechanism exists in this
        agreement.
"""
}

def get_contracts():
    return contracts
```

✅ Step 3: GPT Prompt Template (`contract_prompt.py`)

```
from langchain.prompts import PromptTemplate

contract_template = PromptTemplate.from_template("""
You are an AI legal analyst reviewing contract text.

Given this contract:
"{contract_text}"

Please extract and summarize:
1. Contract duration and renewal terms
2. Termination conditions
3. Liability or indemnity clauses
4. Payment terms
5. Any missing or risky clauses (e.g., dispute resolution, audit
        rights)

Respond in bullet points.
""")
```

✅ Step 4: Contract Analysis Logic (`contract_agent.py`)

```python
from contract_data import get_contracts
from contract_prompt import contract_template
from langchain.chat_models import ChatOpenAI

def analyze_contracts():
    contracts = get_contracts()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for name, text in contracts.items():
        prompt = contract_template.format(contract_text=text)
        result = llm.predict(prompt)
        results.append((name, result))

    return results
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from contract_agent import analyze_contracts

st.title("📜 Contract Analysis Agent")

if st.button("Analyze Contracts"):
    results = analyze_contracts()

    for name, summary in results:
        st.subheader(f"📄 {name}")
        st.text_area("Contract Summary", summary, height=300)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Contract A Summary:

- Duration: 12 months
- Termination: 30-day written notice
- Payment: Net 60
- Risk: No indemnification clause – potential exposure
- Missing: No mention of renewal policy

🧾 Agent #23: Purchase Order Automation Agent

📝 Overview

This AI agent automates the creation and validation of purchase orders (POs) by extracting product, vendor, and pricing details from procurement requests or contracts. In this lab, you'll simulate incoming procurement requests and use GPT to auto-generate structured purchase orders for approval.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate unstructured procurement requests
- Use GPT to extract order details and generate PO drafts
- Validate required fields and vendor match
- Display formatted POs via Streamlit

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir po_automation_agent
cd po_automation_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate procurement requests (`request_data.py`)

```
requests = {
    "Request A": """
Please order 50 ergonomic chairs from ErgoFurnish Ltd at $120 per
        chair. Need delivery by end of the month. Include taxes and
        shipping in total.
""",
```

```
    "Request B": """
We need 10 high-speed printers from PrintMax Inc., model PX900, cost
        $450 each. Please initiate the purchase with 5-day delivery
        terms.
""",
    "Request C": """
Requesting 25 licenses of AcmeDesign Pro software (annual plan) at
        $99/license. Vendor is AcmeDesign Software. Immediate delivery
        preferred.
"""
}

def get_requests():
    return requests
```

✅ Step 3: GPT Prompt Template (`po_prompt.py`)

```
from langchain.prompts import PromptTemplate

po_template = PromptTemplate.from_template("""
You are an AI purchase order generator.

Given this procurement request:
"{request_text}"

Extract and generate a structured purchase order with:
- Vendor Name
- Item(s) Description
- Quantity
- Unit Price
- Total Cost
- Delivery Terms
- Notes/Additional Instructions
Respond in JSON format.
""")
```

✅ Step 4: GPT Automation Logic (`po_agent.py`)

```
from request_data import get_requests
from po_prompt import po_template
from langchain.chat_models import ChatOpenAI

def generate_pos():
    requests = get_requests()
    llm = ChatOpenAI(temperature=0.2)
    results = []

    for name, text in requests.items():
        prompt = po_template.format(request_text=text)
```

```
        result = llm.predict(prompt)
        results.append((name, result))

    return results
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from po_agent import generate_pos

st.title("📄 Purchase Order Automation Agent")

if st.button("Generate Purchase Orders"):
    results = generate_pos()

    for name, po_json in results:
        st.subheader(f"📄 {name}")
        st.text_area("Generated Purchase Order", po_json, height=300)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
{
  "Vendor Name": "ErgoFurnish Ltd",
  "Item(s)": "Ergonomic Chair",
  "Quantity": 50,
  "Unit Price": 120,
  "Total Cost": 6000,
  "Delivery Terms": "End of the month",
  "Notes": "Include taxes and shipping"
}
```

🤝 Agent #24: Vendor Engagement Agent

📝 Overview

This AI agent streamlines communication with vendors by generating personalized follow-ups, reminders, onboarding instructions, or feedback requests. In this lab, you'll simulate vendor profiles and interactions, and use GPT to draft dynamic, context-aware engagement emails.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate vendor profiles with engagement context
- Use GPT to generate follow-up or onboarding messages
- Personalize tone based on relationship type and status
- Display drafts in a vendor communication dashboard (Streamlit)

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir vendor_engagement_agent
cd vendor_engagement_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Vendor Profile Simulation (`vendor_data.py`)

```
import pandas as pd

def get_vendors():
    return pd.DataFrame({
        "Vendor": ["Acme Logistics", "GreenSource Supplies",
        "DataGuard Inc."],
        "Engagement Type": ["Follow-up on proposal", "Onboarding
        instructions", "Request for performance feedback"],
        "Relationship Status": ["New", "Long-term", "Preferred"],
        "Last Interaction": [
            "Submitted proposal 2 weeks ago, no response.",
            "Contract signed, waiting on kickoff documents.",
            "Completed Q2 delivery, request post-project feedback."
        ]
    })
```

✅ Step 3: GPT Prompt Template (`vendor_prompt.py`)

```
from langchain.prompts import PromptTemplate

vendor_template = PromptTemplate.from_template("""
You are a Vendor Relationship Manager AI.

Given:
- Vendor: {vendor}
- Engagement Type: {engagement}
- Relationship Status: {status}
- Last Interaction Notes: {interaction}

Generate a professional and polite email message for the engagement
        context.
""")
```

✅ Step 4: Engagement Generator (`vendor_agent.py`)

```
from vendor_data import get_vendors
from vendor_prompt import vendor_template
from langchain.chat_models import ChatOpenAI

def generate_messages():
    df = get_vendors()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for _, row in df.iterrows():
        prompt = vendor_template.format(
            vendor=row["Vendor"],
            engagement=row["Engagement Type"],
            status=row["Relationship Status"],
            interaction=row["Last Interaction"]
        )
        result = llm.predict(prompt)
        results.append((row["Vendor"], result))

    return df, results
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from vendor_agent import generate_messages

st.title("🤝 Vendor Engagement Agent")

if st.button("Generate Vendor Messages"):
```

```
    df, results = generate_messages()

    for vendor, message in results:
        st.subheader(f"📨 Message to {vendor}")
        st.text_area("Generated Email", message, height=300)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

To: GreenSource Supplies *Subject: Welcome to our Procurement Network*

Hi GreenSource Team, Welcome aboard! We're excited to kick off our partnership. Please find attached the onboarding documents to help you get started. Feel free to reach out with any questions, and let us know your availability for a kickoff call next week.

Best regards, [Your Name] Procurement Operations

🤖 Agent #25: AI-driven Negotiation Agent

📝 Overview

This AI agent supports procurement teams by generating negotiation responses for pricing, delivery terms, and contractual clauses. It can counter vendor offers while maintaining a professional tone and strategic posture. In this lab, you'll simulate vendor proposals and use GPT to craft negotiation replies with rationale.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate incoming vendor proposals
- Use GPT to generate counteroffers and negotiation messages
- Adjust tone (firm, collaborative, concessionary)
- Present output in a Streamlit negotiation interface

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir negotiation_agent
cd negotiation_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate Vendor Offers (`proposal_data.py`)

```
import pandas as pd

def get_vendor_offers():
    return pd.DataFrame({
        "Vendor": ["LogiXpress", "TechNova Ltd", "PrintMax
        Solutions"],
        "Offer Summary": [
            "Proposes $120 per unit for 500 units. Delivery time 30
        days. No bulk discount offered.",
            "Requests upfront payment for 12-month software license at
        $15,000. No cancellation policy.",
            "Quotes $10,000 for maintenance package. Delivery support
        only during weekdays. No penalty clause for SLA breaches."
        ],
        "Negotiation Goal": [
            "Seek 10% discount or faster delivery",
            "Request phased payment or cancellation clause",
            "Include penalty clause and weekend support"
        ],
        "Tone Preference": ["Firm", "Collaborative", "Diplomatic"]
    })
```

✅ Step 3: GPT Prompt Template (`negotiation_prompt.py`)

```
from langchain.prompts import PromptTemplate

negotiation_template = PromptTemplate.from_template("""
You are an AI procurement negotiator.

Given:
- Vendor: {vendor}
- Offer: {offer}
```

```
- Negotiation Objective: {goal}
- Tone: {tone}

Write a professional negotiation reply addressing the vendor. Ensure
        clarity, logic, and strategic positioning.
""")
```

✅ Step 4: AI Negotiation Generator (`negotiation_agent.py`)

```python
from proposal_data import get_vendor_offers
from negotiation_prompt import negotiation_template
from langchain.chat_models import ChatOpenAI

def draft_negotiations():
    df = get_vendor_offers()
    llm = ChatOpenAI(temperature=0.4)
    responses = []

    for _, row in df.iterrows():
        prompt = negotiation_template.format(
            vendor=row["Vendor"],
            offer=row["Offer Summary"],
            goal=row["Negotiation Goal"],
            tone=row["Tone Preference"]
        )
        result = llm.predict(prompt)
        responses.append((row["Vendor"], result))

    return df, responses
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from negotiation_agent import draft_negotiations

st.title("🤖 AI-Driven Negotiation Agent")

if st.button("Generate Negotiation Replies"):
    df, responses = draft_negotiations()

    for vendor, message in responses:
        st.subheader(f"📩 Response to {vendor}")
        st.text_area("Negotiation Draft", message, height=300)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

To: LogiXpress *Subject: Response to Supply Proposal*

Dear LogiXpress Team, Thank you for your proposal. While we appreciate your offer of $120 per unit, we'd like to explore a bulk pricing discount or a faster delivery timeline to better align with our operational goals. Could you propose a revised package that accommodates either?

Best regards, Procurement Lead

💸 Agent #26: Procurement Spend Analysis Agent

📝 Overview

This AI agent analyzes historical procurement spend data to uncover trends, highlight anomalies, flag cost-saving opportunities, and recommend strategic sourcing actions. In this lab, you'll simulate spend data, use GPT to summarize insights, and visualize key findings in a dashboard.

🧪 Lab Objectives

By the end of this lab, you will:

- Load or simulate procurement spend data
- Use GPT to generate natural language insights
- Visualize spend breakdowns by category, vendor, and time
- Display key metrics and recommendations in Streamlit

🧰 Tech Stack

- **Python**
- **Pandas**
- **Matplotlib / Plotly**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir spend_analysis_agent
cd spend_analysis_agent
```

```
python -m venv venv
source venv/bin/activate
pip install pandas matplotlib plotly openai langchain streamlit
```

✅ Step 2: Simulate Spend Data (`spend_data.py`)

```
import pandas as pd

def get_spend_data():
    data = {
        "Month": ["Jan", "Feb", "Mar", "Apr", "May", "Jun"],
        "Category": ["Office Supplies", "IT Hardware", "Logistics",
        "IT Hardware", "Office Supplies", "Logistics"],
        "Vendor": ["Staples", "TechNova", "LogiXpress", "TechNova",
        "Staples", "LogiXpress"],
        "Amount": [4500, 12000, 8000, 11000, 5000, 9500]
    }
    return pd.DataFrame(data)
```

✅ Step 3: GPT Prompt Template (`spend_prompt.py`)

```
from langchain.prompts import PromptTemplate

spend_template = PromptTemplate.from_template("""
You are a procurement data analyst AI.

Given this spend data:
{summary}

Provide:
1. Key spend trends and spikes
2. Vendor concentration concerns
3. Recommendations for cost savings
Limit to 5 bullet points.
""")
```

✅ Step 4: GPT Analysis Logic (`spend_agent.py`)

```
from spend_data import get_spend_data
from spend_prompt import spend_template
from langchain.chat_models import ChatOpenAI

def analyze_spend():
    df = get_spend_data()
    summary = df.groupby(["Category",
        "Vendor"]).sum().reset_index().to_string(index=False)
```

```
    prompt = spend_template.format(summary=summary)
    llm = ChatOpenAI(temperature=0.3)
    result = llm.predict(prompt)

    return df, result
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
import plotly.express as px
from spend_agent import analyze_spend

st.title("🛰️ Procurement Spend Analysis Agent")

if st.button("Run Analysis"):
    df, gpt_insight = analyze_spend()

    st.subheader("📊 Spend Breakdown")
    fig = px.bar(df, x="Vendor", y="Amount", color="Category",
        barmode="group")
    st.plotly_chart(fig)

    st.subheader("🧠 AI-Generated Insights")
    st.text_area("Summary", gpt_insight, height=250)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

- IT Hardware spending dominated by TechNova over 2 months
- Logistics costs increased 18.75% between March and June
- Over-reliance on Staples for office supplies
- Consider negotiating bulk pricing with LogiXpress
- Review TechNova contracts for volume discounts

📈 Agent #27: Supplier Performance Monitoring Agent

📝 Overview

This AI agent tracks and evaluates supplier performance using delivery timeliness, quality metrics, incident reports, and feedback scores. It flags underperforming vendors and suggests corrective actions. In this lab, you'll simulate supplier KPIs and use GPT to generate performance summaries and alerts.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate performance metrics for suppliers
- Use GPT to evaluate and categorize supplier performance
- Generate improvement suggestions for flagged vendors
- Visualize performance dashboards using Streamlit

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**
- **Plotly**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir supplier_monitoring_agent
cd supplier_monitoring_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain plotly streamlit
```

✅ Step 2: Simulate Supplier KPI Data (`supplier_data.py`)

```
import pandas as pd

def get_supplier_kpis():
    return pd.DataFrame({
        "Supplier": ["SteelCore", "EcoLogix", "ByteCom", "GreenWare"],
        "On-time Delivery (%)": [92, 76, 88, 95],
        "Defect Rate (%)": [1.2, 4.8, 2.1, 0.5],
        "Support Responsiveness (1–5)": [4.5, 3.0, 3.8, 4.9],
        "Avg Feedback Score (1–5)": [4.2, 2.8, 3.9, 4.6]
    })
```

✅ Step 3: GPT Prompt Template (`performance_prompt.py`)

```
from langchain.prompts import PromptTemplate
```

```
performance_template = PromptTemplate.from_template("""
You are a Supplier Performance Analyst AI.

Given the following metrics for {supplier}:
- On-time Delivery: {delivery}%
- Defect Rate: {defect}%
- Support Responsiveness: {support}/5
- Feedback Score: {feedback}/5

Please:
1. Assess performance level (Excellent, Good, Needs Improvement)
2. Justify your assessment
3. Suggest 1 corrective or engagement action if needed
Respond in 3 short bullet points.
""")
```

✅ Step 4: GPT Evaluation Logic (`monitoring_agent.py`)

```
from supplier_data import get_supplier_kpis
from performance_prompt import performance_template
from langchain.chat_models import ChatOpenAI

def evaluate_suppliers():
    df = get_supplier_kpis()
    llm = ChatOpenAI(temperature=0.3)
    evaluations = []

    for _, row in df.iterrows():
        prompt = performance_template.format(
            supplier=row["Supplier"],
            delivery=row["On-time Delivery (%)"],
            defect=row["Defect Rate (%)"],
            support=row["Support Responsiveness (1–5)"],
            feedback=row["Avg Feedback Score (1–5)"]
        )
        result = llm.predict(prompt)
        evaluations.append((row["Supplier"], result))

    return df, evaluations
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
import plotly.express as px
from monitoring_agent import evaluate_suppliers

st.title("📈 Supplier Performance Monitoring Agent")
```

```
if st.button("Evaluate Suppliers"):
    df, evaluations = evaluate_suppliers()

    st.subheader("📊 Performance Overview")
    fig = px.bar(df, x="Supplier", y="On-time Delivery (%)",
        color="Avg Feedback Score (1–5)")
    st.plotly_chart(fig)

    st.subheader("🧠 AI Assessments")
    for supplier, assessment in evaluations:
        st.subheader(f"{supplier}")
        st.text_area("Performance Summary", assessment, height=200)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (SteelCore):

- Performance: **Excellent**
- Justification: High delivery rate (92%), low defects (1.2%), strong feedback
- Suggestion: Maintain consistent communication to preserve performance

EcoLogix:

- Performance: **Needs Improvement**
- Justification: Delivery delays (76%) and high defect rate (4.8%)
- Suggestion: Schedule a quarterly review to address quality control

---

⚖️ Agent #28: Compliance and Regulatory Agent

📝 Overview

This AI agent analyzes internal documents, vendor contracts, or audit reports to flag compliance gaps, check for alignment with regulations, and summarize risk exposure. In this lab, you'll simulate policy excerpts and vendor data, then use GPT to identify compliance issues and generate regulatory summaries.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate compliance-related documents and vendor records
- Use GPT to check regulatory alignment (e.g., GDPR, SOX, HIPAA)
- Flag missing clauses or risky patterns

- Summarize findings and suggest remediation actions

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir compliance_agent
cd compliance_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Simulate Compliance Data (`compliance_data.py`)

```
compliance_documents = {
    "Vendor A": """
The vendor collects customer PII but does not provide clear opt-out
        options. Retention policy is loosely defined, with no mention
        of data deletion timelines. No reference to third-party data
        processors.
""",
    "Vendor B": """
The vendor ensures end-to-end encryption, maintains audit logs for 3
        years, and aligns with GDPR guidelines on data minimization
        and consent. DPA is signed and in effect.
""",
    "Vendor C": """
Data storage is offshore, with no clarity on breach notification
        procedures. Policies reference SOX but lack concrete
        enforcement language. Incident response SLA is undefined.
"""
}

def get_documents():
    return compliance_documents
```

✅ Step 3: GPT Prompt Template (`compliance_prompt.py`)

```
from langchain.prompts import PromptTemplate

compliance_template = PromptTemplate.from_template("""
You are an AI compliance auditor.

Given the following document:
"{text}"

Check against general regulatory best practices (GDPR, SOX, HIPAA),
        and identify:
1. Compliance gaps or violations
2. Missing clauses or weak policies
3. Suggested actions to become compliant

Summarize your findings in 3–5 bullet points.
""")
```

✅ Step 4: Compliance Evaluation Logic (`compliance_agent.py`)

```
from compliance_data import get_documents
from compliance_prompt import compliance_template
from langchain.chat_models import ChatOpenAI

def evaluate_compliance():
    docs = get_documents()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for vendor, text in docs.items():
        prompt = compliance_template.format(text=text)
        result = llm.predict(prompt)
        results.append((vendor, result))

    return results
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from compliance_agent import evaluate_compliance

st.title("⚖️ Compliance and Regulatory Agent")

if st.button("Run Compliance Check"):
    results = evaluate_compliance()

    for vendor, summary in results:
```

```
        st.subheader(f"📄 {vendor}")
        st.text_area("Compliance Summary", summary, height=250)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Vendor A:

- Lacks clear opt-out for PII processing — potential GDPR violation
- No data retention or deletion standards
- Suggest including DPA, opt-out procedures, and breach reporting SLA

Vendor C:

- Offshore storage without compliance guarantees
- SOX references present but weak enforcement
- Recommend defining breach response SLAs and regulatory audit coverage

---

📦 Agent #29: Inventory Reduction Agent

📝 Overview

This AI agent helps reduce excess inventory by analyzing product turnover rates, identifying slow-moving SKUs, and suggesting actions like discounting, bundling, or supplier renegotiation. In this lab, you'll simulate inventory data and use GPT to recommend data-driven reduction strategies.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate inventory turnover data
- Identify low-turnover SKUs and surplus items
- Use GPT to suggest tailored reduction strategies
- Display actionable inventory reports in Streamlit

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**

- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir inventory_reduction_agent
cd inventory_reduction_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain streamlit
```

✅ Step 2: Simulate Inventory Data (`inventory_data.py`)

```
import pandas as pd

def get_inventory_data():
    return pd.DataFrame({
        "SKU": ["CHAIR001", "TABLE002", "MONITOR003", "CABLE004",
        "DESK005"],
        "Product Name": ["Ergo Chair", "Meeting Table", "24''
        Monitor", "HDMI Cable", "Standing Desk"],
        "Units in Stock": [320, 50, 190, 800, 70],
        "Monthly Sales Avg": [12, 25, 60, 100, 5],
        "Last Restocked": ["2024-11-01", "2025-03-15", "2025-06-01",
        "2025-05-10", "2024-09-30"]
    })
```

✅ Step 3: GPT Prompt Template (`reduction_prompt.py`)

```
from langchain.prompts import PromptTemplate

reduction_template = PromptTemplate.from_template("""
You are an AI inventory analyst.

Given this inventory item:
- SKU: {sku}
- Product: {name}
- Stock: {stock} units
- Monthly Sales: {sales} units
- Last Restocked: {restocked}

Identify:
1. Is this overstocked? (Yes/No)
2. Suggested actions (e.g., discount, bundle, reduce reorder)
3. Justify briefly
```

```
Respond in 3 concise bullet points.
""")
```

✅ Step 4: Inventory Evaluation Logic (`reduction_agent.py`)

```python
from inventory_data import get_inventory_data
from reduction_prompt import reduction_template
from langchain.chat_models import ChatOpenAI

def reduce_inventory():
    df = get_inventory_data()
    llm = ChatOpenAI(temperature=0.3)
    recommendations = []

    for _, row in df.iterrows():
        prompt = reduction_template.format(
            sku=row["SKU"],
            name=row["Product Name"],
            stock=row["Units in Stock"],
            sales=row["Monthly Sales Avg"],
            restocked=row["Last Restocked"]
        )
        result = llm.predict(prompt)
        recommendations.append((row["Product Name"], result))

    return df, recommendations
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from reduction_agent import reduce_inventory

st.title("📦 Inventory Reduction Agent")

if st.button("Run Inventory Optimization"):
    df, recs = reduce_inventory()

    for product, summary in recs:
        st.subheader(f"📦 {product}")
        st.text_area("AI Recommendation", summary, height=200)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Ergo Chair:

- Overstocked: Yes
- Suggested Action: Offer a 15% discount or bundle with desks
- Justification: Only 12 sold per month vs 320 in stock; old restock date

Standing Desk:

- Overstocked: Yes
- Suggested Action: Freeze reorders and offer promotion
- Justification: Only 5 monthly sales, stock aging since 2024

📈 Agent #3: Advanced Prediction Agent

📝 Overview

This AI agent forecasts future financial trends such as revenue, expenses, or cash flow using historical data. In this lab, you'll build a time series predictor that takes mock revenue data and uses a fine-tuned GPT prompt (or Prophet model) to generate forecast summaries and visualization.

🧪 Lab Objectives

By the end of this lab, you will:

- Load and visualize historical revenue data
- Use GPT to interpret trends and forecast future values
- Generate human-readable summaries and charts
- Present everything in a Streamlit interface

🧰 Tech Stack

- **Python**
- **OpenAI GPT-4 / GPT-3.5**
- **Pandas + Matplotlib**
- **LangChain**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If you're continuing from the previous labs, skip this step.

```
mkdir prediction_agent
cd prediction_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas matplotlib streamlit
```

✅ Step 2: Create mock revenue data (`data_loader.py`)

```
import pandas as pd
import numpy as np

def get_revenue_data():
    dates = pd.date_range(start="2023-01-01", periods=18, freq='M')
    revenue = [10000 + np.random.randint(-1000, 1500) + i*300 for i in
        range(len(dates))]
    df = pd.DataFrame({'Month': dates, 'Revenue': revenue})
    return df
```

✅ Step 3: Define the GPT prompt for forecasting (`forecast_prompt.py`)

```
from langchain.prompts import PromptTemplate

forecast_template = PromptTemplate.from_template("""
You are a financial forecasting analyst. Below is the monthly revenue
        data for the past 18 months:

{revenue_table}

Tasks:
1. Identify growth patterns or seasonality.
2. Forecast revenue for the next 3 months.
3. Provide actionable suggestions based on the forecast.

Respond in business-friendly language.
""")
```

✅ Step 4: Analyze and predict with GPT (`forecast_agent.py`)

```
import pandas as pd
from data_loader import get_revenue_data
```

```
from forecast_prompt import forecast_template
from langchain.chat_models import ChatOpenAI

def run_forecast_analysis():
    df = get_revenue_data()
    revenue_table = df.to_string(index=False)

    prompt = forecast_template.format(revenue_table=revenue_table)
    llm = ChatOpenAI(temperature=0.3)
    forecast_summary = llm.predict(prompt)

    return df, forecast_summary
```

✅ Step 5: Visualize in Streamlit (`app.py`)

```
import streamlit as st
import matplotlib.pyplot as plt
from forecast_agent import run_forecast_analysis

st.title("📊 Advanced Prediction Agent")

if st.button("Run Forecast"):
    df, summary = run_forecast_analysis()

    # Line Chart
    st.subheader("📈 Revenue Over Time")
    fig, ax = plt.subplots()
    ax.plot(df["Month"], df["Revenue"], marker='o')
    ax.set_xlabel("Month")
    ax.set_ylabel("Revenue")
    ax.set_title("Historical Revenue Trend")
    st.pyplot(fig)

    # Summary
    st.subheader("🧠 AI Forecast Summary")
    st.write(summary)
```

Launch the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Over the past 18 months, revenue has steadily increased by
approximately $300/month with occasional seasonal dips in July and
December. Based on this trend, projected revenues for the next 3
months are: $19,800, $20,200, and $20,600.
```

```
Recommendation: Consider allocating additional marketing budget for
Q4, when growth tends to spike. Monitor expense-to-revenue ratio to
preserve margins.
```

📊 Agent #30: Demand Forecasting Agent

📝 Overview

This AI agent predicts future product demand using historical sales data, seasonality patterns, and recent trends. It helps reduce stockouts and overstock by providing actionable forecasts for procurement and planning. In this lab, you'll simulate past sales data and use GPT to generate demand forecasts with natural language summaries and visuals.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate monthly sales data for multiple products
- Use GPT to generate demand forecasts and trends
- Visualize predictions and confidence ranges
- Display a dashboard with actionable AI summaries

🧰 Tech Stack

- **Python**
- **Pandas + NumPy**
- **LangChain + GPT-4 or GPT-3.5**
- **Matplotlib or Plotly**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir demand_forecasting_agent
cd demand_forecasting_agent
python -m venv venv
source venv/bin/activate
pip install pandas numpy openai langchain matplotlib streamlit
```

✅ Step 2: Simulate Sales Data (`sales_data.py`)

```
import pandas as pd
import numpy as np

def get_sales_data():
    months = pd.date_range(start="2024-01-01", periods=18, freq="M")
    data = {
        "Month": list(months) * 3,
        "Product": ["Ergo Chair"] * 18 + ["HDMI Cable"] * 18 +
        ["Standing Desk"] * 18,
        "Sales": list(np.random.poisson(30, 18)) +
        list(np.random.poisson(100, 18)) + list(np.random.poisson(10,
        18))
    }
    return pd.DataFrame(data)
```

✅ Step 3: GPT Prompt Template (`forecast_prompt.py`)

```
from langchain.prompts import PromptTemplate

forecast_template = PromptTemplate.from_template("""
You are a demand forecasting AI.

Given the past 18 months of sales for the product "{product}",
        generate:
1. A 3-month demand forecast (Jul–Sep 2025)
2. Commentary on sales trends and seasonality
3. Actionable advice for procurement

Here is the monthly data:
{sales_series}
""")
```

✅ Step 4: Forecast Logic (`forecast_agent.py`)

```
from sales_data import get_sales_data
from forecast_prompt import forecast_template
from langchain.chat_models import ChatOpenAI

def generate_forecasts():
    df = get_sales_data()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for product in df["Product"].unique():
        product_data = df[df["Product"] == product]
```

```
        sales_series = product_data[["Month",
        "Sales"]].to_string(index=False)
        prompt = forecast_template.format(product=product,
        sales_series=sales_series)
        forecast = llm.predict(prompt)
        results.append((product, product_data, forecast))

    return results
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
import matplotlib.pyplot as plt
from forecast_agent import generate_forecasts

st.title("📊 Demand Forecasting Agent")

if st.button("Run Forecasts"):
    results = generate_forecasts()

    for product, df, forecast in results:
        st.subheader(f"📦 {product}")
        st.text_area("AI Forecast Summary", forecast, height=300)

        st.line_chart(df.set_index("Month")["Sales"])
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

HDMI Cable:

- Forecast: July 105 units, August 110 units, September 102 units
- Trend: Steady monthly growth with slight summer peak
- Recommendation: Increase buffer stock in Q3 to avoid stockouts

---

🎯 Agent #31: Lead Scoring Agent

📝 Overview

This AI agent evaluates inbound leads based on criteria like industry, budget, engagement, and firmographics to assign a score indicating sales potential. It helps sales teams prioritize high-quality leads. In this lab, you'll simulate lead data and use GPT to

assign scores with rationale.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate lead attributes (size, activity, budget, etc.)
- Use GPT to assign lead scores and justify them
- Classify leads as hot, warm, or cold
- Display the output in a sales dashboard

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir lead_scoring_agent
cd lead_scoring_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain streamlit
```

✅ Step 2: Simulate Lead Data (`lead_data.py`)

```
import pandas as pd

def get_leads():
    return pd.DataFrame({
        "Company": ["AlphaTech", "BlueWave Inc.", "NovaHealth",
        "CloudUnity"],
        "Industry": ["SaaS", "Manufacturing", "Healthcare", "Cloud
        Services"],
        "Company Size": ["50-100", "500+", "100-250", "10-50"],
        "Recent Activity": ["Visited pricing page, downloaded
        whitepaper",
                            "Attended webinar, requested demo",
                            "Clicked newsletter link",
                            "No recent activity"],
```

```
        "Stated Budget": ["$20,000", "$100,000+", "$50,000", "$5,000"]
    })
```

✅ Step 3: GPT Prompt Template (`lead_prompt.py`)

```python
from langchain.prompts import PromptTemplate

lead_template = PromptTemplate.from_template("""
You are a B2B lead scoring AI.

Given this lead profile:
- Company: {company}
- Industry: {industry}
- Size: {size}
- Activity: {activity}
- Budget: {budget}

Score the lead from 1–100. Classify it as:
- Hot (>80)
- Warm (50–80)
- Cold (<50)

Then explain your rationale in 2–3 sentences.
""")
```

✅ Step 4: Lead Evaluation Logic (`lead_agent.py`)

```python
from lead_data import get_leads
from lead_prompt import lead_template
from langchain.chat_models import ChatOpenAI

def score_leads():
    df = get_leads()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for _, row in df.iterrows():
        prompt = lead_template.format(
            company=row["Company"],
            industry=row["Industry"],
            size=row["Company Size"],
            activity=row["Recent Activity"],
            budget=row["Stated Budget"]
        )
        result = llm.predict(prompt)
        results.append((row["Company"], result))

    return df, results
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from lead_agent import score_leads

st.title("🎯 Lead Scoring Agent")

if st.button("Score Leads"):
    df, results = score_leads()

    for company, summary in results:
        st.subheader(f"🏢 {company}")
        st.text_area("AI Scoring Summary", summary, height=200)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

BlueWave Inc.

- Score: 87 (Hot)
- Rationale: Large manufacturer with strong recent activity and high budget. Demo request shows intent to buy.

CloudUnity

- Score: 42 (Cold)
- Rationale: Small company with no recent engagement and limited budget. Not a priority lead.

🤝 Agent #32: AI Sales Companion Agent

📝 Overview

This agent acts as a real-time assistant for sales reps. It listens to conversations (or reviews meeting summaries), pulls context from CRM and documents, and suggests the next best actions, product info, or deal strategies. In this lab, you'll simulate a sales meeting and use GPT to generate real-time suggestions based on the transcript.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a sales call transcript
- Provide CRM-like context (deal size, persona, product)
- Use GPT to generate in-meeting suggestions
- Display suggestions in a Streamlit dashboard

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**
- *(Optional)*: Whisper for transcription

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_sales_companion
cd ai_sales_companion
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Simulate Call Transcript and Context (`sales_context.py`)

```
call_transcript = """
Sales Rep: Great to meet you! Could you share more about your current
        cloud setup?
Client: Sure. We're currently on AWS but considering GCP for our data
        analytics workloads.
Sales Rep: Interesting. What's prompting the shift?
Client: Mainly cost efficiency and integration with Looker Studio.
"""

deal_context = {
    "Client Name": "NextData Inc.",
    "Deal Stage": "Discovery",
    "Persona": "CTO",
    "Product": "Cloud Cost Optimization Suite",
    "Industry": "Data & Analytics"
}
```

```
def get_call_data():
    return call_transcript, deal_context
```

✅ Step 3: GPT Prompt Template (`sales_prompt.py`)

```
from langchain.prompts import PromptTemplate

sales_prompt = PromptTemplate.from_template("""
You are an AI sales assistant in a live call.

Given:
- Client: {client}
- Deal Stage: {stage}
- Persona: {persona}
- Product: {product}
- Industry: {industry}
- Call Transcript: {transcript}

Suggest:
1. One recommended question to deepen the discussion
2. One product capability to highlight next
3. One follow-up action for the rep
Use bullet points.
""")
```

✅ Step 4: Sales Suggestion Logic (`sales_agent.py`)

```
from sales_context import get_call_data
from sales_prompt import sales_prompt
from langchain.chat_models import ChatOpenAI

def get_sales_assistance():
    transcript, context = get_call_data()
    llm = ChatOpenAI(temperature=0.4)

    prompt = sales_prompt.format(
        client=context["Client Name"],
        stage=context["Deal Stage"],
        persona=context["Persona"],
        product=context["Product"],
        industry=context["Industry"],
        transcript=transcript
    )
    result = llm.predict(prompt)
    return result
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from sales_agent import get_sales_assistance

st.title("🤝 AI Sales Companion Agent")

if st.button("Get Suggestions"):
    suggestions = get_sales_assistance()
    st.subheader("💡 AI Suggestions for the Sales Rep")
    st.text_area("AI Assistant Output", suggestions, height=250)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

- **Question**: “What KPIs are you tracking to evaluate this migration’s success?”
- **Capability**: “Highlight real-time cost anomaly detection and alerts.”
- **Follow-up**: “Send a tailored ROI calculator comparing AWS vs GCP.”

📈 Agent #33: Sales Forecasting Agent

📝 Overview

This AI agent generates forward-looking sales forecasts using historical data, deal pipelines, and trends across products, regions, or reps. It helps teams plan revenue targets, capacity, and strategy. In this lab, you’ll simulate CRM pipeline data and use GPT to create a natural language forecast summary.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate CRM-style pipeline and sales history data
- Use GPT to summarize trends and predict outcomes
- Visualize historical vs projected sales
- Display recommendations in Streamlit

🧰 Tech Stack

- **Python**

- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**
- *(Optional)*: Prophet or statsmodels for numeric forecasting

---

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir sales_forecasting_agent
cd sales_forecasting_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain streamlit
```

✅ Step 2: Simulate Sales Pipeline Data (`sales_data.py`)

```
import pandas as pd

def get_sales_pipeline():
    return pd.DataFrame({
        "Month": ["Apr", "May", "Jun", "Jul", "Aug"],
        "Closed Deals": [220000, 250000, 270000, 0, 0],
        "Pipeline Value": [300000, 280000, 290000, 310000, 330000],
        "Conversion Rate (%)": [75, 78, 81, None, None]
    })
```

✅ Step 3: GPT Prompt Template (`forecast_prompt.py`)

```
from langchain.prompts import PromptTemplate

forecast_template = PromptTemplate.from_template("""
You are an AI sales strategist.

Given the monthly sales performance:
{data}

Please:
1. Forecast expected sales for July and August
2. Identify any trend or seasonality
3. Provide 2 strategic recommendations to close pipeline gaps

Output in a clear, 3-bullet format.
""")
```

✅ Step 4: Sales Forecast Logic (`forecast_agent.py`)

```
from sales_data import get_sales_pipeline
from forecast_prompt import forecast_template
from langchain.chat_models import ChatOpenAI

def generate_forecast():
    df = get_sales_pipeline()
    data_str = df.to_string(index=False)
    llm = ChatOpenAI(temperature=0.3)

    prompt = forecast_template.format(data=data_str)
    result = llm.predict(prompt)

    return df, result
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
import matplotlib.pyplot as plt
from forecast_agent import generate_forecast

st.title("📈 Sales Forecasting Agent")

if st.button("Generate Forecast"):
    df, forecast = generate_forecast()

    st.subheader("📊 Historical Sales Data")
    st.dataframe(df)

    st.subheader("🧠 AI Forecast Summary")
    st.text_area("Forecast", forecast, height=250)

    st.subheader("📉 Chart View")
    fig, ax = plt.subplots()
    ax.plot(df["Month"], df["Closed Deals"], marker='o', label='Closed
        Deals')
    ax.plot(df["Month"], df["Pipeline Value"], marker='x',
        label='Pipeline Value')
    ax.legend()
    st.pyplot(fig)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

- **Forecast**: July: $255,000; August: $265,000 based on rising pipeline
- **Trend**: Steady growth and improving conversion rates
- **Strategy**: Focus on mid-pipeline deals and introduce Q3 incentives for enterprise accounts

---

💬 Agent #34: AI Chatbot for Sales

📝 Overview

This agent acts as an intelligent 24/7 sales assistant on a website or product page. It qualifies leads, answers product questions, recommends offerings, and routes serious prospects to human reps. In this lab, you'll simulate a product catalog and build a GPT-powered sales chatbot that answers customer queries.

🧪 Lab Objectives

By the end of this lab, you will:

- Create a product catalog as chatbot knowledge
- Set up a LangChain-powered chatbot using GPT
- Answer product and pricing questions from sample users
- Deploy it with a simple Streamlit UI

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**
- *(Optional)*: FAISS for long product catalogs

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_sales_chatbot
cd ai_sales_chatbot
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Define Product Catalog (`product_knowledge.py`)

```
products = {
    "Ergo Chair": {
        "description": "Ergonomic office chair with lumbar support and
        adjustable height",
        "price": "$299",
        "best_for": "Remote workers and office teams"
    },
    "SmartDesk 2": {
        "description": "Sit-stand electric desk with programmable
        presets",
        "price": "$499",
        "best_for": "Tech startups and health-conscious professionals"
    },
    "Acoustic Panels": {
        "description": "Sound-dampening panels for video call
        clarity",
        "price": "$99",
        "best_for": "Content creators and sales teams"
    }
}

def get_product_info():
    return products
```

✅ Step 3: GPT Prompt Template (`chatbot_prompt.py`)

```
from langchain.prompts import PromptTemplate

chat_prompt = PromptTemplate.from_template("""
You are a helpful sales assistant. Use the following product catalog
        to answer customer questions.

{catalog}

Customer: {question}

Respond in a friendly, helpful tone.
""")
```

✅ Step 4: Chatbot Engine (`chatbot_agent.py`)

```
from product_knowledge import get_product_info
from chatbot_prompt import chat_prompt
from langchain.chat_models import ChatOpenAI

def handle_chat(query):
```

```
    catalog = get_product_info()
    formatted_catalog = "\n".join([
        f"{name}: {info['description']} | Price: {info['price']} |
        Best for: {info['best_for']}"
        for name, info in catalog.items()
    ])

    prompt = chat_prompt.format(catalog=formatted_catalog,
        question=query)
    llm = ChatOpenAI(temperature=0.4)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Chat Interface (`app.py`)

```
import streamlit as st
from chatbot_agent import handle_chat

st.title("💬 AI Chatbot for Sales")

user_query = st.text_input("Ask me about our products:")

if st.button("Get Response") and user_query:
    reply = handle_chat(user_query)
    st.markdown("**AI Chatbot Says:**")
    st.success(reply)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Interactions:

User: “Which desk helps reduce back pain?” **Bot:** “Our SmartDesk 2 is ideal for reducing back pain with its sit-stand design and programmable presets.”

User: “What’s best for a sales team working from home?” **Bot:** “I’d recommend the Ergo Chair for posture and Acoustic Panels to improve call clarity.”

📄 Agent #35: Automated Proposal Generator

📝 Overview

This agent auto-generates tailored sales proposals based on client data, product details, and pricing rules. It ensures consistency, reduces manual effort, and accelerates closing. In this lab, you’ll simulate a client inquiry and use GPT to generate a customized PDF proposal.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a client's input (industry, needs, budget)
- Combine it with product and pricing data
- Use GPT to generate a proposal
- Export the proposal to a downloadable PDF using `pdfkit` or `xhtml2pdf`

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**
- **PDF generation**: `pdfkit`, `xhtml2pdf`, or `reportlab`

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir proposal_generator_agent
cd proposal_generator_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit pdfkit
brew install wkhtmltopdf  # For macOS (or apt-get install for Linux)
```

✅ Step 2: Define Client & Product Info (`client_data.py`)

```
client_profile = {
    "Client Name": "NextData Inc.",
    "Industry": "Analytics & Cloud",
    "Needs": "Reduce cloud costs, increase DevOps automation",
    "Budget": "$50,000"
}

products = {
    "Cloud Optimization": "Analyzes multi-cloud spend and recommends
        actions",
    "DevOps Toolkit": "CI/CD pipelines, monitoring, and anomaly
        alerts"
}
```

```
def get_client_and_products():
    return client_profile, products
```

✅ Step 3: GPT Prompt Template (`proposal_prompt.py`)

```
from langchain.prompts import PromptTemplate

proposal_prompt = PromptTemplate.from_template("""
You are an enterprise proposal generator.

Client: {client_name}
Industry: {industry}
Needs: {needs}
Budget: {budget}
Products Offered:
{product_info}

Write a 1-page sales proposal with:
1. Executive Summary
2. Recommended Solutions
3. Estimated Cost
4. Contact CTA

Use a professional tone.
""")
```

✅ Step 4: Proposal Generator Logic (`proposal_agent.py`)

```
from client_data import get_client_and_products
from proposal_prompt import proposal_prompt
from langchain.chat_models import ChatOpenAI

def generate_proposal():
    client, products = get_client_and_products()
    product_info = "\n".join([f"- {name}: {desc}" for name, desc in
        products.items()])

    prompt = proposal_prompt.format(
        client_name=client["Client Name"],
        industry=client["Industry"],
        needs=client["Needs"],
        budget=client["Budget"],
        product_info=product_info
    )

    llm = ChatOpenAI(temperature=0.4)
    proposal_text = llm.predict(prompt)
    return proposal_text
```

✅ Step 5: Streamlit Interface + PDF Download (`app.py`)

```
import streamlit as st
import pdfkit
from proposal_agent import generate_proposal

st.title("📄 Automated Proposal Generator")

if st.button("Generate Proposal"):
    proposal = generate_proposal()
    st.text_area("Generated Proposal", proposal, height=400)

    with open("proposal.html", "w") as f:
        f.write(f"<html><body><pre>{proposal}</pre></body></html>")

    pdfkit.from_file("proposal.html", "proposal.pdf")
    with open("proposal.pdf", "rb") as f:
        st.download_button("Download Proposal PDF", f, "proposal.pdf",
        "application/pdf")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (GPT):

Executive Summary NextData Inc. seeks to reduce cloud expenses and improve DevOps efficiency. Our proposal combines Cloud Optimization and DevOps Toolkit to meet these goals within a $50,000 budget.

Recommended Solutions

- Cloud Optimization for real-time multi-cloud spend management
- DevOps Toolkit to automate deployment and monitoring

Estimated Cost: $47,000 **Next Step**: Schedule a demo via [email/phone]

🧠 Agent #36: AI-powered CRM Assistant

📝 Overview

This AI agent automates CRM workflows by summarizing meetings, updating deal notes, suggesting follow-ups, and surfacing key insights. It increases CRM hygiene and ensures sales reps spend more time selling, not typing. In this lab, you'll simulate CRM data and GPT-generated activity summaries.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate CRM deal and meeting records
- Use GPT to generate follow-up suggestions and update summaries
- Display and optionally export the updates to CSV or JSON
- Build a clean Streamlit dashboard

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir crm_assistant_agent
cd crm_assistant_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain streamlit
```

✅ Step 2: Simulate CRM Data (`crm_data.py`)

```
import pandas as pd

def get_deals():
    return pd.DataFrame([
        {
            "Deal ID": "D101",
            "Client": "NextData Inc.",
            "Stage": "Demo Completed",
            "Last Interaction": "2025-07-15",
            "Meeting Notes": "Client interested in pricing details and
        case studies. Mentioned Q3 rollout goal."
        },
        {
            "Deal ID": "D102",
            "Client": "HealthFlow",
            "Stage": "Negotiation",
            "Last Interaction": "2025-07-10",
```

```
        "Meeting Notes": "Asked for deeper security compliance
    details. CTO joined unexpectedly."
    }
])
```

✅ Step 3: GPT Prompt Template (`crm_prompt.py`)

```
from langchain.prompts import PromptTemplate

crm_template = PromptTemplate.from_template("""
You are an AI CRM assistant.

Given this client interaction:
- Client: {client}
- Deal Stage: {stage}
- Notes: {notes}

Generate:
1. A one-line deal summary
2. Suggested follow-up action
3. Tags (comma-separated) for CRM

Respond in 3 labeled bullet points.
""")
```

✅ Step 4: CRM Logic (`crm_agent.py`)

```
from crm_data import get_deals
from crm_prompt import crm_template
from langchain.chat_models import ChatOpenAI

def process_crm_updates():
    df = get_deals()
    llm = ChatOpenAI(temperature=0.3)
    results = []

    for _, row in df.iterrows():
        prompt = crm_template.format(
            client=row["Client"],
            stage=row["Stage"],
            notes=row["Meeting Notes"]
        )
        result = llm.predict(prompt)
        results.append((row["Deal ID"], row["Client"], result))

    return df, results
```

✅ Step 5: Streamlit CRM Dashboard (`app.py`)

```
import streamlit as st
from crm_agent import process_crm_updates

st.title("🧠 AI-powered CRM Assistant")

if st.button("Update CRM"):
    df, results = process_crm_updates()

    for deal_id, client, summary in results:
        st.subheader(f"🔹 Deal: {deal_id} | {client}")
        st.text_area("CRM Update Summary", summary, height=200)
        st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

D101 – NextData Inc.

- **Summary**: Client ready for Q3 rollout, exploring pricing and validation.
- **Follow-up**: Share pricing sheet and customer success story PDF.
- **Tags**: pricing, Q3, case-study, demo-complete

🔁 Agent #37: Sales Follow-up Agent

📝 Overview

This agent ensures timely and personalized follow-ups after a sales interaction. It crafts tailored emails or messages based on meeting notes, buyer persona, and deal stage. In this lab, you'll simulate recent sales conversations and use GPT to generate custom follow-up messages.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate deal context and meeting outcomes
- Use GPT to create 3 follow-up styles (email, LinkedIn message, or internal note)
- Personalize messages by buyer persona and stage
- Display and copy the follow-up text from a dashboard

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir sales_followup_agent
cd sales_followup_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Simulate Sales Context (`followup_data.py`)

```
deal_context = {
    "Client": "NovaHealth",
    "Persona": "COO",
    "Deal Stage": "Post-demo",
    "Meeting Notes": "The client is interested in rollout timing and
        integration with their patient portal. Asked for a timeline
        breakdown and compliance documents."
}

def get_deal_context():
    return deal_context
```

✅ Step 3: GPT Prompt Template (`followup_prompt.py`)

```
from langchain.prompts import PromptTemplate

followup_prompt = PromptTemplate.from_template("""
You are an AI sales assistant.

Write a follow-up message after a sales call using the info below:
- Client: {client}
- Persona: {persona}
- Stage: {stage}
- Notes: {notes}

Generate:
1. A professional email (150–200 words)
2. A LinkedIn-style follow-up message (max 300 characters)
```

```
3. An internal Slack note for the sales team

Label each section clearly.
""")
```

✅ Step 4: Generate Follow-ups (`followup_agent.py`)

```python
from followup_data import get_deal_context
from followup_prompt import followup_prompt
from langchain.chat_models import ChatOpenAI

def generate_followups():
    context = get_deal_context()
    prompt = followup_prompt.format(
        client=context["Client"],
        persona=context["Persona"],
        stage=context["Deal Stage"],
        notes=context["Meeting Notes"]
    )
    llm = ChatOpenAI(temperature=0.4)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```python
import streamlit as st
from followup_agent import generate_followups

st.title("🔁 Sales Follow-up Agent")

if st.button("Generate Follow-up"):
    result = generate_followups()
    st.text_area("📩 AI-Generated Follow-up", result, height=400)
    st.download_button("Download as .txt", result,
        file_name="followup.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Email: Hi [Client Name], Thank you for your time during our recent demo. I've attached a detailed timeline for rollout along with HIPAA compliance documents. We're excited about the potential of integrating with your patient portal and look forward to discussing next steps…

LinkedIn Message: Thanks for the great discussion! Sent over the rollout details and compliance docs—let me know when you're ready to align.

Slack Note: 👋 Just finished demo with NovaHealth COO. They're serious. Shared compliance docs. Need to prep a rollout plan by EOW.

⚖️ Agent #38: AI Contract Negotiation Agent

📝 Overview

This agent supports contract negotiation by reviewing draft terms, highlighting risks, and suggesting improved clauses. It helps sales and legal teams reduce deal friction and turnaround time. In this lab, you'll simulate a contract draft and use GPT to analyze, summarize concerns, and rewrite risky sections.

🧪 Lab Objectives

By the end of this lab, you will:

- Load a draft contract or terms document
- Use GPT to identify negotiable clauses, risks, and inconsistencies
- Suggest revised terms for risk mitigation
- Highlight critical areas in a Streamlit interface

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4**
- **Streamlit**
- *(Optional)*: `pdfplumber` or `PyMuPDF` to parse actual contracts

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir contract_negotiation_agent
cd contract_negotiation_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Sample Contract Text (`contract_text.py`)

```
contract_draft = """
This Service Agreement grants the vendor full data access for
        performance monitoring. The client waives liability in the
        case of indirect losses. Service Level Agreements (SLAs) are
        not guaranteed. Either party may terminate with 10 days'
        notice, without reason. Data will be retained indefinitely.
"""

def get_contract():
    return contract_draft
```

✅ Step 3: GPT Prompt Template (`contract_prompt.py`)

```
from langchain.prompts import PromptTemplate

contract_prompt = PromptTemplate.from_template("""
You are an AI contract reviewer.

Review this contract draft:
{contract_text}

Identify:
1. Clauses that may pose legal or business risk
2. Suggested improved versions of those clauses
3. A summary of the top 3 negotiation priorities

Output as clearly labeled sections.
""")
```

✅ Step 4: Contract Negotiation Logic (`contract_agent.py`)

```
from contract_text import get_contract
from contract_prompt import contract_prompt
from langchain.chat_models import ChatOpenAI

def analyze_contract():
    contract_text = get_contract()
    llm = ChatOpenAI(temperature=0.3)
    prompt = contract_prompt.format(contract_text=contract_text)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from contract_agent import analyze_contract

st.title("⚖️ AI Contract Negotiation Agent")

if st.button("Review Contract"):
    review = analyze_contract()
    st.text_area("📄 Contract Analysis", review, height=500)
    st.download_button("Download Review", review,
        file_name="contract_review.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

⚠️ Risky Clauses Identified:

- Indefinite data retention may violate GDPR.
- Termination with 10-day notice lacks cause requirement.
- Liability waiver is overly broad.

✅ Suggested Rewrites:

- “Data will be retained for no longer than 90 days post-termination.”
- “Termination requires 30-day notice and written justification.”

📌 Top 3 Priorities:

1. Clarify data retention and compliance
2. Strengthen SLA guarantees
3. Require mutual accountability on liability

🔄 Agent #39: Customer Churn Prediction Agent

📝 Overview

This agent analyzes customer behavior and signals to predict the likelihood of churn. It enables proactive retention by surfacing at-risk accounts. In this lab, you’ll simulate customer data, train a basic ML model, and use GPT to generate retention strategy suggestions.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate historical customer data (activity, satisfaction, revenue)
- Train a churn prediction model
- Use GPT to interpret results and suggest retention actions
- Visualize churn risk in Streamlit

🧰 Tech Stack

- **Python**
- **Pandas, scikit-learn**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir churn_prediction_agent
cd churn_prediction_agent
python -m venv venv
source venv/bin/activate
pip install pandas scikit-learn openai langchain streamlit
```

✅ Step 2: Simulate Customer Data (`customer_data.py`)

```python
import pandas as pd
import numpy as np

def get_customer_data():
    np.random.seed(42)
    data = pd.DataFrame({
        "CustomerID": range(1, 101),
        "UsageFrequency": np.random.randint(1, 10, 100),
        "SupportTickets": np.random.randint(0, 5, 100),
        "SatisfactionScore": np.random.randint(1, 6, 100),
        "ContractLengthMonths": np.random.randint(3, 24, 100),
        "Churned": np.random.choice([0, 1], size=100, p=[0.8, 0.2])
    })
    return data
```

✅ Step 3: Train Prediction Model (`churn_model.py`)

```python
from customer_data import get_customer_data
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

def train_churn_model():
    df = get_customer_data()
    X = df[["UsageFrequency", "SupportTickets", "SatisfactionScore",
        "ContractLengthMonths"]]
    y = df["Churned"]

    X_train, X_test, y_train, y_test = train_test_split(X, y,
        test_size=0.2)
    model = RandomForestClassifier()
    model.fit(X_train, y_train)

    df["ChurnRisk"] = model.predict_proba(X)[:, 1]
    return df.sort_values("ChurnRisk",
        ascending=False).reset_index(drop=True)
```

✅ Step 4: GPT Insight Generator (`churn_prompt.py`)

```python
from langchain.prompts import PromptTemplate

churn_prompt = PromptTemplate.from_template("""
You are a customer success analyst.

Based on this churn risk summary:
{summary}

Suggest 3 retention strategies:
- One for high-risk customers
- One for mid-risk
- One for low-risk

Use practical SaaS examples.
""")
```

✅ Step 5: Run GPT Analysis (`churn_agent.py`)

```python
from churn_model import train_churn_model
from churn_prompt import churn_prompt
from langchain.chat_models import ChatOpenAI

def generate_retention_strategies():
    df = train_churn_model().head(10)
    summary = df[["CustomerID", "ChurnRisk"]].to_string(index=False)
```

```
    llm = ChatOpenAI(temperature=0.4)
    prompt = churn_prompt.format(summary=summary)
    suggestions = llm.predict(prompt)

    return df, suggestions
```

✅ Step 6: Streamlit Dashboard (`app.py`)

```
import streamlit as st
from churn_agent import generate_retention_strategies

st.title("🔄 Customer Churn Prediction Agent")

if st.button("Run Churn Analysis"):
    df, suggestions = generate_retention_strategies()

    st.subheader("Top 10 Customers by Churn Risk")
    st.dataframe(df[["CustomerID", "ChurnRisk"]])

    st.subheader("🧠 GPT Retention Suggestions")
    st.text_area("Strategy Summary", suggestions, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Retention Strategies:

- **High Risk**: Assign CSM, offer 20% discount to renew early
- **Mid Risk**: Invite to webinar on advanced features
- **Low Risk**: Send satisfaction survey and loyalty points

🛡️ Agent #4: Fraud Detection Agent

📝 Overview

This agent monitors financial transactions to detect potentially fraudulent behavior using pattern recognition and anomaly detection. In this lab, you'll simulate transaction data, flag suspicious activities using simple heuristics or Z-score analysis, and generate GPT-based explanations for each flag.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate financial transactions with a mix of normal and anomalous entries
- Use statistical and rules-based techniques to detect anomalies
- Use GPT to explain why a transaction was flagged as suspicious
- Present everything via Streamlit in a clean UI

🧰 Tech Stack

- **Python**
- **Pandas + NumPy**
- **OpenAI GPT-4 / GPT-3.5**
- **LangChain**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If you've already done this for previous labs, skip it.

```
mkdir fraud_detection_agent
cd fraud_detection_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas numpy streamlit
```

✅ Step 2: Generate mock transaction data (`transaction_data.py`)

```
import pandas as pd
import numpy as np

def generate_transactions(n=100):
    np.random.seed(42)
    amounts = np.random.normal(loc=200, scale=50, size=n)

    # Inject some fraud-like spikes
    amounts[np.random.randint(0, n, 5)] *= np.random.randint(5, 10)

    transactions = pd.DataFrame({
        "Transaction ID": [f"TX-{i+1:04d}" for i in range(n)],
        "Amount": amounts.round(2),
        "Type": np.random.choice(["Payment", "Refund", "Transfer",
        "Withdrawal"], size=n),
```

```
        "Account": np.random.choice(["A101", "A102", "A103"], size=n)
    })
    return transactions
```

✅ Step 3: Flag anomalies (`fraud_detector.py`)

```
import pandas as pd
import numpy as np
from transaction_data import generate_transactions

def flag_anomalies(df: pd.DataFrame):
    threshold = 3  # Z-score threshold
    mean = df["Amount"].mean()
    std = df["Amount"].std()

    df["Z-Score"] = (df["Amount"] - mean) / std
    df["Flagged"] = df["Z-Score"].abs() > threshold

    return df
```

✅ Step 4: Create GPT-based fraud explanation (`explain_fraud.py`)

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

explanation_prompt = PromptTemplate.from_template("""
You are a forensic accountant AI. A financial transaction has been
        flagged for possible fraud:

Transaction ID: {transaction_id}
Amount: ${amount}
Type: {type}
Account: {account}
Z-Score: {z_score}

Explain why this transaction might be suspicious and what actions a
        finance team should take.
""")

def explain_transaction(row):
    llm = ChatOpenAI(temperature=0.3)
    prompt = explanation_prompt.format(
        transaction_id=row["Transaction ID"],
        amount=row["Amount"],
        type=row["Type"],
        account=row["Account"],
        z_score=round(row["Z-Score"], 2)
```

```
    )
    return llm.predict(prompt)
```

✅ Step 5: Build the Streamlit app (`app.py`)

```
import streamlit as st
from fraud_detector import flag_anomalies
from transaction_data import generate_transactions
from explain_fraud import explain_transaction

st.title("🛡️ Fraud Detection Agent")

if st.button("Run Analysis"):
    df = generate_transactions()
    df = flag_anomalies(df)

    st.subheader("🔍 All Transactions")
    st.dataframe(df)

    flagged_df = df[df["Flagged"]]
    st.subheader("🚨 Flagged Transactions")
    st.dataframe(flagged_df)

    if not flagged_df.empty:
        st.subheader("💬 AI Explanation")
        selected_id = st.selectbox("Choose a flagged Transaction ID",
        flagged_df["Transaction ID"].tolist())
        row = flagged_df[flagged_df["Transaction ID"] ==
        selected_id].iloc[0]
        explanation = explain_transaction(row)
        st.write(explanation)
```

Run it:

```
streamlit run app.py
```

🧪 Example Output:

```
Transaction TX-0042 involves a payment of $1470.23, which is
significantly higher than average based on its Z-score of 6.2. Such
spikes often suggest mistaken data entry, fraud, or misuse. Finance
teams should verify source documentation, validate the recipient, and
cross-reference timing with other system logs before releasing funds.
```

🗺️ Agent #40: Sales Territory Intelligence Agent

📝 Overview

This agent analyzes performance by sales territory, surfaces regional trends, and recommends actions to improve rep allocation, targeting, and quotas. In this lab, you'll simulate territory-based sales data, visualize patterns, and use GPT to generate actionable insights per region.

🧪 Lab Objectives

By the end of this lab, you will:

- Create a dataset of sales by territory and rep
- Analyze key performance indicators (KPIs)
- Use GPT to suggest improvements and realignment strategies
- Visualize territory heatmaps in Streamlit

🧰 Tech Stack

- **Python**
- **Pandas, Plotly**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir territory_intelligence_agent
cd territory_intelligence_agent
python -m venv venv
source venv/bin/activate
pip install pandas openai langchain streamlit plotly
```

✅ Step 2: Simulate Territory Sales Data (`territory_data.py`)

```
import pandas as pd

def get_sales_territory_data():
    return pd.DataFrame([
        {"Territory": "Northeast", "Rep": "Alice", "Quota": 120000,
        "Sales": 95000},
```

```
        {"Territory": "Southeast", "Rep": "Bob", "Quota": 100000,
        "Sales": 112000},
        {"Territory": "Midwest", "Rep": "Carlos", "Quota": 90000,
        "Sales": 87000},
        {"Territory": "West", "Rep": "Dana", "Quota": 110000, "Sales":
        134000},
        {"Territory": "Southwest", "Rep": "Eva", "Quota": 95000,
        "Sales": 67000}
    ])
```

✅ Step 3: GPT Prompt Template (`territory_prompt.py`)

```
from langchain.prompts import PromptTemplate

territory_prompt = PromptTemplate.from_template("""
You are a regional sales strategist.

Here is the territory performance summary:
{territory_table}

Analyze:
1. Which reps or territories are underperforming?
2. Where are there overachievements?
3. Recommend reallocation, retraining, or quota changes.

Respond in 3 clear sections.
""")
```

✅ Step 4: GPT-Based Analysis (`territory_agent.py`)

```
from territory_data import get_sales_territory_data
from territory_prompt import territory_prompt
from langchain.chat_models import ChatOpenAI

def generate_territory_insights():
    df = get_sales_territory_data()
    formatted_table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.4)
    prompt = territory_prompt.format(territory_table=formatted_table)
    result = llm.predict(prompt)

    return df, result
```

✅ Step 5: Streamlit Visualization (`app.py`)

```
import streamlit as st
import plotly.express as px
from territory_agent import generate_territory_insights

st.title("🗺️ Sales Territory Intelligence Agent")

if st.button("Analyze Territories"):
    df, gpt_result = generate_territory_insights()

    st.subheader("📊 Territory Sales Performance")
    df["Performance (%)"] = (df["Sales"] / df["Quota"]) * 100
    st.dataframe(df)

    st.subheader("🌍 Visualize Territory Performance")
    fig = px.bar(df, x="Territory", y="Performance (%)", color="Rep",
        title="Sales vs Quota")
    st.plotly_chart(fig)

    st.subheader("🧠 GPT Recommendations")
    st.text_area("Insights", gpt_result, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Underperforming:

- Southwest territory is under 71% of quota. Eva may need coaching or a territory reassignment.
- Midwest is slightly under but consistent.

Overachievers:

- Dana exceeded quota by 22%. Consider additional accounts or a leadership role.

Recommendations:

- Adjust quotas based on market size
- Add inside sales support in Southwest
- Incentivize referrals in Northeast

✍️ Agent #41: Content Generation Agent

📝 Overview

This agent creates original marketing content such as blog posts, product descriptions, LinkedIn updates, and more—based on brief prompts or campaign needs. In this lab, you'll build an AI writer that generates posts for different formats and tones based on user input.

🧪 Lab Objectives

By the end of this lab, you will:

- Define input prompts (topic, target audience, format, tone)
- Use GPT to generate different types of content (e.g., blog, tweet, email)
- Provide a Streamlit interface to preview and download content
- Optionally, include word count and tone controls

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir content_generation_agent
cd content_generation_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Define Prompt Template (`content_prompt.py`)

```
from langchain.prompts import PromptTemplate

content_prompt = PromptTemplate.from_template("""
You are a creative AI content writer.

Write a {format} on the topic: "{topic}"
Audience: {audience}
```

```
Tone: {tone}
Word count target: {length} words

Make it engaging, relevant, and well-structured.
""")
```

✅ Step 3: Content Generator Logic (`content_agent.py`)

```python
from content_prompt import content_prompt
from langchain.chat_models import ChatOpenAI

def generate_content(topic, audience, format, tone, length):
    prompt = content_prompt.format(
        topic=topic,
        audience=audience,
        format=format,
        tone=tone,
        length=length
    )
    llm = ChatOpenAI(temperature=0.6)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit UI (`app.py`)

```python
import streamlit as st
from content_agent import generate_content

st.title("✍️ AI Content Generation Agent")

topic = st.text_input("Enter topic:", "AI in healthcare")
audience = st.text_input("Target audience:", "healthcare executives")
format = st.selectbox("Content format:", ["Blog Post", "LinkedIn
        Post", "Email", "Product Description"])
tone = st.selectbox("Tone:", ["Professional", "Conversational",
        "Witty", "Urgent"])
length = st.slider("Word count target:", 50, 1000, 300)

if st.button("Generate Content"):
    content = generate_content(topic, audience, format, tone, length)
    st.subheader("📝 Generated Content")
    st.text_area("Content", content, height=400)
    st.download_button("Download as .txt", content,
        file_name="content.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (Blog Post – Conversational Tone – 300 words):

Topic: AI in healthcare **Audience**: Healthcare executives **Tone**: Conversational **Format**: Blog Post

Excerpt: AI is reshaping healthcare—from diagnostic imaging to predictive analytics. But beyond the buzzwords lies a real opportunity: saving lives, reducing burnout, and optimizing operations. If you're a healthcare leader wondering where to begin, the answer isn't complex tech—it's identifying where inefficiencies hurt most…

📈 Agent #42: Campaign Optimization Agent

📝 Overview

This agent analyzes past marketing campaigns and recommends optimizations based on performance data—suggesting budget shifts, audience tweaks, A/B test ideas, and content changes. In this lab, you'll simulate campaign metrics and use GPT to generate insights and improvement strategies.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate campaign performance data (CTR, CPC, ROAS, etc.)
- Visualize key metrics and trends
- Use GPT to generate optimization recommendations
- Highlight underperforming areas and suggest A/B tests

🧰 Tech Stack

- **Python**
- **Pandas, Plotly**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir campaign_optimization_agent
cd campaign_optimization_agent
python -m venv venv
```

```
source venv/bin/activate
pip install pandas plotly openai langchain streamlit
```

✅ Step 2: Simulate Campaign Data (`campaign_data.py`)

```
import pandas as pd

def get_campaign_data():
    return pd.DataFrame([
        {"Campaign": "Summer Promo", "Channel": "Google Ads", "CTR":
        0.045, "CPC": 1.2, "ROAS": 2.8, "Spend": 1500},
        {"Campaign": "Webinar Push", "Channel": "LinkedIn", "CTR":
        0.022, "CPC": 3.0, "ROAS": 1.5, "Spend": 2000},
        {"Campaign": "Holiday Sale", "Channel": "Facebook", "CTR":
        0.065, "CPC": 0.8, "ROAS": 4.0, "Spend": 1800},
        {"Campaign": "Launch Teaser", "Channel": "Email", "CTR": 0.12,
        "CPC": 0.0, "ROAS": 5.1, "Spend": 500},
        {"Campaign": "Nurture Sequence", "Channel": "Email", "CTR":
        0.06, "CPC": 0.0, "ROAS": 2.2, "Spend": 700}
    ])
```

✅ Step 3: GPT Prompt Template (`campaign_prompt.py`)

```
from langchain.prompts import PromptTemplate

campaign_prompt = PromptTemplate.from_template("""
You are a digital marketing analyst.

Analyze the following campaign performance data:
{campaign_table}

Identify:
1. Campaigns with low performance (based on CTR, CPC, ROAS)
2. Suggested changes (budget reallocation, creative type, A/B testing)
3. Optimized strategy for next month

Format the response as a bulleted report.
""")
```

✅ Step 4: GPT Insight Engine (`campaign_agent.py`)

```
from campaign_data import get_campaign_data
from campaign_prompt import campaign_prompt
from langchain.chat_models import ChatOpenAI

def generate_campaign_insights():
    df = get_campaign_data()
```

```
    formatted_table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = campaign_prompt.format(campaign_table=formatted_table)
    insights = llm.predict(prompt)

    return df, insights
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
import plotly.express as px
from campaign_agent import generate_campaign_insights

st.title("📈 Campaign Optimization Agent")

if st.button("Analyze Campaigns"):
    df, insights = generate_campaign_insights()

    st.subheader("📊 Performance Metrics")
    st.dataframe(df)

    fig = px.bar(df, x="Campaign", y="ROAS", color="Channel",
        title="ROAS by Campaign")
    st.plotly_chart(fig)

    st.subheader("🧠 GPT Optimization Suggestions")
    st.text_area("AI Recommendations", insights, height=350)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Underperformers:

- *Webinar Push (LinkedIn)*: High CPC and low CTR
- *Summer Promo (Google Ads)*: Below average ROAS

Suggestions:

- Shift 20% of spend from LinkedIn to Facebook
- A/B test ad copy for Summer Promo
- Double down on Email campaigns (high ROI, low spend)

Next Month Strategy: Focus on Email + Facebook. Add urgency banners to increase CTR on Google Ads.

🧩 Agent #43: Customer Segmentation Agent

📝 Overview

This agent groups customers into meaningful segments based on behavior, demographics, or engagement metrics. It enables targeted marketing, product personalization, and lifecycle planning. In this lab, you'll cluster customer data and use GPT to explain each segment's traits and value.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate customer behavior data (spend, visits, region, loyalty)
- Use clustering (K-Means) to segment customers
- Visualize clusters with color-coded plots
- Use GPT to describe each segment's behavior and suggest targeted actions

---

🧰 Tech Stack

- **Python**
- **Pandas, scikit-learn, Plotly**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir customer_segmentation_agent
cd customer_segmentation_agent
python -m venv venv
source venv/bin/activate
pip install pandas scikit-learn plotly openai langchain streamlit
```

✅ Step 2: Generate Customer Data (`customer_data.py`)

```
import pandas as pd
import numpy as np

def get_customer_df():
    np.random.seed(42)
    data = pd.DataFrame({
```

```
        "CustomerID": range(1, 101),
        "AvgSpend": np.random.uniform(50, 500, 100),
        "VisitsPerMonth": np.random.randint(1, 15, 100),
        "Region": np.random.choice(["North", "South", "East", "West"],
        100),
        "LoyaltyScore": np.random.randint(1, 11, 100)
    })
    return data
```

✅ Step 3: Apply K-Means Clustering (`segmentation_model.py`)

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from customer_data import get_customer_df

def segment_customers(n_clusters=3):
    df = get_customer_df()
    features = df[["AvgSpend", "VisitsPerMonth", "LoyaltyScore"]]
    scaled = StandardScaler().fit_transform(features)

    model = KMeans(n_clusters=n_clusters, random_state=42)
    df["Segment"] = model.fit_predict(scaled)

    return df
```

✅ Step 4: GPT Description of Segments (`segmentation_prompt.py`)

```
from langchain.prompts import PromptTemplate

segment_prompt = PromptTemplate.from_template("""
You are a customer segmentation expert.

Based on the customer summary below:
{segment_summary}

Describe each segment:
- What defines it
- What marketing strategies would work best

Label each segment clearly.
""")
```

✅ Step 5: GPT Analysis (`segmentation_agent.py`)

```
from segmentation_model import segment_customers
from segmentation_prompt import segment_prompt
from langchain.chat_models import ChatOpenAI
```

```
def analyze_segments():
    df = segment_customers()
    summary = df.groupby("Segment")[["AvgSpend", "VisitsPerMonth",
        "LoyaltyScore"]].mean().to_string()

    llm = ChatOpenAI(temperature=0.4)
    prompt = segment_prompt.format(segment_summary=summary)
    response = llm.predict(prompt)

    return df, response
```

✅ Step 6: Streamlit Interface (`app.py`)

```
import streamlit as st
import plotly.express as px
from segmentation_agent import analyze_segments

st.title("🧩 Customer Segmentation Agent")

if st.button("Segment Customers"):
    df, gpt_desc = analyze_segments()

    st.subheader("🔍 Customer Segments")
    st.dataframe(df)

    fig = px.scatter(df, x="AvgSpend", y="VisitsPerMonth",
        color="Segment",
                     size="LoyaltyScore", hover_data=["CustomerID",
        "Region"])
    st.plotly_chart(fig)

    st.subheader("🧠 GPT Segment Descriptions")
    st.text_area("Insights", gpt_desc, height=400)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Segment 0: Budget Loyalists

- Moderate visits, low spend, high loyalty
- Use rewards programs, exclusive points offers

Segment 1: High-Value Frequent Buyers

- High spend and high visit frequency
- Premium upsells and personalized bundles recommended

Segment 2: One-Time Shoppers

- Low spend, low loyalty
- Target re-engagement via discount emails and seasonal flash sales

---

📬 Agent #44: Automated Outreach Agent

📝 Overview

This agent automates the creation and scheduling of outreach emails or messages to different customer segments. It generates personalized outreach based on segment traits, lifecycle stage, and past behavior. In this lab, you'll simulate an outreach pipeline using GPT and generate custom emails for each segment.

🧪 Lab Objectives

By the end of this lab, you will:

- Use GPT to generate personalized outreach messages
- Simulate multiple customer segments and outreach goals
- Customize email tone, intent (nurture, re-engage, upsell)
- Provide a Streamlit UI to generate and review messages

🧰 Tech Stack

- **Python**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir automated_outreach_agent
cd automated_outreach_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Simulate Segment Data (`segments.py`)

```python
import pandas as pd

def get_segments():
    return pd.DataFrame([
        {"Segment": "New Leads", "Behavior": "Visited pricing page but
        didn't sign up"},
        {"Segment": "High-Value Customers", "Behavior": "Subscribed >
        6 months, high usage"},
        {"Segment": "Dormant Users", "Behavior": "Last login > 90
        days"},
        {"Segment": "Free Trial Users", "Behavior": "Trial ending in 3
        days"}
    ])
```

✅ Step 3: GPT Prompt Template (`outreach_prompt.py`)

```python
from langchain.prompts import PromptTemplate

outreach_prompt = PromptTemplate.from_template("""
You are a marketing automation agent.

Create a personalized {type} outreach message for the following
        segment:
- Segment: {segment}
- Behavior: {behavior}
- Tone: {tone}

Make the message persuasive and action-oriented.
""")
```

✅ Step 4: Message Generator Logic (`outreach_agent.py`)

```python
from outreach_prompt import outreach_prompt
from langchain.chat_models import ChatOpenAI

def generate_outreach(segment, behavior, type, tone):
    prompt = outreach_prompt.format(
        segment=segment,
        behavior=behavior,
        type=type,
        tone=tone
    )
    llm = ChatOpenAI(temperature=0.5)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
import pandas as pd
from segments import get_segments
from outreach_agent import generate_outreach

st.title("📬 Automated Outreach Agent")

segments_df = get_segments()
selected = st.selectbox("Choose Segment", segments_df["Segment"])
behavior = segments_df.loc[segments_df["Segment"] == selected,
        "Behavior"].values[0]

outreach_type = st.selectbox("Outreach Type", ["Email", "LinkedIn
        Message", "SMS"])
tone = st.selectbox("Tone", ["Friendly", "Professional", "Urgent"])

if st.button("Generate Outreach"):
    message = generate_outreach(selected, behavior, outreach_type,
        tone)
    st.subheader("📝 Generated Message")
    st.text_area("Message", message, height=300)
    st.download_button("Download Message", message,
        file_name="outreach_message.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output (for Dormant Users):

Subject: We Miss You! Here's 25% Off to Reignite Your Journey 💡

Body: Hi there, It's been a while since we last saw you. We'd love to have you back with exclusive access to our newest features. Here's 25% off your next subscription if you reactivate this week. Let's pick up where we left off 🚀 [Reactivate Now]

🎯 Agent #45: AI-driven Ad Bidding Agent

📝 Overview

This agent dynamically adjusts bids for digital advertising campaigns using real-time signals like CTR, conversion rates, and budget constraints. In this lab, you'll simulate ad performance data and use a simple AI model to optimize bid suggestions per keyword or segment.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a dataset with ad groups, CTR, conversions, and current bids
- Use a basic model to compute optimal bid adjustments
- Use GPT to explain bidding strategies in plain language
- Visualize adjustments using Streamlit

🧰 Tech Stack

- **Python**
- **Pandas, NumPy**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ad_bidding_agent
cd ad_bidding_agent
python -m venv venv
source venv/bin/activate
pip install pandas numpy openai langchain streamlit
```

✅ Step 2: Simulate Ad Performance Data (`ad_data.py`)

```
import pandas as pd
import numpy as np

def get_ad_data():
    np.random.seed(42)
    return pd.DataFrame([
        {"Keyword": "ai software", "CTR": 0.04, "ConversionRate":
        0.05, "Bid": 1.20},
        {"Keyword": "crm tool", "CTR": 0.06, "ConversionRate": 0.09,
        "Bid": 1.80},
        {"Keyword": "marketing automation", "CTR": 0.03,
        "ConversionRate": 0.04, "Bid": 1.00},
        {"Keyword": "customer data platform", "CTR": 0.07,
        "ConversionRate": 0.12, "Bid": 2.00},
        {"Keyword": "email outreach", "CTR": 0.05, "ConversionRate":
        0.06, "Bid": 1.50}
    ])
```

✅ Step 3: Optimization Logic (`bid_optimizer.py`)

```
from ad_data import get_ad_data

def adjust_bids():
    df = get_ad_data()
    df["Efficiency"] = df["ConversionRate"] / df["Bid"]

    avg_eff = df["Efficiency"].mean()
    df["BidChange"] = df["Efficiency"].apply(lambda x: 0.1 if x >
        avg_eff else -0.1)
    df["OptimizedBid"] = (df["Bid"] + df["BidChange"]).round(2)

    return df
```

✅ Step 4: GPT Prompt Template (`bidding_prompt.py`)

```
from langchain.prompts import PromptTemplate

bidding_prompt = PromptTemplate.from_template("""
You are a digital ads strategist.

Based on the following keyword bidding data:
{bidding_summary}

Explain:
1. Which keywords should increase bids
2. Which should decrease
3. The reasoning behind each decision

Format the output as a recommendation memo.
""")
```

✅ Step 5: GPT Explanation Engine (`bidding_agent.py`)

```
from bid_optimizer import adjust_bids
from bidding_prompt import bidding_prompt
from langchain.chat_models import ChatOpenAI

def generate_bid_recommendations():
    df = adjust_bids()
    summary = df[["Keyword", "CTR", "ConversionRate", "Bid",
        "OptimizedBid"]].to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = bidding_prompt.format(bidding_summary=summary)
    memo = llm.predict(prompt)
```

```
    return df, memo
```

✅ Step 6: Streamlit Dashboard (`app.py`)

```
import streamlit as st
from bidding_agent import generate_bid_recommendations

st.title("🎯 AI-driven Ad Bidding Agent")

if st.button("Optimize Ad Bids"):
    df, memo = generate_bid_recommendations()

    st.subheader("🔍 Ad Bidding Overview")
    st.dataframe(df)

    st.subheader("🧠 GPT Bid Recommendations")
    st.text_area("Strategy Memo", memo, height=400)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Increase Bids:

- “customer data platform”: High CTR and Conversion Rate. Scaling opportunity.
- “crm tool”: Efficient ROI, can support a bid bump.

Decrease Bids:

- “marketing automation”: Low conversions, not cost-effective. Reduce bid by $0.10.

Rationale: Focus budget on high-performing keywords while trimming inefficient ones. Use conversion per dollar as guide.

🧠 Agent #46: Brand Sentiment Monitoring Agent

📝 Overview

This agent continuously tracks how people feel about your brand across channels like social media, reviews, and forums. It performs sentiment analysis, detects spikes in negative feedback, and summarizes brand reputation trends using AI. In this lab, you’ll simulate brand mentions and build a real-time sentiment monitor.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate or input brand mentions with text content
- Perform sentiment analysis using a pre-trained model or GPT
- Visualize sentiment trends over time
- Summarize brand sentiment insights using GPT

🧰 Tech Stack

- **Python**
- **Pandas, Matplotlib or Plotly**
- **Transformers (HuggingFace)** or **OpenAI GPT**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir sentiment_monitoring_agent
cd sentiment_monitoring_agent
python -m venv venv
source venv/bin/activate
pip install pandas matplotlib openai langchain transformers streamlit
```

✅ Step 2: Simulate Brand Mentions (`mentions.py`)

```
import pandas as pd
from datetime import datetime, timedelta
import random

def generate_mentions():
    base_time = datetime.now()
    data = []
    sentiments = ["positive", "neutral", "negative"]
    templates = {
        "positive": ["I love this brand!", "Excellent customer
        service.", "Top quality!"],
        "neutral": ["Just tried it today.", "Okay product, nothing
        special."],
        "negative": ["Terrible support.", "Very disappointed with the
        quality.", "Wouldn't recommend."]
    }

    for i in range(50):
```

```
        sentiment = random.choice(sentiments)
        text = random.choice(templates[sentiment])
        timestamp = base_time - timedelta(hours=random.randint(1, 72))
        data.append({"text": text, "timestamp": timestamp, "label":
        sentiment})

    return pd.DataFrame(data)
```

✅ Step 3: Sentiment Analysis (Simple Rule-Based) (`sentiment_model.py`)

You can either:

- Use the HuggingFace pipeline (`transformers`) for pre-trained sentiment analysis, or
- Use GPT for interpretation.

Option A – Using HuggingFace:

```
from transformers import pipeline

sentiment_pipeline = pipeline("sentiment-analysis")

def analyze_sentiment(text):
    result = sentiment_pipeline(text)[0]
    return result['label'].lower()
```

Option B – Using GPT (optional):

```
from langchain.chat_models import ChatOpenAI

def analyze_sentiment_gpt(text):
    llm = ChatOpenAI(temperature=0.3)
    response = llm.predict(f"What is the sentiment of this brand
        mention: '{text}'? Respond with 'positive', 'neutral', or
        'negative'.")
    return response.strip().lower()
```

✅ Step 4: GPT Insight Summary (`summary_prompt.py`)

```
from langchain.prompts import PromptTemplate

summary_prompt = PromptTemplate.from_template("""
You are a brand analyst AI.

Given this sentiment breakdown:
{sentiment_data}

1. What is the overall brand sentiment?
2. Are there warning signs (e.g., recent spikes in negativity)?
3. Recommend 2 actions for brand reputation improvement.
```

```
Respond in 3 short paragraphs.
""")
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```python
import streamlit as st
import matplotlib.pyplot as plt
from mentions import generate_mentions
from sentiment_model import analyze_sentiment
from summary_prompt import summary_prompt
from langchain.chat_models import ChatOpenAI

st.title("🧠 Brand Sentiment Monitoring Agent")

if st.button("Run Sentiment Analysis"):
    df = generate_mentions()
    df["Sentiment"] = df["text"].apply(analyze_sentiment)
    sentiment_counts = df["Sentiment"].value_counts()

    # Pie chart
    st.subheader("🔎 Sentiment Breakdown")
    fig, ax = plt.subplots()
    ax.pie(sentiment_counts, labels=sentiment_counts.index,
        autopct="%1.1f%%")
    st.pyplot(fig)

    # GPT Summary
    summary_text = df["Sentiment"].value_counts().to_string()
    llm = ChatOpenAI(temperature=0.4)
    gpt_summary =
        llm.predict(summary_prompt.format(sentiment_data=summary_text))

    st.subheader("🧠 GPT Summary & Recommendations")
    st.text_area("Insights", gpt_summary, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output:

Overall Sentiment: The majority of feedback (60%) is positive, showing strong brand appreciation. Users frequently praise quality and support.

Warning Signs: A noticeable 22% of mentions are negative, clustered in the last 24 hours. This may indicate a service disruption or product issue.

Recommended Actions:

1. Quickly investigate the negative spike and issue a public response.
2. Encourage satisfied customers to leave public reviews to outweigh negativity.

🚀 Agent #47: AI-powered SEO Optimization Agent

📝 Overview

This agent analyzes web content and provides SEO optimization suggestions, including keyword improvements, meta description rewrites, title tag enhancements, and semantic content suggestions. In this lab, you'll input sample blog/article content and use GPT to suggest SEO improvements based on target keywords.

🧪 Lab Objectives

By the end of this lab, you will:

- Input or upload webpage/article content
- Specify a target keyword or phrase
- Use GPT to analyze title, meta description, and body content
- Generate SEO suggestions, including keyword density, title rewrites, and content gaps

---

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir seo_optimization_agent
cd seo_optimization_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: SEO Prompt Template (`seo_prompt.py`)

```
from langchain.prompts import PromptTemplate

seo_prompt = PromptTemplate.from_template("""
You are an SEO optimization expert.

Given the following content:
{content}

And the target keyword: "{keyword}"

Provide:
1. A rewritten SEO-optimized title (max 60 characters)
2. An improved meta description (max 155 characters)
3. Keyword usage feedback (density and placement)
4. Suggested headings or sections to improve semantic richness

Format clearly with bullet points.
""")
```

✅ Step 3: GPT Optimization Engine (`seo_agent.py`)

```
from seo_prompt import seo_prompt
from langchain.chat_models import ChatOpenAI

def generate_seo_insights(content, keyword):
    llm = ChatOpenAI(temperature=0.3)
    prompt = seo_prompt.format(content=content, keyword=keyword)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit UI (`app.py`)

```
import streamlit as st
from seo_agent import generate_seo_insights

st.title("🚀 AI-powered SEO Optimization Agent")

content = st.text_area("Paste your content:", height=300)
keyword = st.text_input("Target keyword:", "AI for marketing")

if st.button("Optimize SEO"):
    if content and keyword:
        insights = generate_seo_insights(content, keyword)
        st.subheader("🔍 SEO Suggestions")
        st.text_area("Results", insights, height=400)
```

```
    else:
        st.warning("Please provide content and a target keyword.")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

SEO Title: "Mastering AI for Marketing: Boost Strategy & Results"

Meta Description: "Unlock AI's full potential in marketing. Improve targeting, personalization, and ROI."

Keyword Analysis:

- Target keyword appears 3 times (~0.8% density)
- Appears in body and one heading
- Missing in title and meta (before optimization)

Suggestions:

- Add a section: "How AI Improves Campaign ROI"
- Use related phrases like "AI-driven personalization" and "predictive targeting"
- Include internal links to related blog posts

📧 Agent #48: Personalized Email Marketing Agent

📝 Overview

This agent generates customized marketing emails tailored to customer segments, behavior, and product preferences. In this lab, you'll simulate user data, let the user pick a segment and goal (upsell, onboarding, reactivation), and use GPT to create highly personalized emails with adaptive tone and content.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate or select a user persona or segment
- Choose a campaign goal (e.g., upsell, reactivation)
- Generate a personalized marketing email using GPT
- Customize subject line, tone, CTA, and dynamic content

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir personalized_email_agent
cd personalized_email_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Create Customer Segments (`segments.py`)

```
import pandas as pd

def get_segments():
    return pd.DataFrame([
        {
            "Segment": "New Users",
            "Behavior": "Signed up recently, has not used the product
        much",
            "Preferences": "Interested in tutorials and getting
        started tips"
        },
        {
            "Segment": "Power Users",
            "Behavior": "Use the product daily, explore advanced
        features",
            "Preferences": "Want productivity hacks and power-user
        features"
        },
        {
            "Segment": "Lapsed Users",
            "Behavior": "Hasn't logged in for 60+ days",
            "Preferences": "Need reminders, offers to return, success
        stories"
        },
        {
            "Segment": "Premium Subscribers",
            "Behavior": "Paying customers using premium features",
            "Preferences": "Upsell cross-platform tools, loyalty
        perks"
```

```
        }
    ])
```

✅ Step 3: Email Prompt Template (`email_prompt.py`)

```python
from langchain.prompts import PromptTemplate

email_prompt = PromptTemplate.from_template("""
You are an expert email copywriter.

Write a personalized marketing email for the following customer:

- Segment: {segment}
- Behavior: {behavior}
- Preferences: {preferences}
- Campaign Goal: {goal}
- Tone: {tone}

Include:
1. A compelling subject line
2. A warm greeting
3. Main message tailored to the user type
4. A call to action (CTA)

Keep it friendly, natural, and conversion-oriented.
""")
```

✅ Step 4: GPT Message Generator (`email_agent.py`)

```python
from segments import get_segments
from email_prompt import email_prompt
from langchain.chat_models import ChatOpenAI

def generate_email(segment, behavior, preferences, goal, tone):
    prompt = email_prompt.format(
        segment=segment,
        behavior=behavior,
        preferences=preferences,
        goal=goal,
        tone=tone
    )
    llm = ChatOpenAI(temperature=0.5)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit UI (`app.py`)

```
import streamlit as st
from segments import get_segments
from email_agent import generate_email

st.title("📧 Personalized Email Marketing Agent")

segments_df = get_segments()
segment_names = segments_df["Segment"].tolist()
selected_segment = st.selectbox("Choose Segment", segment_names)
segment_info = segments_df[segments_df["Segment"] ==
        selected_segment].iloc[0]

goal = st.selectbox("Campaign Goal", ["Onboarding", "Reactivation",
        "Upsell", "Announcement"])
tone = st.selectbox("Tone", ["Friendly", "Professional", "Urgent",
        "Playful"])

if st.button("Generate Email"):
    email = generate_email(
        segment_info["Segment"],
        segment_info["Behavior"],
        segment_info["Preferences"],
        goal,
        tone
    )
    st.subheader("📝 Generated Email")
    st.text_area("Email Content", email, height=400)
    st.download_button("Download Email", email,
        file_name="personalized_email.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (for "Lapsed Users", goal = "Reactivation"):

Subject: We Miss You—Here's 25% Off to Come Back!

Hi there, It's been a while since your last visit, and we're saving your seat! Our latest features and success stories are waiting for you—and for this week only, here's 25% off to welcome you back. Ready to dive in again? 👉 [Reactivate Your Account]

Warmly, The [Your Company] Team

📱 Agent #49: Social Media Engagement Agent

📝 Overview

This agent generates and schedules engaging posts across platforms (Twitter, LinkedIn, Instagram), customized by audience, tone, and event type. It also suggests hashtags and responds to user comments or mentions using sentiment-aware responses. In this lab, you'll create post templates, generate GPT-driven posts, and simulate comment replies.

🧪 Lab Objectives

By the end of this lab, you will:

- Generate platform-specific social media posts using GPT
- Customize tone, format, hashtags, and emojis
- Simulate post comment responses using GPT sentiment logic
- Visualize a posting calendar interface in Streamlit

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional)* Scheduling/Publishing APIs like Buffer, Hootsuite, or Zapier

---

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir social_engagement_agent
cd social_engagement_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Create Post Template Prompt (`post_prompt.py`)

```
from langchain.prompts import PromptTemplate

post_prompt = PromptTemplate.from_template("""
You are a social media strategist.

Create a {platform} post for:
```

```
- Topic: {topic}
- Goal: {goal}
- Audience: {audience}
- Tone: {tone}

Include a CTA, 2–3 relevant hashtags, and an emoji or two.
Keep it within platform constraints (e.g., Twitter = 280 characters).
""")
```

✅ Step 3: Comment Reply Prompt (`comment_reply_prompt.py`)

```
from langchain.prompts import PromptTemplate

comment_prompt = PromptTemplate.from_template("""
You are a social media community manager.

Given this user comment: "{comment}"
Respond in a {tone} tone, appropriate to its sentiment ({sentiment}).

Keep the reply short, human, and on-brand.
""")
```

✅ Step 4: GPT Post Generator (`social_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from post_prompt import post_prompt
from comment_reply_prompt import comment_prompt

llm = ChatOpenAI(temperature=0.6)

def generate_post(platform, topic, goal, audience, tone):
    prompt = post_prompt.format(platform=platform, topic=topic,
        goal=goal, audience=audience, tone=tone)
    return llm.predict(prompt)

def respond_to_comment(comment, sentiment, tone):
    prompt = comment_prompt.format(comment=comment,
        sentiment=sentiment, tone=tone)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
from social_agent import generate_post, respond_to_comment

st.title("📱 Social Media Engagement Agent")
```

```
st.subheader("🎯 Post Generator")
platform = st.selectbox("Platform", ["Twitter", "LinkedIn",
        "Instagram"])
topic = st.text_input("Topic", "AI in Education")
goal = st.selectbox("Goal", ["Brand Awareness", "Event Promo",
        "Product Launch", "Lead Generation"])
audience = st.text_input("Audience", "EdTech CEOs and Product
        Managers")
tone = st.selectbox("Tone", ["Friendly", "Professional", "Witty",
        "Inspirational"])

if st.button("Generate Post"):
    post = generate_post(platform, topic, goal, audience, tone)
    st.text_area("Generated Post", post, height=200)

st.subheader("💬 Comment Reply Generator")
user_comment = st.text_input("User Comment", "This tool saved me hours
        of work!")
sentiment = st.selectbox("Comment Sentiment", ["positive", "neutral",
        "negative"])
reply_tone = st.selectbox("Reply Tone", ["Supportive", "Playful",
        "Grateful", "Empathetic"])

if st.button("Generate Reply"):
    reply = respond_to_comment(user_comment, sentiment, reply_tone)
    st.text_area("Generated Reply", reply, height=150)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Post (Twitter): Just launched our AI-powered course builder for educators! 🎓 Start building smarter today 👉 [link] #AIinEducation #EdTech 🚀

Reply (to positive comment): Thank you so much! We're thrilled it's been helpful 🙌 Stay tuned for even more updates soon!

🧾 Agent #5: Automated Invoicing Agent

📝 Overview

This AI agent automatically generates invoices by pulling data from mock orders or timesheets, calculates totals, applies taxes or discounts, and formats the invoice as a human-readable document. You'll build a GPT-based agent that turns structured data into invoice summaries and optionally outputs them as PDFs.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate invoice data from timesheets or purchase orders
- Automatically generate invoice details and summaries
- Use GPT to format invoice explanations
- Output a clean invoice via Streamlit UI

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + OpenAI**
- **Streamlit**
- **FPDF or WeasyPrint** (optional for PDF export)

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If you've been working on previous agents, continue. If not:

```
mkdir invoicing_agent
cd invoicing_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

To export invoices as PDFs:

```
pip install fpdf
```

✅ Step 2: Create mock invoice data (`invoice_data.py`)

```
import pandas as pd

def get_invoice_items():
    return pd.DataFrame({
        "Description": ["Consulting – Project A", "Design Work",
        "Training Session"],
        "Hours": [10, 5, 3],
        "Rate": [150, 120, 200]
    })
```

✅ Step 3: Format with GPT (`invoice_agent.py`)

```
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from invoice_data import get_invoice_items

invoice_prompt = PromptTemplate.from_template("""
You're an invoicing assistant AI. Based on the following billing data,
        generate:
1. A short summary of services rendered
2. A line-item breakdown
3. A polite closing message

Data:
{invoice_table}
""")

def generate_invoice_text():
    df = get_invoice_items()
    df["Total"] = df["Hours"] * df["Rate"]
    invoice_table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = invoice_prompt.format(invoice_table=invoice_table)
    output = llm.predict(prompt)

    total_amount = df["Total"].sum()
    return df, output, total_amount
```

✅ Step 4: Streamlit Interface (`app.py`)

```
import streamlit as st
from invoice_agent import generate_invoice_text

st.title("🧾 Automated Invoicing Agent")

if st.button("Generate Invoice"):
    df, summary, total = generate_invoice_text()

    st.subheader("📋 Invoice Items")
    st.dataframe(df)

    st.subheader("🧠 AI-Generated Summary")
    st.write(summary)

    st.subheader("💰 Total Amount Due")
    st.write(f"${total:.2f}")
```

✅ Optional: PDF Export (`pdf_generator.py`)

```
from fpdf import FPDF

def export_invoice_to_pdf(invoice_df, summary, total,
        filename="invoice.pdf"):
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Arial", size=12)

    pdf.cell(200, 10, txt="INVOICE", ln=True, align="C")

    pdf.ln(10)
    for i, row in invoice_df.iterrows():
        pdf.cell(200, 10, txt=f"{row['Description']}: {row['Hours']}
        hrs x ${row['Rate']} = ${row['Total']}", ln=True)

    pdf.ln(5)
    pdf.multi_cell(0, 10, txt=summary)

    pdf.ln(10)
    pdf.cell(200, 10, txt=f"Total Due: ${total}", ln=True)
    pdf.output(filename)
```

🧪 Example Output:

```
Summary:
The client was provided with 10 hours of consulting, 5 hours of design
work, and 3 hours of training. All services were delivered remotely
during the first week of July.

Line Items:
- Consulting: $1,500
- Design: $600
- Training: $600

Thank you for your business. Please remit payment within 14 days.
```

🎬 Agent #50: Video Content Creation Agent

📝 Overview

This agent generates short-form video scripts and storyboard outlines using GPT based on a selected topic, audience, and tone. Optionally, it can auto-generate voiceover scripts, scene descriptions, and caption text. In this lab, you'll create a video storyboard and script for platforms like YouTube Shorts, TikTok, or Instagram Reels.

🧪 Lab Objectives

By the end of this lab, you will:

- Select a video topic, audience, and tone
- Generate a 5–7 scene storyboard with GPT
- Create voiceover lines and captions per scene
- Visualize the structure in a Streamlit interface

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir video_content_agent
cd video_content_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain streamlit
```

✅ Step 2: Storyboard Prompt Template (`video_prompt.py`)

```
from langchain.prompts import PromptTemplate

video_prompt = PromptTemplate.from_template("""
You are a video content strategist.

Generate a storyboard for a short-form video with:
- Topic: {topic}
- Platform: {platform}
- Audience: {audience}
- Tone: {tone}

Create 5–7 scenes. For each scene include:
1. Scene description
2. Voiceover script
3. On-screen text (caption)
```

```
Keep it dynamic and under 60 seconds total.
""")
```

✅ Step 3: GPT Script Generator (`video_agent.py`)

```
from video_prompt import video_prompt
from langchain.chat_models import ChatOpenAI

def generate_storyboard(topic, platform, audience, tone):
    llm = ChatOpenAI(temperature=0.6)
    prompt = video_prompt.format(
        topic=topic,
        platform=platform,
        audience=audience,
        tone=tone
    )
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Interface (`app.py`)

```
import streamlit as st
from video_agent import generate_storyboard

st.title("🎬 Video Content Creation Agent")

topic = st.text_input("Video Topic", "AI for Small Business")
platform = st.selectbox("Platform", ["YouTube Shorts", "Instagram
        Reels", "TikTok"])
audience = st.text_input("Target Audience", "Startup founders")
tone = st.selectbox("Tone", ["Motivational", "Informative", "Funny",
        "Casual"])

if st.button("Generate Storyboard"):
    storyboard = generate_storyboard(topic, platform, audience, tone)
    st.subheader("📋 Storyboard Output")
    st.text_area("Generated Storyboard", storyboard, height=500)
    st.download_button("Download Script", storyboard,
        file_name="video_storyboard.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Title: How AI Saves Time for Founders ⏰ **Platform:** Instagram Reels **Tone:** Informative + Casual **Audience:** Startup founders

Scene 1: *Description:* Founder juggling emails and spreadsheets *Voiceover:* "Running a startup is non-stop chaos…" *Caption:* "Too many tasks, not enough time!"

Scene 2: *Description:* AI tool automates daily planning *Voiceover:* "But with AI, your morning starts with clarity." *Caption:* "AI daily planner in action 🧠 "

…

Scene 7: *Description:* Smiling founder sipping coffee *Voiceover:* "Now that's a productivity upgrade ☕ " *Caption:* "Founders, this is your new co-pilot!"

🛠️ Agent #51: Automated Ticket Resolution Agent

📝 Overview

This agent automatically resolves common IT support tickets using predefined knowledge base (KB) answers and LLM-based reasoning. It classifies ticket type, finds relevant KB entries or uses GPT to draft a response, and logs the solution. In this lab, you'll simulate ticket intake, use GPT for solution generation, and output a resolution log.

🧪 Lab Objectives

By the end of this lab, you will:

- Input or simulate incoming IT support tickets
- Classify the issue type (e.g., password reset, network error)
- Retrieve or generate automated responses using GPT
- Visualize the resolution and response log in a dashboard

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **(Optional)** SQLite for ticket logging

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ticket_resolution_agent
cd ticket_resolution_agent
python -m venv venv
```

```
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Sample Ticket Generator (`tickets.py`)

```
import pandas as pd
import random

def sample_tickets():
    return pd.DataFrame([
        {"Ticket ID": "TCK001", "Subject": "Can't connect to VPN",
        "Description": "VPN gives timeout when working from home."},
        {"Ticket ID": "TCK002", "Subject": "Forgot my password",
        "Description": "Need help resetting my company email
        password."},
        {"Ticket ID": "TCK003", "Subject": "Laptop running slow",
        "Description": "My device takes 10 minutes to boot."},
        {"Ticket ID": "TCK004", "Subject": "Software installation",
        "Description": "Need Adobe Acrobat Pro installed for team
        use."},
        {"Ticket ID": "TCK005", "Subject": "Printer offline",
        "Description": "Office printer won't connect, showing
        offline."}
    ])
```

✅ Step 3: GPT Prompt for Ticket Resolution (`ticket_prompt.py`)

```
from langchain.prompts import PromptTemplate

ticket_prompt = PromptTemplate.from_template("""
You are an IT support agent.

Resolve this ticket:
Subject: {subject}
Description: {description}

Steps:
1. Identify the issue type
2. Provide a clear resolution
3. Suggest any follow-up actions

Respond in professional IT support tone.
""")
```

✅ Step 4: GPT Resolution Engine (`resolution_agent.py`)

```
from ticket_prompt import ticket_prompt
from langchain.chat_models import ChatOpenAI
```

```
def resolve_ticket(subject, description):
    llm = ChatOpenAI(temperature=0.3)
    prompt = ticket_prompt.format(subject=subject,
        description=description)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
from tickets import sample_tickets
from resolution_agent import resolve_ticket

st.title("🛠️ Automated Ticket Resolution Agent")

df = sample_tickets()
ticket_list = df["Ticket ID"].tolist()
selected = st.selectbox("Select a Ticket", ticket_list)
ticket = df[df["Ticket ID"] == selected].iloc[0]

if st.button("Resolve Ticket"):
    resolution = resolve_ticket(ticket["Subject"],
        ticket["Description"])
    st.subheader("📋 Ticket Details")
    st.write(ticket["Description"])

    st.subheader("✅ Suggested Resolution")
    st.text_area("GPT Response", resolution, height=300)
    st.download_button("Download Resolution", resolution, file_name=f"
        {ticket['Ticket ID']}_resolution.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (for “VPN timeout”):

Issue Type: Connectivity Issue – VPN **Resolution:**

1. Instruct the user to restart their router and try reconnecting.
2. Ensure VPN client is updated to the latest version.
3. If error persists, escalate to Network Admin with logs. **Follow-Up:** Ask user to confirm when fixed.

🛡️ Agent #52: Security Threat Detection Agent

📝 Overview

This agent monitors system logs, API access patterns, or network traffic to detect suspicious behavior. It uses AI to flag anomalies like brute-force login attempts, abnormal API calls, or data exfiltration patterns. In this lab, you'll simulate log entries, apply simple detection rules, and use GPT to classify and summarize threats.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate security logs with various activities
- Apply basic anomaly detection logic
- Use GPT to label suspicious behaviors and suggest actions
- Generate an incident summary report

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir security_threat_agent
cd security_threat_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Simulated Security Logs (`logs.py`)

```
import pandas as pd
import random
from datetime import datetime, timedelta

def generate_logs():
    events = [
```

```
        "Login Success", "Login Failed", "File Access", "API Call",
        "Privilege Escalation", "Port Scan", "Database Query",
        "Multiple Login Failures", "Unusual File Download"
    ]
    ips = ["192.168.1.1", "10.0.0.2", "172.16.0.5", "192.168.1.45"]
    data = []

    for _ in range(50):
        timestamp = datetime.now() -
        timedelta(minutes=random.randint(0, 720))
        event = random.choice(events)
        ip = random.choice(ips)
        user = random.choice(["alice", "bob", "charlie", "admin"])
        data.append({
            "timestamp": timestamp,
            "user": user,
            "ip": ip,
            "event": event
        })

    return pd.DataFrame(data)
```

✅ Step 3: GPT Prompt for Threat Summary (`threat_prompt.py`)

```
from langchain.prompts import PromptTemplate

threat_prompt = PromptTemplate.from_template("""
You are a cybersecurity analyst.

Given the following suspicious events:
{events}

1. Identify the type of threat(s)
2. Suggest actions to mitigate or escalate
3. Summarize key users or IPs involved

Be concise and structured like an incident response memo.
""")
```

✅ Step 4: GPT Detection Engine (`threat_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from threat_prompt import threat_prompt

def generate_threat_summary(events: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = threat_prompt.format(events=events)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from logs import generate_logs
from threat_agent import generate_threat_summary

st.title("🛡️ Security Threat Detection Agent")

df = generate_logs()
st.subheader("🔍 Simulated Logs")
st.dataframe(df)

suspicious_events = df[df['event'].isin([
    "Login Failed", "Multiple Login Failures",
    "Privilege Escalation", "Port Scan",
    "Unusual File Download"
])]

if st.button("Analyze Threats"):
    event_str = suspicious_events.to_string(index=False)
    summary = generate_threat_summary(event_str)

    st.subheader("⚠️ GPT Threat Summary")
    st.text_area("Incident Report", summary, height=400)
    st.download_button("Download Report", summary,
        file_name="threat_report.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

Incident Report Threats Detected:

- Multiple login failures and port scanning suggest a brute-force attack
- Privilege escalation and unusual file downloads indicate a possible insider threat

Key Entities:

- User: `admin`, `bob`
- IPs: `192.168.1.45`, `10.0.0.2`

Suggested Actions:

- Temporarily disable affected accounts
- Block suspicious IPs via firewall
- Initiate forensic review of file access logs

🗂️ Agent #53: IT Ticket Prioritization Agent

📝 Overview

This agent reads incoming IT support tickets and automatically assigns a priority level (High, Medium, Low) based on urgency, impact, and keywords. It uses GPT to reason through the ticket content and outputs a structured priority log. In this lab, you'll simulate ticket intake, analyze content, and use GPT to assign priorities.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a set of incoming IT tickets
- Use GPT to classify priority (High / Medium / Low)
- Visualize and export a prioritized ticket queue
- Optionally assign escalation suggestions

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ticket_prioritization_agent
cd ticket_prioritization_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Sample Ticket Generator (`tickets.py`)

```
import pandas as pd

def get_tickets():
    return pd.DataFrame([
```

```
        {"Ticket ID": "IT001", "Subject": "Email not syncing",
        "Description": "My Outlook is not syncing emails since
        morning."},
        {"Ticket ID": "IT002", "Subject": "Laptop overheating",
        "Description": "Device heats up even when idle. Very hot to
        touch."},
        {"Ticket ID": "IT003", "Subject": "VPN not connecting",
        "Description": "Unable to connect to VPN from home."},
        {"Ticket ID": "IT004", "Subject": "Access request",
        "Description": "Need access to Jira and Confluence for my new
        project."},
        {"Ticket ID": "IT005", "Subject": "Data loss", "Description":
        "Files deleted from shared drive accidentally. Need recovery
        ASAP."}
    ])
```

✅ Step 3: GPT Prompt for Priority Assignment (`priority_prompt.py`)

```
from langchain.prompts import PromptTemplate

priority_prompt = PromptTemplate.from_template("""
You are an IT service desk manager.

Given the following ticket:
Subject: {subject}
Description: {description}

Assign a priority: High, Medium, or Low.

Rules:
- High = business-critical or data/security risks
- Medium = blocks daily work but has a workaround
- Low = access/setup requests or minor issues

Also provide a short explanation for the priority.
Format:
Priority: <priority>
Reason: <reason>
""")
```

✅ Step 4: GPT-Based Classifier (`priority_agent.py`)

```
from priority_prompt import priority_prompt
from langchain.chat_models import ChatOpenAI

def prioritize_ticket(subject, description):
    llm = ChatOpenAI(temperature=0.2)
    prompt = priority_prompt.format(subject=subject,
        description=description)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
from tickets import get_tickets
from priority_agent import prioritize_ticket

st.title("📂 IT Ticket Prioritization Agent")

tickets = get_tickets()
ticket_ids = tickets["Ticket ID"].tolist()
selected_id = st.selectbox("Select Ticket", ticket_ids)
selected_ticket = tickets[tickets["Ticket ID"] == selected_id].iloc[0]

if st.button("Assign Priority"):
    result = prioritize_ticket(
        selected_ticket["Subject"],
        selected_ticket["Description"]
    )
    st.subheader("📋 Ticket Info")
    st.write(selected_ticket)

    st.subheader("🎯 Priority Assignment")
    st.text_area("GPT Output", result, height=200)
    st.download_button("Download Result", result, file_name=f"
        {selected_id}_priority.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Subject: Data loss **Description:** Files deleted from shared drive accidentally. Need recovery ASAP.

GPT Output:

```
Priority: High
Reason: Involves data loss with potential business impact and urgency.
Requires immediate recovery action.
```

🖥️ Agent #54: System Monitoring Agent

📝 Overview

This agent continuously scans simulated system health metrics (CPU usage, memory, disk, uptime) and flags abnormal behavior using threshold-based rules or GPT-based diagnostics. It generates system health summaries and alerts for unusual patterns. In this lab, you'll simulate system metrics and use GPT to describe and assess the system state.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate system performance data (CPU, RAM, disk, etc.)
- Detect anomalies using rule-based and LLM-based checks
- Generate a system health summary using GPT
- Display the results in a Streamlit dashboard

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**
- *(Optional: psutil for real system data)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir system_monitoring_agent
cd system_monitoring_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Simulated System Data (`metrics.py`)

```
import pandas as pd
import random

def generate_metrics():
    return pd.DataFrame([
```

```
        {
            "Metric": "CPU Usage (%)", "Value":
        round(random.uniform(10, 95), 2)
        },
        {
            "Metric": "Memory Usage (%)", "Value":
        round(random.uniform(20, 95), 2)
        },
        {
            "Metric": "Disk Usage (%)", "Value":
        round(random.uniform(30, 98), 2)
        },
        {
            "Metric": "System Uptime (hours)", "Value":
        round(random.uniform(2, 240), 1)
        },
        {
            "Metric": "Network Activity (Mbps)", "Value":
        round(random.uniform(0.5, 150), 2)
        }
    ])
```

✅ Step 3: GPT Prompt for Health Summary (`monitor_prompt.py`)

```
from langchain.prompts import PromptTemplate

monitor_prompt = PromptTemplate.from_template("""
You are a system reliability engineer.

Here are recent system metrics:
{metrics}

1. Identify any abnormal readings
2. Summarize system health status
3. Recommend actions if necessary

Keep it short and professional.
""")
```

✅ Step 4: GPT Agent for System Health (`monitor_agent.py`)

```
from monitor_prompt import monitor_prompt
from langchain.chat_models import ChatOpenAI

def analyze_metrics(metrics_table: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = monitor_prompt.format(metrics=metrics_table)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from metrics import generate_metrics
from monitor_agent import analyze_metrics

st.title("🖥️ System Monitoring Agent")

df = generate_metrics()
st.subheader("📊 Simulated Metrics")
st.dataframe(df)

metrics_string = df.to_string(index=False)

if st.button("Analyze System Health"):
    report = analyze_metrics(metrics_string)
    st.subheader("🩺 System Health Summary")
    st.text_area("GPT Summary", report, height=300)
    st.download_button("Download Report", report,
        file_name="system_health_report.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

System Health Summary:

- CPU usage at 93% and memory usage at 89% indicate potential overload.
- Disk usage at 94% is nearing critical capacity.
- System uptime is 227 hours—recommend scheduled restart.
- Network activity is within normal range. **Action:** Restart non-critical processes, clear disk space, schedule maintenance.

👨‍💻 Agent #55: AI-driven Help Desk Agent

📝 Overview

This agent serves as a 24/7 virtual help desk assistant that responds to IT or HR-related questions, guides users through troubleshooting, and escalates unresolved issues. It uses natural language inputs and GPT-powered answers. In this lab, you'll build a chatbot-style help desk interface for common internal support questions.

🧪 Lab Objectives

By the end of this lab, you will:

- Create a chatbot interface for employee support
- Accept user queries and respond using GPT
- Route unresolved issues to human support escalation
- Maintain a chat history per session

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add Twilio, Zendesk, or Slack API later)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir help_desk_agent
cd help_desk_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template for Help Desk (`help_prompt.py`)

```
from langchain.prompts import PromptTemplate

help_prompt = PromptTemplate.from_template("""
You are a friendly and professional help desk assistant.

User query: "{query}"

1. Answer clearly and directly
2. If unsure or critical, recommend escalation
3. Keep responses under 120 words
""")
```

✅ Step 3: GPT Response Generator (`help_agent.py`)

```
from help_prompt import help_prompt
from langchain.chat_models import ChatOpenAI
```

```
def help_desk_reply(query):
    llm = ChatOpenAI(temperature=0.4)
    prompt = help_prompt.format(query=query)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Chat Interface (`app.py`)

```
import streamlit as st
from help_agent import help_desk_reply

st.title("🤖 AI-driven Help Desk Agent")
st.caption("Ask anything related to IT or HR support")

if "chat" not in st.session_state:
    st.session_state.chat = []

query = st.text_input("Enter your question here:")

if st.button("Ask"):
    if query:
        response = help_desk_reply(query)
        st.session_state.chat.append((query, response))

for i, (q, r) in enumerate(reversed(st.session_state.chat)):
    st.markdown(f"**You:** {q}")
    st.markdown(f"**Help Desk:** {r}")
    st.markdown("---")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Dialog

User: How do I reset my email password? **Help Desk:** You can reset your email password via the [Company Portal > Account Settings > Change Password]. If you're locked out, please submit a ticket to IT support for manual reset.

User: My laptop won't start even after charging. **Help Desk:** Please try holding the power button for 10 seconds. If the issue persists, escalate to IT support for hardware inspection.

🧾 Agent #56: IT Asset Management Agent

📝 Overview

This agent tracks IT assets like laptops, monitors, licenses, and peripherals, providing visibility into ownership, warranty status, location, and usage. It uses GPT for smart queries like "Who owns MacBook Pro #2031?" or "Show all licenses expiring this month." In this lab, you'll build a searchable IT asset dashboard with GPT-enhanced natural language queries.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate an asset inventory database
- Use GPT to interpret asset-related queries
- Display asset details, owners, status, and expiry alerts
- Generate a human-readable asset summary from data

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir it_asset_agent
cd it_asset_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Simulated Asset Inventory (`assets.py`)

```
import pandas as pd
from datetime import datetime, timedelta

def get_assets():
    today = datetime.today()
```

```
    return pd.DataFrame([
        {"Asset ID": "LTP-001", "Type": "Laptop", "Owner": "Alice",
        "Location": "NY Office", "Purchase Date": today -
        timedelta(days=700), "Warranty Expiry": today +
        timedelta(days=30)},
        {"Asset ID": "MON-104", "Type": "Monitor", "Owner": "Bob",
        "Location": "Remote", "Purchase Date": today -
        timedelta(days=400), "Warranty Expiry": today +
        timedelta(days=365)},
        {"Asset ID": "LIC-008", "Type": "Adobe License", "Owner":
        "Charlie", "Location": "NY Office", "Purchase Date": today -
        timedelta(days=300), "Warranty Expiry": today +
        timedelta(days=5)},
        {"Asset ID": "LTP-009", "Type": "Laptop", "Owner": "Denise",
        "Location": "SF Office", "Purchase Date": today -
        timedelta(days=900), "Warranty Expiry": today -
        timedelta(days=10)},
    ])
```

✅ Step 3: GPT Prompt for Asset Query (`asset_prompt.py`)

```
from langchain.prompts import PromptTemplate

asset_prompt = PromptTemplate.from_template("""
You are an IT asset assistant.

Given this asset data:
{asset_table}

User query: "{user_query}"

1. Identify the asset(s) involved
2. Summarize key information (owner, type, expiry)
3. Mention if warranty is expiring soon or expired

Respond clearly and concisely.
""")
```

✅ Step 4: GPT-Powered Query Handler (`asset_agent.py`)

```
from asset_prompt import asset_prompt
from langchain.chat_models import ChatOpenAI

def handle_asset_query(asset_table: str, user_query: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = asset_prompt.format(asset_table=asset_table,
        user_query=user_query)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit UI (`app.py`)

```
import streamlit as st
from assets import get_assets
from asset_agent import handle_asset_query

st.title("🧾 IT Asset Management Agent")

df = get_assets()
st.subheader("📋 Current Inventory")
st.dataframe(df)

user_query = st.text_input("Ask a question about your IT assets:",
        "Which licenses are expiring soon?")

if st.button("Ask"):
    result = handle_asset_query(df.to_string(index=False), user_query)
    st.subheader("🤖 GPT Response")
    st.text_area("Answer", result, height=300)
    st.download_button("Download Summary", result,
        file_name="asset_summary.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

User Query: Show all laptops with expired warranty. **GPT Response:**

- **LTP-009** (Laptop) is assigned to **Denise** at **SF Office**.
    - Warranty expired 10 days ago.
- Recommend notifying the user for replacement or extended coverage.

👨‍💻 Agent #57: AI Code Review Agent

📝 Overview

This agent reviews code snippets for bugs, security risks, code smells, and adherence to best practices. It uses GPT to analyze and suggest improvements in plain language. In this lab, you'll build a web-based AI-powered code review tool where developers can paste code and receive instant feedback.

🧪 Lab Objectives

By the end of this lab, you will:

- Paste code into a web interface
- Use GPT to detect issues and suggest improvements
- Display structured review output (Bugs, Security, Style)
- Enable download of review reports

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: GitHub API for repo analysis later)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_code_review_agent
cd ai_code_review_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template for Code Review (`review_prompt.py`)

```
from langchain.prompts import PromptTemplate

review_prompt = PromptTemplate.from_template("""
You are a senior software engineer performing a code review.

Please analyze the following code:

{code}


1. Identify any syntax errors or bugs
2. Suggest security improvements if needed
3. Recommend best practices or code style fixes

Provide output in this format:
- Issues
- Suggestions
```

```
- Overall Assessment
""")
```

✅ Step 3: GPT Review Engine (`review_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from review_prompt import review_prompt

def review_code(code: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = review_prompt.format(code=code)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit UI (`app.py`)

```python
import streamlit as st
from review_agent import review_code

st.title("🧑‍💻 AI Code Review Agent")

code_input = st.text_area("Paste your code here:", height=300)

if st.button("Run Review"):
    with st.spinner("Analyzing..."):
        output = review_code(code_input)
        st.subheader("📋 Review Output")
        st.text_area("AI Feedback", output, height=400)
        st.download_button("Download Report", output,
        file_name="code_review.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Input Code (Python):

```python
def fetch_data(url):
    import requests
    data = requests.get(url)
    return data.json()
```

GPT Review:

```
- Issues:
  • No error handling if `requests.get` fails
```

```
  • Import inside function is non-standard

- Suggestions:
  • Move `import requests` to top of file
  • Wrap `requests.get` in try-except block

- Overall Assessment:
  The function is functional but lacks robustness. Consider
modularizing and adding logging.
```

🚨 Agent #58: AI-driven Incident Response Agent

📝 Overview

This agent monitors for IT incidents, helps triage them based on severity and impact, and generates an action plan using GPT. It can summarize logs, recommend responses, and escalate issues if needed. In this lab, you'll simulate incident reports, assess risk levels, and generate GPT-powered incident response plans.

🧪 Lab Objectives

By the end of this lab, you will:

- Input or simulate incidents (e.g., downtime, security alerts)
- Use GPT to analyze severity, suggest actions, and escalate
- Generate a structured incident response memo
- Visualize the incident timeline and recommended actions

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate PagerDuty/Zendesk for escalation)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir incident_response_agent
cd incident_response_agent
python -m venv venv
```

```
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Simulate Incidents (`incidents.py`)

```
import pandas as pd
from datetime import datetime, timedelta
import random

def generate_incidents():
    now = datetime.now()
    incidents = [
        {
            "Incident ID": "INC-001",
            "Timestamp": now - timedelta(minutes=15),
            "Category": "Downtime",
            "Description": "Website unresponsive for 10 minutes"
        },
        {
            "Incident ID": "INC-002",
            "Timestamp": now - timedelta(hours=1),
            "Category": "Security",
            "Description": "Multiple failed login attempts on admin
        portal"
        },
        {
            "Incident ID": "INC-003",
            "Timestamp": now - timedelta(days=1),
            "Category": "Performance",
            "Description": "API latency increased beyond SLA
        threshold"
        },
        {
            "Incident ID": "INC-004",
            "Timestamp": now - timedelta(minutes=30),
            "Category": "Data Loss",
            "Description": "Lost customer data from form submission
        due to backend crash"
        }
    ]
    return pd.DataFrame(incidents)
```

✅ Step 3: GPT Prompt Template (`incident_prompt.py`)

```
from langchain.prompts import PromptTemplate

incident_prompt = PromptTemplate.from_template("""
You are an incident response manager.
```

```
Incident Details:
Category: {category}
Description: {description}

Tasks:
1. Determine severity (Low, Medium, High, Critical)
2. Recommend immediate actions
3. Draft a response memo (1–2 paragraphs)

Respond in a structured format.
""")
```

✅ Step 4: GPT-based Incident Responder (`incident_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from incident_prompt import incident_prompt

def respond_to_incident(category: str, description: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = incident_prompt.format(category=category,
        description=description)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from incidents import generate_incidents
from incident_agent import respond_to_incident

st.title("🚨 AI-driven Incident Response Agent")

df = generate_incidents()
st.subheader("📌 Reported Incidents")
st.dataframe(df)

incident_ids = df["Incident ID"].tolist()
selected = st.selectbox("Select an Incident", incident_ids)
incident = df[df["Incident ID"] == selected].iloc[0]

if st.button("Generate Response Plan"):
    output = respond_to_incident(
        incident["Category"], incident["Description"]
    )
    st.subheader("🧠 GPT Incident Response")
    st.text_area("Response Plan", output, height=400)
    st.download_button("Download Plan", output, file_name=f"
        {selected}_response.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Category: Security **Description:** Multiple failed login attempts on admin portal

GPT Response:

```
Severity: High
Immediate Action:
- Block suspicious IPs
- Reset admin passwords
- Notify security team

Memo:
At 10:32 AM, the system detected 17 failed login attempts to the admin
portal from a suspicious IP range. This may indicate a brute-force
attempt. IPs have been blocked and access logs escalated to the
security team. Further monitoring has been enabled for the next 24
hours.
```

🧠 Agent #59: Software Recommendation Agent

📝 Overview

This agent suggests the most suitable software tools based on user needs—e.g., project management, design, accounting, or CRM—by comparing feature sets, costs, integrations, and user personas. GPT interprets natural language queries and recommends tailored software options with reasons. In this lab, you'll build a GPT-powered assistant that takes in use cases and outputs software suggestions.

🧪 Lab Objectives

By the end of this lab, you will:

- Accept user queries describing their software needs
- Use GPT to recommend relevant tools with justification
- Display alternatives and decision factors
- Enable download of recommendation reports

🧰 Tech Stack

- **Python**
- **Streamlit**

- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir software_recommendation_agent
cd software_recommendation_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template for Recommendations (`recommend_prompt.py`)

```
from langchain.prompts import PromptTemplate

recommend_prompt = PromptTemplate.from_template("""
You are a software consultant.

User requirements:
"{user_needs}"

1. Recommend 2–3 software tools
2. Explain why each fits the need (features, cost, integrations)
3. Mention potential trade-offs or limitations

Format your response professionally and clearly.
""")
```

✅ Step 3: GPT Recommendation Engine (`recommend_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from recommend_prompt import recommend_prompt

def get_software_recommendation(user_needs: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = recommend_prompt.format(user_needs=user_needs)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit UI (`app.py`)

```
import streamlit as st
from recommend_agent import get_software_recommendation
```

```
st.title("🧠 Software Recommendation Agent")
st.caption("Describe what you're looking for, and get the right
        tools")

user_input = st.text_area(
    "Describe your use case:",
    "We need a project management tool for a remote team of 10 with
        task tracking, file sharing, and calendar integration."
)

if st.button("Get Recommendations"):
    result = get_software_recommendation(user_input)
    st.subheader("📋 Recommended Tools")
    st.text_area("GPT Suggestions", result, height=400)
    st.download_button("Download Report", result,
        file_name="software_recommendation.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

User Input: "We need an accounting tool for a small startup that integrates with Stripe and supports multi-currency billing."

GPT Response:

```
1. **Xero** – Cloud-based, integrates with Stripe, supports multi-
currency, and is ideal for startups. Easy to use and affordable.

2. **QuickBooks Online** – Offers robust accounting, multi-currency
support, and strong reporting. Slightly more expensive but better for
scalability.

3. **Zoho Books** – Good for automation, supports integrations, but
may lack some advanced features of QuickBooks.

Trade-offs: Xero has great usability but fewer customization options.
Zoho is cheaper but may not scale as well.
```

💳 Agent #6: Expense Management Agent

📝 Overview

This AI agent helps employees and finance teams manage expense reports by extracting data from receipts, categorizing spending, checking policy compliance, and preparing reimbursement summaries. In this lab, you'll simulate receipt entries, categorize

expenses, and use GPT to generate a compliance summary.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate a dataset of employee expenses
- Automatically categorize and validate expenses
- Use GPT to summarize policy compliance and suggest actions
- Display everything in an interactive Streamlit app

🧰 Tech Stack

- **Python**
- **Pandas**
- **OpenAI GPT-4 / GPT-3.5**
- **LangChain**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If continuing from earlier labs, skip this. Otherwise:

```
mkdir expense_management_agent
cd expense_management_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate mock expense data (`expense_data.py`)

```
import pandas as pd

def load_expenses():
    data = {
        "Date": ["2025-07-01", "2025-07-02", "2025-07-03", "2025-07-
        03", "2025-07-04"],
        "Description": ["Flight to NYC", "Hotel Stay", "Team Dinner",
        "Taxi", "Conference Fee"],
        "Category": ["Travel", "Lodging", "Meals", "Transport",
        "Training"],
        "Amount": [450, 600, 180, 45, 1200]
    }
```

```
    df = pd.DataFrame(data)
    return df
```

✅ Step 3: Define a policy compliance prompt (`expense_prompt.py`)

```
from langchain.prompts import PromptTemplate

expense_prompt = PromptTemplate.from_template("""
You are an AI expense compliance auditor.

Below is a list of expenses submitted by an employee:

{expense_table}

Tasks:
1. Check if any items exceed reasonable expense policy limits.
   - Travel > $500
   - Meals > $100
   - Training > $1000
2. Suggest which items should be reviewed or flagged.
3. Provide a summary for the finance team.
""")
```

✅ Step 4: Analyze using GPT (`expense_agent.py`)

```
import pandas as pd
from expense_data import load_expenses
from expense_prompt import expense_prompt
from langchain.chat_models import ChatOpenAI

def analyze_expenses():
    df = load_expenses()
    table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.2)
    prompt = expense_prompt.format(expense_table=table)
    output = llm.predict(prompt)
    return df, output
```

✅ Step 5: Build Streamlit app (`app.py`)

```
import streamlit as st
from expense_agent import analyze_expenses

st.title("💳 Expense Management Agent")
```

```
if st.button("Audit My Expenses"):
    df, result = analyze_expenses()

    st.subheader("📂 Expense Report")
    st.dataframe(df)

    st.subheader("🔍 GPT Expense Review")
    st.write(result)
```

Run it:

```
streamlit run app.py
```

🧪 Example Output:

```
The submitted expense report contains two items that exceed standard
policy limits:

- "Flight to NYC" at $450 is within limits.
- "Hotel Stay" at $600 is acceptable.
- "Team Dinner" at $180 exceeds the $100 meal limit and should be
reviewed.
- "Conference Fee" at $1200 exceeds the $1000 training cap.

Recommendations:
- Request itemized receipt for dinner.
- Validate conference details for policy exceptions.
```

☁️ Agent #60: Cloud Cost Optimization Agent

📝 Overview

This agent reviews simulated or real cloud usage data (AWS, Azure, GCP), identifies cost inefficiencies (idle instances, overprovisioned VMs, unused storage), and provides optimization strategies. Using GPT, it analyzes data and recommends actions like rightsizing, spot instance usage, or deletion. In this lab, you'll build a tool to upload cloud cost logs and generate GPT-based savings recommendations.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a cloud cost report (CSV) or use a simulated dataset
- Analyze high-cost resources
- Get GPT-powered cost optimization advice

- Visualize top spenders and savings opportunities

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas / Matplotlib**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir cloud_cost_agent
cd cloud_cost_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas matplotlib
```

✅ Step 2: Simulated Cloud Usage Data (`cloud_data.py`)

```
import pandas as pd

def get_cloud_costs():
    data = [
        {"Service": "EC2", "Instance": "t3.large", "Region": "us-east-
        1", "Monthly Cost": 120.00, "Usage Hours": 180, "Utilization
        %": 20},
        {"Service": "S3", "Instance": "N/A", "Region": "us-west-2",
        "Monthly Cost": 90.00, "Usage Hours": None, "Utilization %":
        None},
        {"Service": "RDS", "Instance": "db.m5.large", "Region": "us-
        east-1", "Monthly Cost": 200.00, "Usage Hours": 720,
        "Utilization %": 85},
        {"Service": "Lambda", "Instance": "N/A", "Region": "us-east-
        2", "Monthly Cost": 30.00, "Usage Hours": None, "Utilization
        %": None},
        {"Service": "EC2", "Instance": "m5.xlarge", "Region": "us-
        west-1", "Monthly Cost": 300.00, "Usage Hours": 100,
        "Utilization %": 10}
    ]
    return pd.DataFrame(data)
```

✅ Step 3: Prompt Template for Optimization (`cloud_prompt.py`)

```
from langchain.prompts import PromptTemplate

cloud_prompt = PromptTemplate.from_template("""
You are a cloud cost optimization expert.

Here is recent cloud usage data:
{cloud_table}

Analyze:
1. Identify high-cost, low-usage resources
2. Recommend optimization actions (e.g. rightsizing, deletion, spot
        instances)
3. Estimate potential monthly savings

Keep it concise and action-oriented.
""")
```

✅ Step 4: GPT Advisor Logic (`cloud_agent.py`)

```
from cloud_prompt import cloud_prompt
from langchain.chat_models import ChatOpenAI

def analyze_cloud_costs(table_str: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = cloud_prompt.format(cloud_table=table_str)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Dashboard (`app.py`)

```
import streamlit as st
from cloud_data import get_cloud_costs
from cloud_agent import analyze_cloud_costs
import matplotlib.pyplot as plt

st.title("☁️ Cloud Cost Optimization Agent")

df = get_cloud_costs()
st.subheader("💸 Cloud Spend Overview")
st.dataframe(df)

# Bar Chart
st.subheader("🔍 Top Services by Cost")
fig, ax = plt.subplots()
df.groupby("Service")["Monthly Cost"].sum().plot(kind="bar", ax=ax)
st.pyplot(fig)
```

```
if st.button("Generate Optimization Plan"):
    result = analyze_cloud_costs(df.to_string(index=False))
    st.subheader("🧠 GPT Optimization Recommendations")
    st.text_area("Response", result, height=400)
    st.download_button("Download Report", result,
        file_name="cloud_optimization.txt")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

GPT Optimization Summary:

```
- EC2 instance m5.xlarge is severely underutilized (10%) — consider
rightsizing or using spot instances.
- t3.large also shows low usage (20%) — evaluate if it can be paused
or consolidated.
- S3 costs are high — check for redundant or untagged buckets.
Estimated savings: $200–300/month with changes.
```

💬 Agent #61: AI Chatbot for Customer Support

📝 Overview

This agent acts as a 24/7 customer support representative, answering FAQs, resolving common issues, and escalating complex queries. Powered by GPT, it uses context-aware responses, maintains session memory, and handles polite escalation. In this lab, you'll build an interactive GPT-based chatbot for simulated customer support interactions.

🧪 Lab Objectives

By the end of this lab, you will:

- Create a live chatbot interface using Streamlit
- Accept customer queries and return GPT responses
- Maintain basic conversation memory
- Simulate escalation for complex or unresolved issues

🧰 Tech Stack

- **Python**
- **Streamlit**

- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add RAG or vector memory later)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_customer_chatbot
cd ai_customer_chatbot
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template (`chat_prompt.py`)

```
from langchain.prompts import PromptTemplate

chat_prompt = PromptTemplate.from_template("""
You are an AI customer service chatbot.

Respond to the customer message below:
"{user_message}"

Your goals:
1. Be polite and professional
2. Solve the issue if possible
3. If the query is too complex, say: "Let me connect you with a human
        support agent."

Limit your reply to 100 words.
""")
```

✅ Step 3: GPT Chat Logic (`chat_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from chat_prompt import chat_prompt

def customer_reply(user_message):
    llm = ChatOpenAI(temperature=0.4)
    prompt = chat_prompt.format(user_message=user_message)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Chatbot UI (`app.py`)

```
import streamlit as st
from chat_agent import customer_reply

st.title("💬 AI Chatbot for Customer Support")
st.caption("Ask a product or support-related question")

if "history" not in st.session_state:
    st.session_state.history = []

user_input = st.text_input("Your message:")

if st.button("Send"):
    if user_input:
        response = customer_reply(user_input)
        st.session_state.history.append(("Customer", user_input))
        st.session_state.history.append(("AI Support", response))

for speaker, message in reversed(st.session_state.history[-10:]):
    st.markdown(f"**{speaker}:** {message}")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Dialog

Customer: I didn't receive my order placed last Thursday. **AI Chatbot:** I'm sorry to hear that! Please check your tracking number in your confirmation email. If you still can't locate it, I'll connect you with a human support agent.

Customer: How do I return an item? **AI Chatbot:** You can return your item within 30 days of delivery. Visit your order history, select the item, and click "Return Item" to generate a return label.

😊 Agent #62: Sentiment Analysis Agent

📝 Overview

This agent analyzes customer feedback, support conversations, or reviews to determine sentiment—positive, negative, or neutral. It summarizes emotions and provides actionable insights for product or service teams. In this lab, you'll build a GPT-powered sentiment analysis tool with text input and batch CSV upload functionality.

🧪 Lab Objectives

By the end of this lab, you will:

- Accept single or batch text inputs
- Classify sentiment using GPT (positive, neutral, negative)
- Provide short explanations and suggestions
- Visualize sentiment distribution

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas / Matplotlib**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir sentiment_analysis_agent
cd sentiment_analysis_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas matplotlib
```

✅ Step 2: Prompt Template (`sentiment_prompt.py`)

```
from langchain.prompts import PromptTemplate

sentiment_prompt = PromptTemplate.from_template("""
You are a customer feedback analyst.

Feedback: "{feedback}"

Tasks:
1. Classify sentiment as: Positive, Neutral, or Negative
2. Provide a 1-sentence explanation
3. Recommend an action if appropriate

Respond clearly and concisely.
""")
```

✅ Step 3: GPT Sentiment Engine (`sentiment_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from sentiment_prompt import sentiment_prompt

def analyze_sentiment(feedback: str):
    llm = ChatOpenAI(temperature=0.2)
    prompt = sentiment_prompt.format(feedback=feedback)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit App with Upload & Chart (`app.py`)

```python
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from sentiment_agent import analyze_sentiment

st.title("😊 Sentiment Analysis Agent")
st.caption("Understand customer feedback instantly")

# Single Input
feedback = st.text_area("Enter a feedback message:")

if st.button("Analyze Sentiment"):
    result = analyze_sentiment(feedback)
    st.subheader("🧠 GPT Analysis")
    st.text_area("Result", result, height=200)

# Batch Upload
st.divider()
st.subheader("📁 Upload CSV for Batch Sentiment")

file = st.file_uploader("Upload CSV with a 'Feedback' column", type=
        ["csv"])
if file:
    df = pd.read_csv(file)
    sentiments = []
    for fb in df["Feedback"]:
        try:
            response = analyze_sentiment(fb)
            sentiments.append(response.split("\n")[0].split(":")
        [-1].strip())  # crude extraction
        except:
            sentiments.append("Error")

    df["Sentiment"] = sentiments
    st.dataframe(df)

    st.download_button("Download Results", df.to_csv(index=False),
        "sentiment_results.csv")
```

```
    # Plot
    st.subheader("📊 Sentiment Distribution")
    fig, ax = plt.subplots()
    df["Sentiment"].value_counts().plot(kind="bar", ax=ax, color=
        ["green", "gray", "red"])
    st.pyplot(fig)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Feedback: “I had to wait on hold for 45 minutes just to talk to someone!” **GPT Response:**

- **Sentiment:** Negative
- **Explanation:** The customer expresses frustration about long wait times.
- **Recommended Action:** Investigate call center staffing or offer callbacks.

📞 Agent #63: Automated Call Center Agent

📝 Overview

This agent simulates a voice-based call center representative capable of handling customer queries, booking appointments, answering FAQs, or routing calls using voice inputs and GPT-generated responses. In this lab, you’ll build a prototype that mimics an automated call flow using text input, but can be expanded to voice integration with tools like Twilio or SpeechRecognition.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate an automated call experience using a conversation flow
- Use GPT to respond to user queries in plain language
- Handle call routing or escalation instructions
- (Optionally) Prepare the base for voice-to-text integration

---

🧰 Tech Stack

- **Python**
- **Streamlit**

- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Twilio, Google Speech-to-Text, Whisper API)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir automated_call_center_agent
cd automated_call_center_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template (`call_prompt.py`)

```
from langchain.prompts import PromptTemplate

call_prompt = PromptTemplate.from_template("""
You are an AI-powered automated phone support agent.

Customer said: "{user_input}"

Respond with:
1. A polite greeting or next step
2. Simple, clear instructions
3. Offer to connect to a human agent if the request is too complex

Limit to 100 words and simulate a voice assistant tone.
""")
```

✅ Step 3: GPT Agent Logic (`call_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from call_prompt import call_prompt

def handle_call_input(user_input: str):
    llm = ChatOpenAI(temperature=0.4)
    prompt = call_prompt.format(user_input=user_input)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Call Simulation Interface (`app.py`)

```
import streamlit as st
from call_agent import handle_call_input
```

```
st.title("📞 Automated Call Center Agent")
st.caption("Simulate a phone-based AI conversation")

if "call_log" not in st.session_state:
    st.session_state.call_log = []

user_input = st.text_input("Customer says:")

if st.button("Respond"):
    response = handle_call_input(user_input)
    st.session_state.call_log.append(("Customer", user_input))
    st.session_state.call_log.append(("AI Agent", response))

for speaker, message in reversed(st.session_state.call_log[-10:]):
    st.markdown(f"**{speaker}:** {message}")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Call Simulation

Customer: I want to know my account balance. **AI Agent:** Sure! Please say or type your account number. If you'd prefer to speak with a support representative, I can connect you now.

Customer: I need to cancel my flight reservation. **AI Agent:** I can help with that. May I have your booking ID? If you need assistance beyond cancellation, I'll route you to a human agent.

🎙️ Agent #64: Voice Analytics Agent

📝 Overview

This agent analyzes audio from customer support calls or meetings to extract key insights like speaker sentiment, call intent, interruptions, and action items. It transcribes the call, performs sentiment tagging, and summarizes outcomes using GPT. In this lab, we'll use pre-recorded audio, convert it to text (via Whisper or SpeechRecognition), and feed it to GPT for analysis.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a customer support call (MP3/WAV)

- Transcribe the audio into text
- Use GPT to extract call insights and sentiment
- View a structured summary and recommended actions

🧰 Tech Stack

- **Python**
- **Streamlit**
- **OpenAI Whisper or `whisper` Python package**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir voice_analytics_agent
cd voice_analytics_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain whisper
```

✅ Step 2: Download Whisper Model (Optional)

```
# Optional: If using whisper locally
import whisper
model = whisper.load_model("base")
```

✅ Step 3: Transcription Logic (`transcribe.py`)

```
import whisper

def transcribe_audio(file_path):
    model = whisper.load_model("base")
    result = model.transcribe(file_path)
    return result["text"]
```

✅ Step 4: GPT Call Summary Prompt (`call_summary_prompt.py`)

```
from langchain.prompts import PromptTemplate

call_summary_prompt = PromptTemplate.from_template("""
You are an AI voice analytics agent.
```

```
Here is the transcript of a customer support call:
"{transcript}"

Extract and respond with:
1. Call intent
2. Customer sentiment (Positive, Neutral, Negative)
3. Key concerns raised
4. Recommended follow-up action

Be professional and concise.
""")
```

✅ Step 5: GPT Analysis (`voice_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from call_summary_prompt import call_summary_prompt

def analyze_transcript(transcript):
    llm = ChatOpenAI(temperature=0.3)
    prompt = call_summary_prompt.format(transcript=transcript)
    return llm.predict(prompt)
```

✅ Step 6: Streamlit App (`app.py`)

```
import streamlit as st
from transcribe import transcribe_audio
from voice_agent import analyze_transcript
import tempfile

st.title("🎙️ Voice Analytics Agent")
st.caption("Analyze calls for sentiment, intent, and action items")

audio_file = st.file_uploader("Upload an MP3 or WAV file", type=
        ["mp3", "wav"])

if audio_file:
    with tempfile.NamedTemporaryFile(delete=False, suffix=".wav") as
        tmp:
        tmp.write(audio_file.read())
        tmp_path = tmp.name

    with st.spinner("Transcribing..."):
        transcript = transcribe_audio(tmp_path)

    st.subheader("📝 Transcript")
    st.text_area("Call Transcript", transcript, height=200)
```

```
    if st.button("Analyze Call"):
        with st.spinner("Analyzing..."):
            result = analyze_transcript(transcript)
        st.subheader("🧠 Call Insights")
        st.text_area("GPT Analysis", result, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Transcript Snippet: *"I've been trying to resolve a billing issue for weeks. No one's responding to my emails!"*

GPT Response:

```
- Intent: Resolve a billing issue
- Sentiment: Negative
- Key Concerns: Unresponsive support, unresolved charges
- Recommendation: Escalate to billing supervisor, respond within 24 hours
```

💎 Agent #65: Customer Loyalty Program Agent

📝 Overview

This agent designs, recommends, or personalizes customer loyalty programs using behavioral data, past purchases, and sentiment. It uses GPT to suggest rewards, tiering, and retention strategies tailored to each customer segment. In this lab, you'll build a loyalty program assistant that analyzes customer profiles and outputs loyalty strategy suggestions.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload or input customer data (CSV or manual)
- Generate loyalty strategies using GPT
- Output reward suggestions and engagement tactics
- Download a personalized loyalty plan per customer

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir loyalty_program_agent
cd loyalty_program_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Sample Customer Dataset (`sample_data.csv`)

```
CustomerID,Name,TotalSpend,LoyaltyTier,LastPurchaseDays,FeedbackSentime
C001,Alice,2400,Gold,10,Positive
C002,Bob,600,Silver,45,Neutral
C003,Charlie,300,Bronze,80,Negative
```

✅ Step 3: GPT Prompt Template (`loyalty_prompt.py`)

```
from langchain.prompts import PromptTemplate

loyalty_prompt = PromptTemplate.from_template("""
You are an AI loyalty program designer.

Here is a customer profile:
Name: {Name}
Total Spend: {TotalSpend}
Current Tier: {LoyaltyTier}
Last Purchase: {LastPurchaseDays} days ago
Sentiment: {FeedbackSentiment}

Recommend:
1. New loyalty tier (if applicable)
2. Personalized reward or promotion
3. Retention strategy or message

Keep it concise and brand-friendly.
""")
```

✅ Step 4: GPT Analysis Logic (`loyalty_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from loyalty_prompt import loyalty_prompt

def generate_loyalty_strategy(customer_dict):
    llm = ChatOpenAI(temperature=0.3)
    prompt = loyalty_prompt.format(**customer_dict)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
import pandas as pd
from loyalty_agent import generate_loyalty_strategy

st.title("💎 Customer Loyalty Program Agent")
st.caption("Create personalized loyalty plans")

file = st.file_uploader("Upload CSV (with headers: Name, TotalSpend,
        LoyaltyTier, LastPurchaseDays, FeedbackSentiment)", type=
        ["csv"])

if file:
    df = pd.read_csv(file)
    results = []

    for _, row in df.iterrows():
        strategy = generate_loyalty_strategy(row.to_dict())
        results.append(strategy)

    df["GPT_Loyalty_Strategy"] = results
    st.dataframe(df)
    st.download_button("Download Plan", df.to_csv(index=False),
        file_name="loyalty_program_output.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (For Alice)

```
- Loyalty Tier: Remain Gold
- Reward: 15% off coupon for next order + early access to new products
- Message: "Thanks for being a top customer! Enjoy exclusive early
access this month."
```

🛠️ Agent #66: AI-driven Complaint Resolution Agent

📝 Overview

This agent reads customer complaints—via form, chat, or email—and generates structured responses, prioritizes cases, and recommends resolutions. It uses GPT to draft empathetic, professional replies and suggest next steps. In this lab, you'll build a complaint resolution assistant that ingests complaint text and outputs an apology, a proposed resolution, and whether escalation is needed.

🧪 Lab Objectives

By the end of this lab, you will:

- Accept free-text customer complaints
- Use GPT to extract issue type and urgency
- Generate a resolution email or message
- Recommend whether to escalate or resolve immediately

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir complaint_resolution_agent
cd complaint_resolution_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Template (`complaint_prompt.py`)

```
from langchain.prompts import PromptTemplate

complaint_prompt = PromptTemplate.from_template("""
You are an AI customer service resolution agent.

Here is a customer complaint:
```

```
"{complaint}"

Tasks:
1. Summarize the core issue in one line
2. Classify urgency as High, Medium, or Low
3. Generate a professional response email
4. Indicate whether escalation is needed (Yes/No)

Keep the tone empathetic and solution-oriented.
""")
```

✅ Step 3: GPT Resolution Engine (`resolution_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from complaint_prompt import complaint_prompt

def resolve_complaint(complaint: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = complaint_prompt.format(complaint=complaint)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Interface (`app.py`)

```python
import streamlit as st
from resolution_agent import resolve_complaint

st.title("🛠️ AI-driven Complaint Resolution Agent")
st.caption("Draft polite and prompt replies to customer complaints")

complaint = st.text_area("Paste customer complaint:")

if st.button("Resolve"):
    with st.spinner("Analyzing..."):
        result = resolve_complaint(complaint)
    st.subheader("🧠 GPT Response")
    st.text_area("Resolution Plan", result, height=350)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Complaint Input: *"I ordered a laptop 2 weeks ago and it still hasn't arrived. The tracking number doesn't work and your support team hasn't responded to my emails."*

GPT Output:

```
1. Core Issue: Delayed shipment and lack of support response
2. Urgency: High
3. Response Email:
Dear [Customer Name],
We sincerely apologize for the delay in your laptop delivery and the
inconvenience caused by our lack of response. We're investigating the
shipping issue and will update you within 24 hours. Please accept a
$25 credit as a goodwill gesture.
Thank you for your patience.
Best regards,
Customer Support Team
4. Escalation: Yes
```

🤖 Agent #67: AI-powered Virtual Assistant

📝 Overview

This agent acts as a general-purpose virtual assistant for employees or customers. It can schedule meetings, retrieve knowledge base articles, summarize emails, set reminders, or answer general inquiries using GPT. In this lab, you'll build an interactive assistant that handles natural language queries with contextual responses, simulating productivity-focused support.

🧪 Lab Objectives

By the end of this lab, you will:

- Create an AI assistant interface
- Accept user queries and return context-aware answers
- Simulate actions like reminders or knowledge lookup
- Extend the assistant with plugins or tools

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate with APIs like Google Calendar, Notion, etc.)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir virtual_assistant_agent
cd virtual_assistant_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Assistant Prompt Template (`assistant_prompt.py`)

```
from langchain.prompts import PromptTemplate

assistant_prompt = PromptTemplate.from_template("""
You are a helpful AI-powered virtual assistant.

User message:
"{user_input}"

Your job is to:
1. Understand the request (e.g., schedule meeting, find document,
        summarize email)
2. Respond clearly with the next step or information
3. Ask follow-up if needed

Keep responses polite, actionable, and under 120 words.
""")
```

✅ Step 3: Assistant Engine Logic (`assistant_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from assistant_prompt import assistant_prompt

def respond_to_user(user_input: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = assistant_prompt.format(user_input=user_input)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Assistant App (`app.py`)

```
import streamlit as st
from assistant_agent import respond_to_user

st.title("🤖 AI-powered Virtual Assistant")
```

```
st.caption("Ask me anything to help with your work or planning!")

if "assistant_log" not in st.session_state:
    st.session_state.assistant_log = []

query = st.text_input("Your command or question:")

if st.button("Ask"):
    if query:
        response = respond_to_user(query)
        st.session_state.assistant_log.append(("You", query))
        st.session_state.assistant_log.append(("Assistant", response))

for speaker, message in
        reversed(st.session_state.assistant_log[-10:]):
    st.markdown(f"**{speaker}:** {message}")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Interactions

User: Schedule a meeting with the product team next Thursday at 3 PM. **Assistant:** I've noted the meeting for Thursday at 3 PM. Do you want me to send calendar invites or check availability?

User: Summarize this email: "Hi team, the Q2 report is due Friday. Please send your inputs by Wednesday." **Assistant:** The email requests Q2 report inputs by Wednesday, ahead of the Friday deadline.

User: Set a reminder to check campaign results tomorrow morning. **Assistant:** Got it! I'll remind you tomorrow morning to check the campaign results.

🗣️ Agent #68: AI-powered Feedback Management Agent

📝 Overview

This agent collects, categorizes, and summarizes customer or employee feedback from various channels (surveys, reviews, emails, chats). It uses GPT to analyze sentiment, tag themes (e.g., "delivery delay", "product quality"), and recommend actions. In this lab, you'll build a feedback analyzer that accepts CSV feedback data and returns structured insights.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a feedback dataset (CSV)
- Use GPT to tag themes and sentiment
- Summarize feedback across customers
- Generate a report with insights and actionables

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **Pandas**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir feedback_management_agent
cd feedback_management_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Sample CSV (`feedback.csv`)

```
CustomerID,Feedback
C001,"The delivery was 3 days late and the support didn't help."
C002,"Absolutely love the product quality and the speed of shipping!"
C003,"I found the onboarding process confusing and slow."
```

✅ Step 3: GPT Prompt Template (`feedback_prompt.py`)

```
from langchain.prompts import PromptTemplate

feedback_prompt = PromptTemplate.from_template("""
You are an AI feedback analyst.

Feedback: "{feedback}"

Respond with:
1. Sentiment (Positive, Neutral, Negative)
```

```
2. Main Theme (e.g., shipping, product, service, UX)
3. Actionable Suggestion (1 sentence)

Output in structured bullet points.
""")
```

✅ Step 4: Feedback Analyzer (`feedback_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from feedback_prompt import feedback_prompt

def analyze_feedback(feedback: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = feedback_prompt.format(feedback=feedback)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```python
import streamlit as st
import pandas as pd
from feedback_agent import analyze_feedback

st.title("🗣️ AI-powered Feedback Management Agent")
st.caption("Analyze, tag, and summarize feedback at scale")

uploaded_file = st.file_uploader("Upload CSV with 'Feedback' column",
        type=["csv"])

if uploaded_file:
    df = pd.read_csv(uploaded_file)
    responses = []
    for fb in df["Feedback"]:
        try:
            result = analyze_feedback(fb)
            responses.append(result)
        except:
            responses.append("Error")

    df["GPT_Analysis"] = responses
    st.dataframe(df)
    st.download_button("Download Analysis", df.to_csv(index=False),
        "feedback_analysis.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (from CSV)

Feedback: "The delivery was 3 days late and the support didn't help." **GPT Output:**

- Sentiment: Negative
- Theme: Shipping & Customer Support
- Suggestion: Improve response times and offer proactive delivery updates.

⚡ Agent #69: Energy Efficiency Optimization Agent

📝 Overview

This agent analyzes energy consumption data from buildings, factories, or devices to recommend cost-saving and sustainability improvements. Using time-series data, it identifies inefficiencies, unusual spikes, and suggests optimization strategies. In this lab, you'll build an AI agent that processes energy usage data and uses GPT to provide actionable recommendations.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload energy consumption data (CSV)
- Use pandas to compute consumption trends
- Feed results to GPT for insights
- Output optimization suggestions (e.g., peak-hour reduction, equipment audit)

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Matplotlib**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir energy_efficiency_agent
cd energy_efficiency_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas matplotlib
```

✅ Step 2: Sample CSV (`energy_data.csv`)

```
Timestamp,Device,Energy_kWh
2025-07-01 00:00:00,HVAC,12.5
2025-07-01 01:00:00,HVAC,14.2
2025-07-01 00:00:00,Lighting,4.3
2025-07-01 01:00:00,Lighting,4.1
...
```

✅ Step 3: Prompt Template (`efficiency_prompt.py`)

```python
from langchain.prompts import PromptTemplate

efficiency_prompt = PromptTemplate.from_template("""
You are an energy efficiency consultant AI.

Here are recent consumption patterns (in kWh):
{summary}

Identify:
1. Devices or time periods with inefficiency
2. Energy-saving recommendations
3. Any anomalies to investigate

Keep your response under 150 words.
""")
```

✅ Step 4: GPT Agent Logic (`efficiency_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from efficiency_prompt import efficiency_prompt

def analyze_energy_efficiency(summary: str):
    llm = ChatOpenAI(temperature=0.2)
    prompt = efficiency_prompt.format(summary=summary)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```python
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from efficiency_agent import analyze_energy_efficiency

st.title("⚡ Energy Efficiency Optimization Agent")
```

```
st.caption("Analyze energy data and recommend optimizations")

file = st.file_uploader("Upload CSV with Timestamp, Device,
        Energy_kWh", type=["csv"])

if file:
    df = pd.read_csv(file, parse_dates=["Timestamp"])

    st.subheader("📈 Energy Trend")
    pivot = df.pivot_table(index="Timestamp", columns="Device",
        values="Energy_kWh", aggfunc="sum")
    st.line_chart(pivot)

    summary = df.groupby("Device")["Energy_kWh"].sum().to_string()
    st.subheader("🧠 GPT Optimization Suggestions")
    response = analyze_energy_efficiency(summary)
    st.text_area("Suggestions", response, height=250)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Data Summary:

```
Device
HVAC        340.8
Lighting    112.4
ServerRack  389.2
```

GPT Output:

```
- The ServerRack has consistently high energy use. Consider power-
saving modes or server load balancing.
- HVAC peaks during midday—review insulation and thermostat settings.
- Lighting is efficient, but sensors or timers could reduce usage
further.
- No severe anomalies detected, but recommend a power audit of server
systems.
```

📉 Agent #7: Financial Risk Assessment Agent

📝 Overview

This AI agent evaluates the financial risk of a decision, transaction, or entity by analyzing internal data (e.g., credit exposure, volatility) and external factors (e.g., market trends, ratings). In this lab, you'll simulate company financial data and use GPT to analyze and

classify risk levels with a rationale.

🧪 Lab Objectives

By the end of this lab, you will:

- Create a simulated financial profile (company or investment)
- Use GPT to assess risk levels (low/medium/high)
- Generate rationale and recommendations
- Display results interactively in Streamlit

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If continuing from earlier labs, skip this. Otherwise:

```
mkdir financial_risk_agent
cd financial_risk_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate financial risk profile data (`risk_data.py`)

```
import pandas as pd

def get_financial_profile():
    data = {
        "Metric": [
            "Debt-to-Equity Ratio",
            "Liquidity Ratio",
            "Revenue Growth (YoY)",
            "Net Margin",
            "Credit Score",
            "Market Volatility (1–10)"
        ],
```

```
        "Value": [2.8, 0.95, -5.0, 3.2, 620, 8]
    }
    return pd.DataFrame(data)
```

✅ Step 3: Create GPT prompt template (`risk_prompt.py`)

```
from langchain.prompts import PromptTemplate

risk_template = PromptTemplate.from_template("""
You are a financial risk analyst AI.

Given the following financial metrics:

{metrics_table}

Please:
1. Assess the financial risk level (low, medium, high).
2. Justify your assessment.
3. Recommend actions to reduce risk if needed.
Keep it concise but insightful.
""")
```

✅ Step 4: Analyze with GPT (`risk_agent.py`)

```
from risk_data import get_financial_profile
from risk_prompt import risk_template
from langchain.chat_models import ChatOpenAI

def analyze_risk():
    df = get_financial_profile()
    table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = risk_template.format(metrics_table=table)
    result = llm.predict(prompt)
    return df, result
```

✅ Step 5: Build the Streamlit interface (`app.py`)

```
import streamlit as st
from risk_agent import analyze_risk

st.title("📉 Financial Risk Assessment Agent")

if st.button("Run Risk Analysis"):
    df, result = analyze_risk()
```

```
    st.subheader("📊 Financial Profile")
    st.dataframe(df)

    st.subheader("🔎 AI Risk Assessment")
    st.write(result)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Risk Level: HIGH

Justification:
- The debt-to-equity ratio (2.8) indicates heavy reliance on
borrowing.
- Liquidity ratio below 1.0 suggests cash flow strain.
- Negative revenue growth reflects operational concerns.
- A credit score of 620 is below investment grade.
- High market volatility further compounds risk.

Recommendations:
- Improve liquidity via cost-cutting or refinancing.
- Stabilize revenue streams.
- Engage in credit repair strategies.
```

📄 Agent #70: Contract Renewal & Expiry Agent

📝 Overview

This agent monitors contract metadata (start/end dates, renewal terms, client name) and notifies users of upcoming expirations or auto-renewals. It can also suggest action steps based on client value, renewal terms, or performance. In this lab, you'll build a system that reads contract data, flags contracts due soon, and uses GPT to recommend next steps.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a contract tracking file (CSV)
- Identify contracts nearing renewal or expiry
- Use GPT to suggest renewal actions
- Output a renewal summary for each client

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + datetime**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir contract_renewal_agent
cd contract_renewal_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pandas
```

✅ Step 2: Sample CSV (`contracts.csv`)

```
ClientName,ContractStart,ContractEnd,AutoRenew,ClientValue,Notes
Acme Inc,2023-09-01,2024-09-01,Yes,High,"Key enterprise client"
Beta Corp,2023-07-15,2024-07-30,No,Medium,"Late on last payment"
Delta LLC,2023-10-01,2024-10-01,Yes,Low,"Under pilot program"
```

✅ Step 3: GPT Prompt Template (`contract_prompt.py`)

```
from langchain.prompts import PromptTemplate

contract_prompt = PromptTemplate.from_template("""
You are a contract renewal advisor AI.

Contract Info:
- Client: {ClientName}
- Ends: {ContractEnd}
- Auto-renew: {AutoRenew}
- Value: {ClientValue}
- Notes: {Notes}

Based on this, recommend:
1. Renewal Action (Renew, Negotiate, Terminate)
2. Reasoning (1-2 sentences)
3. Any follow-up needed
```

```
Respond in clear bullet points.
""")
```

✅ Step 4: GPT Logic (`renewal_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from contract_prompt import contract_prompt

def get_renewal_advice(contract_dict):
    llm = ChatOpenAI(temperature=0.2)
    prompt = contract_prompt.format(**contract_dict)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from datetime import datetime, timedelta
from renewal_agent import get_renewal_advice

st.title("📄 Contract Renewal & Expiry Agent")
st.caption("Track and act on upcoming contract renewals")

file = st.file_uploader("Upload CSV with contract metadata", type=
        ["csv"])

if file:
    df = pd.read_csv(file, parse_dates=["ContractEnd"])
    today = datetime.today()
    df["DaysToEnd"] = (df["ContractEnd"] - today).dt.days
    due_df = df[df["DaysToEnd"] <= 60]  # Flag contracts expiring in
        60 days

    results = []
    for _, row in due_df.iterrows():
        result = get_renewal_advice(row.to_dict())
        results.append(result)

    due_df["GPT_Advice"] = results
    st.dataframe(due_df[["ClientName", "ContractEnd", "DaysToEnd",
        "GPT_Advice"]])
    st.download_button("Download Renewal Actions",
        due_df.to_csv(index=False), "renewals.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Client: Acme Inc **Ends In:** 25 days **GPT Output:**

```
- Renewal Action: Renew
- Reason: High-value client under auto-renew; no issues noted.
- Follow-up: Send renewal confirmation and schedule QBR.
```

🔧 Agent #71: Predictive Maintenance Agent

📝 Overview

This agent analyzes sensor or machine log data to predict equipment failures before they occur. It identifies performance degradation patterns, flags anomalies, and uses GPT to recommend preventive actions. In this lab, you'll build a simplified predictive maintenance assistant that evaluates simulated time-series data for a set of machines and suggests maintenance strategies.

🧪 Lab Objectives

By the end of this lab, you will:

- Load machine performance or sensor data (CSV)
- Calculate moving averages and anomalies
- Use GPT to interpret patterns and suggest actions
- Display actionable maintenance plans per machine

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Matplotlib**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir predictive_maintenance_agent
cd predictive_maintenance_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain matplotlib
```

✅ Step 2: Sample CSV (`machine_data.csv`)

```
Timestamp,MachineID,Temperature,Vibration,Pressure
2025-07-01 00:00:00,M01,75.2,0.35,120
2025-07-01 01:00:00,M01,78.4,0.55,128
2025-07-01 02:00:00,M01,82.1,0.75,140
...
```

✅ Step 3: GPT Prompt Template (`maintenance_prompt.py`)

```python
from langchain.prompts import PromptTemplate

maintenance_prompt = PromptTemplate.from_template("""
You are an AI predictive maintenance advisor.

Machine: {MachineID}
Temperature: {avg_temp:.2f}°C
Vibration: {avg_vib:.2f} mm/s
Pressure: {avg_pres:.2f} psi

Recent observations suggest:
{anomalies}

Provide:
1. Maintenance risk level (Low, Moderate, High)
2. Recommended action (e.g., inspect, replace, continue monitoring)
3. Justification (brief)

Keep it concise and technical.
""")
```

✅ Step 4: GPT Logic (`maintenance_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from maintenance_prompt import maintenance_prompt

def get_maintenance_advice(machine_id, avg_temp, avg_vib, avg_pres,
        anomalies):
    llm = ChatOpenAI(temperature=0.2)
    prompt = maintenance_prompt.format(
        MachineID=machine_id,
        avg_temp=avg_temp,
        avg_vib=avg_vib,
        avg_pres=avg_pres,
        anomalies=anomalies
```

```
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from maintenance_agent import get_maintenance_advice

st.title("🔧 Predictive Maintenance Agent")
st.caption("Analyze machine data and prevent failures")

file = st.file_uploader("Upload machine log CSV", type=["csv"])

if file:
    df = pd.read_csv(file, parse_dates=["Timestamp"])
    machines = df["MachineID"].unique()

    for m in machines:
        st.subheader(f"🛠️ Machine {m}")
        df_m = df[df["MachineID"] == m]

        # Basic anomaly checks
        anomalies = []
        if df_m["Temperature"].max() > 85:
            anomalies.append("High temperature spike")
        if df_m["Vibration"].mean() > 0.6:
            anomalies.append("Elevated vibration levels")
        if df_m["Pressure"].max() > 140:
            anomalies.append("Pressure exceeding threshold")

        advice = get_maintenance_advice(
            machine_id=m,
            avg_temp=df_m["Temperature"].mean(),
            avg_vib=df_m["Vibration"].mean(),
            avg_pres=df_m["Pressure"].mean(),
            anomalies=", ".join(anomalies) or "No significant issues."
        )

        st.markdown(advice)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Machine: M01 **GPT Output:**

```
- Risk Level: Moderate
- Action: Schedule inspection within 48 hours
- Reason: Vibration levels are elevated and pressure peaked above safe
thresholds. Proactive maintenance recommended.
```

🚚 Agent #72: Supply Chain Optimization Agent

📝 Overview

This agent analyzes supply chain data—inventory, supplier performance, lead times, and demand—to recommend improvements in sourcing, routing, and warehousing. In this lab, you'll build a supply chain analyzer that takes input data and uses GPT to generate insights on bottlenecks, risks, and optimization strategies.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a supply chain dataset (CSV)
- Analyze lead times, fill rates, and delivery performance
- Use GPT to suggest optimizations (e.g., vendor changes, route efficiency)
- Generate a summary dashboard of supply chain health

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir supply_chain_agent
cd supply_chain_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample CSV (`supply_chain.csv`)

```
OrderID,Supplier,Item,OrderDate,DeliveryDate,LeadTimeDays,QuantityOrde
ORD001,SupplierA,WidgetA,2025-06-01,2025-06-07,6,100,98
ORD002,SupplierB,WidgetB,2025-06-02,2025-06-12,10,200,180
ORD003,SupplierA,WidgetA,2025-06-05,2025-06-09,4,150,150
...
```

✅ Step 3: GPT Prompt Template (`supply_prompt.py`)

```python
from langchain.prompts import PromptTemplate

supply_prompt = PromptTemplate.from_template("""
You are a supply chain optimization consultant AI.

Supplier Summary:
{summary}

Please provide:
1. Top 2 inefficiencies or risks
2. Actionable recommendations (e.g., new vendor, buffer stock, faster
        freight)
3. High-level supply chain health status (Green, Yellow, Red)

Keep the response under 150 words.
""")
```

✅ Step 4: GPT Logic (`supply_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from supply_prompt import supply_prompt

def get_supply_chain_advice(summary: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = supply_prompt.format(summary=summary)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```python
import streamlit as st
import pandas as pd
from supply_agent import get_supply_chain_advice

st.title("🚚 Supply Chain Optimization Agent")
st.caption("Evaluate supplier performance and optimize logistics")
```

```
file = st.file_uploader("Upload supply chain data CSV", type=["csv"])

if file:
    df = pd.read_csv(file, parse_dates=["OrderDate", "DeliveryDate"])
    df["FillRate"] = df["QuantityReceived"] / df["QuantityOrdered"]
    summary = df.groupby("Supplier").agg({
        "LeadTimeDays": "mean",
        "FillRate": "mean",
        "OrderID": "count"
    }).rename(columns={"OrderID": "TotalOrders"}).round(2).to_string()

    st.subheader("📊 Supplier Performance Summary")
    st.text(summary)

    st.subheader("🧠 GPT Optimization Suggestions")
    suggestion = get_supply_chain_advice(summary)
    st.text_area("Recommendations", suggestion, height=250)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Inefficiencies: SupplierB shows low fill rate and long lead time.
SupplierA has variability in delivery windows.
- Recommendations: Explore alternate vendors for WidgetB, or
renegotiate delivery terms. Consider holding buffer stock for high-
demand items.
- Health Status: Yellow
```

🛠️ Agent #73: AI-powered Field Service Agent

📝 Overview

This agent assists in managing field technicians, work orders, and service schedules. It intelligently assigns jobs based on technician availability, location, and task urgency, and provides real-time suggestions for troubleshooting or rerouting. In this lab, you'll build an AI assistant to analyze incoming service jobs and assign them to optimal field agents with GPT-powered logic.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload job and technician data
- Match jobs to technicians using logic + GPT reasoning
- Flag scheduling or routing inefficiencies
- Generate actionable assignments per job

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Google Maps API or GeoJSON for routing)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir field_service_agent
cd field_service_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample CSVs

Technicians (`technicians.csv`)

```
TechID,Name,Location,SkillLevel,Availability
T001,Alice,Dallas,Expert,Yes
T002,Bob,Houston,Intermediate,Yes
T003,Clara,Austin,Beginner,No
```

Jobs (`jobs.csv`)

```
JobID,Issue,Location,Priority,RequiredSkill
J001,WiFi setup,Dallas,High,Beginner
J002,Server repair,Houston,Medium,Expert
J003,Printer install,Austin,Low,Intermediate
```

✅ Step 3: GPT Prompt Template (`field_prompt.py`)

```
from langchain.prompts import PromptTemplate

field_prompt = PromptTemplate.from_template("""
You are an AI field service dispatcher.
```

```
Match this job to the most appropriate technician:
- JobID: {JobID}
- Location: {Location}
- Priority: {Priority}
- Required Skill: {RequiredSkill}

Available Technicians:
{tech_summary}

Return:
1. Technician name
2. Reason for match (1 sentence)
3. Risk factors (if any)

Keep it short and actionable.
""")
```

✅ Step 4: Matching Logic (`field_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from field_prompt import field_prompt

def recommend_technician(job, tech_summary):
    llm = ChatOpenAI(temperature=0.2)
    prompt = field_prompt.format(
        JobID=job["JobID"],
        Location=job["Location"],
        Priority=job["Priority"],
        RequiredSkill=job["RequiredSkill"],
        tech_summary=tech_summary
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from field_agent import recommend_technician

st.title("🛠️ AI-powered Field Service Agent")
st.caption("Assign technicians to jobs using AI")

tech_file = st.file_uploader("Upload Technicians CSV", type=["csv"])
job_file = st.file_uploader("Upload Jobs CSV", type=["csv"])

if tech_file and job_file:
    tech_df = pd.read_csv(tech_file)
```

```
    job_df = pd.read_csv(job_file)

    available_techs = tech_df[tech_df["Availability"] == "Yes"]
    tech_summary = available_techs.to_string(index=False)

    results = []
    for _, job in job_df.iterrows():
        result = recommend_technician(job, tech_summary)
        results.append(result)

    job_df["Assignment"] = results
    st.dataframe(job_df[["JobID", "Issue", "Location", "Assignment"]])
    st.download_button("Download Assignments",
        job_df.to_csv(index=False), "field_assignments.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

Job: J002 – Server repair in Houston **GPT Output:**

```
- Assigned to: Bob
- Reason: Closest technician with availability and intermediate skill
level; only expert nearby.
- Risk: May require escalation if complexity is high.
```

📦 Agent #74: Inventory Management Agent

📝 Overview

This agent monitors stock levels, tracks inventory movement, and suggests restocking or redistribution strategies to avoid shortages or overstock. It flags slow-moving or high-turnover items and uses GPT to generate actionable insights. In this lab, you'll build an agent that processes inventory data and outputs inventory health insights and stock-level recommendations.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload current inventory data
- Analyze stock status and turnover rate
- Use GPT to provide restock or optimization advice
- Generate alerts for low stock or excess items

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir inventory_agent
cd inventory_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample Inventory CSV (`inventory.csv`)

```
ItemID,ItemName,Category,StockLevel,ReorderPoint,MonthlySales
I001,Widget A,Electronics,45,30,60
I002,Widget B,Electronics,15,20,5
I003,Gadget X,Appliances,120,50,10
I004,Tool Z,Hardware,8,25,35
```

✅ Step 3: GPT Prompt Template (`inventory_prompt.py`)

```
from langchain.prompts import PromptTemplate

inventory_prompt = PromptTemplate.from_template("""
You are an AI inventory optimization expert.

Item Info:
- Item: {ItemName}
- Category: {Category}
- Current Stock: {StockLevel}
- Reorder Point: {ReorderPoint}
- Monthly Sales: {MonthlySales}

Provide:
1. Restock recommendation (Yes/No)
2. Suggested Action (e.g., restock, reallocate, discontinue)
3. Risk Level (Low, Medium, High)
```

```
4. Justification (1 sentence)
""")
```

✅ Step 4: GPT Logic (`inventory_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from inventory_prompt import inventory_prompt

def analyze_inventory(item_dict):
    llm = ChatOpenAI(temperature=0.2)
    prompt = inventory_prompt.format(**item_dict)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from inventory_agent import analyze_inventory

st.title("📦 Inventory Management Agent")
st.caption("Get restock advice and optimize inventory")

file = st.file_uploader("Upload Inventory CSV", type=["csv"])

if file:
    df = pd.read_csv(file)
    results = []

    for _, row in df.iterrows():
        result = analyze_inventory(row.to_dict())
        results.append(result)

    df["GPT_Recommendation"] = results
    st.dataframe(df[["ItemName", "StockLevel", "MonthlySales",
        "GPT_Recommendation"]])
    st.download_button("Download Recommendations",
        df.to_csv(index=False), "inventory_recommendations.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

Item: Tool Z

```
- Restock: Yes
- Action: Reorder 30 units immediately
- Risk: High
- Reason: High monthly sales and stock is well below reorder point;
risk of stockout.
```

🏭 Agent #75: Warehouse Automation Agent

📝 Overview

This agent streamlines warehouse operations by analyzing order volume, storage layout, and item frequency to suggest automation strategies. It can identify inefficiencies in picking paths, space utilization, and labor allocation. In this lab, you'll build an agent that reads warehouse activity data and generates recommendations using GPT.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload warehouse operations data
- Identify fast/slow-moving items and picking frequency
- Use GPT to suggest layout and automation improvements
- Output actionable warehouse efficiency insights

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Matplotlib (optional)**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir warehouse_automation_agent
cd warehouse_automation_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain matplotlib
```

✅ Step 2: Sample Warehouse CSV (`warehouse_data.csv`)

```
ItemID,ItemName,Zone,DailyPickFrequency,StorageSpaceUsed,AvgPickTimeSe
I001,Widget A,Zone A,45,3.2,12
I002,Widget B,Zone B,10,5.1,25
I003,Gadget X,Zone C,2,6.7,30
I004,Tool Z,Zone A,60,2.9,10
```

✅ Step 3: GPT Prompt Template (`warehouse_prompt.py`)

```python
from langchain.prompts import PromptTemplate

warehouse_prompt = PromptTemplate.from_template("""
You are a warehouse automation advisor AI.

Item Data:
- Name: {ItemName}
- Zone: {Zone}
- Daily Picks: {DailyPickFrequency}
- Storage Used: {StorageSpaceUsed} m²
- Average Pick Time: {AvgPickTimeSecs} sec

Recommend:
1. Whether to automate handling (Yes/No)
2. Suggested change (e.g., robot picker, zone relocation, batching)
3. Impact on efficiency (High, Medium, Low)
4. One-sentence rationale
""")
```

✅ Step 4: GPT Logic (`warehouse_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from warehouse_prompt import warehouse_prompt

def recommend_automation(item_dict):
    llm = ChatOpenAI(temperature=0.3)
    prompt = warehouse_prompt.format(**item_dict)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from warehouse_agent import recommend_automation

st.title("🏭 Warehouse Automation Agent")
```

```
st.caption("Analyze picking patterns and optimize warehouse layout")

file = st.file_uploader("Upload Warehouse CSV", type=["csv"])

if file:
    df = pd.read_csv(file)
    recommendations = []

    for _, row in df.iterrows():
        rec = recommend_automation(row.to_dict())
        recommendations.append(rec)

    df["GPT_Recommendation"] = recommendations
    st.dataframe(df[["ItemName", "Zone", "DailyPickFrequency",
        "GPT_Recommendation"]])
    st.download_button("Download Report", df.to_csv(index=False),
        "warehouse_automation.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

Item: Widget A

```
- Automate: Yes
- Suggestion: Assign to robotic picker in Zone A
- Impact: High
- Rationale: High pick frequency and low pick time justify automation
for efficiency and labor savings.
```

🚚 Agent #76: AI-powered Logistics Routing Agent

📝 Overview

This agent analyzes delivery locations, vehicle capacity, route constraints, and delivery windows to generate optimal routing plans. It can minimize travel time, fuel cost, and delays using real-time and historical data. In this lab, you'll build an agent that reads delivery orders and generates AI-enhanced routing suggestions with GPT.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload delivery order and vehicle data

- Simulate route optimization logic
- Use GPT to provide intelligent routing suggestions
- Output routing plans and improvement ideas

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Geopandas (optional)**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Google Maps API or OpenRouteService)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir logistics_routing_agent
cd logistics_routing_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample CSVs

Orders (`orders.csv`)

```
OrderID,DestinationCity,DeliveryWindowStart,DeliveryWindowEnd,PackageWe
ORD001,Houston,08:00,12:00,100
ORD002,Dallas,10:00,14:00,200
ORD003,Austin,09:00,11:00,150
```

Vehicles (`vehicles.csv`)

```
VehicleID,Capacity,DepotCity,AvailableFrom,AvailableUntil
V001,500,Houston,07:00,19:00
V002,300,Dallas,06:00,18:00
```

✅ Step 3: GPT Prompt Template (`routing_prompt.py`)

```
from langchain.prompts import PromptTemplate

routing_prompt = PromptTemplate.from_template("""
You are a logistics optimization AI.

Given:
```

```
- Orders: {order_summary}
- Vehicles: {vehicle_summary}

Provide:
1. Assignment of orders to vehicles
2. Routing strategy (minimize total travel, honor delivery windows)
3. Risks (e.g., late deliveries, overloads)
4. One improvement recommendation

Use clear bullet points.
""")
```

✅ Step 4: GPT Logic (`routing_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from routing_prompt import routing_prompt

def optimize_routes(order_summary, vehicle_summary):
    llm = ChatOpenAI(temperature=0.3)
    prompt = routing_prompt.format(
        order_summary=order_summary,
        vehicle_summary=vehicle_summary
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from routing_agent import optimize_routes

st.title("🚚 AI-powered Logistics Routing Agent")
st.caption("Assign deliveries to vehicles and optimize routing")

order_file = st.file_uploader("Upload Orders CSV", type=["csv"])
vehicle_file = st.file_uploader("Upload Vehicles CSV", type=["csv"])

if order_file and vehicle_file:
    orders = pd.read_csv(order_file)
    vehicles = pd.read_csv(vehicle_file)

    order_summary = orders.to_string(index=False)
    vehicle_summary = vehicles.to_string(index=False)

    st.subheader("📝 Routing Plan Suggestion")
    suggestion = optimize_routes(order_summary, vehicle_summary)
    st.text_area("GPT Routing Output", suggestion, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Assign ORD001 & ORD003 to V001 (Houston depot, capacity 500)
- Assign ORD002 to V002 (Dallas depot)
- Route: Houston → Austin → Houston (V001); Dallas → Dallas (V002)
- Risk: Slight delay for ORD003 if Austin traffic worsens
- Suggestion: Add real-time traffic API integration to reroute
dynamically.
```

📈 Agent #77: Demand Planning Agent

📝 Overview

This agent forecasts product demand using historical sales, seasonality, and promotional data. It identifies trends, predicts stock requirements, and uses GPT to recommend procurement or marketing actions. In this lab, you'll build an agent that forecasts future demand from a dataset and generates intelligent business insights.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload historical demand data
- Forecast next month's demand using moving averages
- Use GPT to interpret the trends and suggest actions
- Output visual and text-based planning insights

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Matplotlib**
- **LangChain + GPT-4 or GPT-3.5**

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir demand_planning_agent
cd demand_planning_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain matplotlib
```

✅ Step 2: Sample CSV (`demand_data.csv`)

```
Month,Product,UnitsSold
2024-01,Widget A,120
2024-02,Widget A,140
2024-03,Widget A,100
2024-04,Widget A,160
2024-05,Widget A,180
2024-06,Widget A,150
```

✅ Step 3: GPT Prompt Template (`demand_prompt.py`)

```
from langchain.prompts import PromptTemplate

demand_prompt = PromptTemplate.from_template("""
You are a demand planning expert AI.

Product: {Product}
Historical Sales: {sales_list}
Predicted Demand Next Month: {forecast}

Provide:
1. Demand trend (e.g., rising, stable, falling)
2. Recommendation (e.g., increase order, launch promotion)
3. Risk level (Low, Medium, High)
4. One-sentence rationale
""")
```

✅ Step 4: GPT Logic (`demand_agent.py`)

```
from langchain.chat_models import ChatOpenAI
from demand_prompt import demand_prompt

def get_demand_advice(product, sales_list, forecast):
    llm = ChatOpenAI(temperature=0.2)
```

```
    prompt = demand_prompt.format(
        Product=product,
        sales_list=sales_list,
        forecast=forecast
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from demand_agent import get_demand_advice

st.title("📈 Demand Planning Agent")
st.caption("Forecast demand and generate recommendations")

file = st.file_uploader("Upload Demand CSV", type=["csv"])

if file:
    df = pd.read_csv(file, parse_dates=["Month"])
    product_list = df["Product"].unique()

    for product in product_list:
        st.subheader(f"📦 Product: {product}")
        df_p = df[df["Product"] == product].copy()
        df_p.sort_values("Month", inplace=True)
        df_p["MA_3"] = df_p["UnitsSold"].rolling(3).mean()

        forecast = round(df_p["MA_3"].iloc[-1], 2)
        sales_list = df_p["UnitsSold"].tolist()

        st.line_chart(df_p.set_index("Month")[["UnitsSold", "MA_3"]])
        st.markdown(f"**Forecast for next month:** {forecast} units")

        advice = get_demand_advice(product, sales_list, forecast)
        st.text_area("AI Recommendation", advice, height=200)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Trend: Rising
- Recommendation: Increase production and pre-order raw materials
- Risk Level: Medium
```

```
- Rationale: Strong upward trend in past 3 months; buffer stock
advised to prevent shortage during peak demand.
```

🔄 Agent #78: Workflow Automation Agent

📝 Overview

This agent automates repetitive business tasks by analyzing patterns in operations data. It identifies manual steps, repetitive approvals, bottlenecks, and suggests workflow automations—like email triggers, form processing, document routing, and notifications. In this lab, you'll build an AI agent that reads workflow data and uses GPT to recommend automation opportunities.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload workflow task logs or sequences
- Identify repetitive or manual tasks
- Use GPT to suggest automation strategies
- Output a summarized automation plan

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate with n8n, Zapier, or Make.com)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir workflow_automation_agent
cd workflow_automation_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample CSV (`workflow_log.csv`)

```
TaskID,Department,StepName,AssignedTo,StepType,AvgCompletionTimeMins,Is
T001,Finance,Invoice Approval,Alice,Approval,5,Yes
T002,HR,Onboarding Email,Bob,Email Trigger,2,Yes
T003,IT,Ticket Routing,System,Routing,1,No
T004,Marketing,Campaign Approval,Clara,Approval,10,Yes
T005,Sales,Proposal Generation,System,Document,3,No
```

✅ Step 3: GPT Prompt Template (`workflow_prompt.py`)

```python
from langchain.prompts import PromptTemplate

workflow_prompt = PromptTemplate.from_template("""
You are an AI business process consultant.

Workflow Data:
{workflow_summary}

For each task, recommend:
1. Can it be automated? (Yes/No)
2. Suggested tool (e.g., Zapier, Script, RPA)
3. Time savings potential (High/Medium/Low)
4. One-line justification

Use bullet points per task.
""")
```

✅ Step 4: GPT Logic (`workflow_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from workflow_prompt import workflow_prompt

def get_automation_advice(summary: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = workflow_prompt.format(workflow_summary=summary)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from workflow_agent import get_automation_advice

st.title("🔄 Workflow Automation Agent")
st.caption("Identify repetitive tasks and recommend automation tools")
```

```
file = st.file_uploader("Upload Workflow CSV", type=["csv"])

if file:
    df = pd.read_csv(file)
    summary = df.to_string(index=False)

    st.subheader("🧠 GPT Recommendations")
    recommendation = get_automation_advice(summary)
    st.text_area("Automation Suggestions", recommendation, height=300)

    st.dataframe(df)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Invoice Approval: Yes → Use Approval workflows in Power Automate;
High time savings; frequent manual action.
- Onboarding Email: Yes → Use Zapier or n8n email trigger; Medium time
savings; can be templated.
- Ticket Routing: No → Already automated.
- Campaign Approval: Yes → Route via Slack + Google Forms; Medium time
savings; often delays execution.
- Proposal Generation: No → System-handled.
```

📋 Agent #79: Automated Audit Agent

📝 Overview

The Automated Audit Agent reviews business process logs, financial transactions, or access records to identify anomalies, compliance issues, and irregularities. It flags suspicious entries, performs rule-based and AI-enhanced checks, and provides GPT-generated summaries. In this lab, you'll build an agent that scans data for audit anomalies and produces a concise audit report.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a mock audit dataset
- Identify rule violations and red flags
- Use GPT to generate an audit summary and risk analysis

- Export a structured audit report

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate rule engine like Great Expectations or Python `assert` rules)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir automated_audit_agent
cd automated_audit_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas openai langchain
```

✅ Step 2: Sample Audit Log CSV (`audit_data.csv`)

```
EntryID,Employee,Action,Amount,Department,Timestamp,PolicyViolation
E001,Alice,Expense Reimbursement,2500,Finance,2024-06-01 09:20,No
E002,Bob,Purchase Order,15000,Procurement,2024-06-02 10:15,Yes
E003,Clara,Data Access,0,IT,2024-06-03 11:00,Yes
E004,David,Travel Reimbursement,800,Sales,2024-06-03 15:10,No
E005,Eva,Bonus Allocation,5000,HR,2024-06-04 12:30,Yes
```

✅ Step 3: GPT Prompt Template (`audit_prompt.py`)

```
from langchain.prompts import PromptTemplate

audit_prompt = PromptTemplate.from_template("""
You are an internal audit AI assistant.

Analyze the following audit log:
{audit_summary}

For each entry with a policy violation:
1. Describe the issue
2. Assess risk level (Low, Medium, High)
3. Suggest action (e.g., flag for review, escalate, document)
4. One-sentence reasoning
```

```
End with a 3-bullet summary of the audit health.
""")
```

✅ Step 4: GPT Logic (`audit_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from audit_prompt import audit_prompt

def analyze_audit_data(summary: str):
    llm = ChatOpenAI(temperature=0.3)
    prompt = audit_prompt.format(audit_summary=summary)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import pandas as pd
from audit_agent import analyze_audit_data

st.title("📋 Automated Audit Agent")
st.caption("Scan logs and get AI-powered audit insights")

file = st.file_uploader("Upload Audit Log CSV", type=["csv"])

if file:
    df = pd.read_csv(file)
    summary = df[df["PolicyViolation"] ==
        "Yes"].to_string(index=False)

    st.subheader("🚨 GPT Audit Report")
    report = analyze_audit_data(summary)
    st.text_area("Audit Summary", report, height=350)

    st.dataframe(df)
    st.download_button("Download Audit Log", df.to_csv(index=False),
        "audit_report.csv")
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Bob: High risk. Unauthorized PO of $15,000. Action: Flag and
escalate. Reason: Large amount and policy breach.
- Clara: Medium risk. Accessed sensitive IT data without reason.
```

```
Action: Review access logs.
- Eva: High risk. Bonus allocation without documented approval.
Action: Escalate to HR lead.

Summary:
• 3 policy violations detected
• 2 high-risk events flagged
• Recommend immediate review and access audit in IT
```

🧾 Agent #8: Tax Compliance Agent

📝 Overview

This AI agent checks whether a company's financial records comply with tax rules — including sales tax, VAT, corporate income tax thresholds, and payroll obligations. In this lab, we'll simulate financial records and build a GPT-powered assistant that flags potential compliance issues and recommends corrections.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate income, expenses, and tax liabilities
- Identify potential non-compliance (e.g., underpayment, missing VAT)
- Use GPT to generate a tax audit-style summary
- Present everything via a Streamlit interface

🧰 Tech Stack

- **Python**
- **Pandas**
- **LangChain + GPT-4 or GPT-3.5**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If continuing from earlier labs, skip this. Otherwise:

```bash
mkdir tax_compliance_agent
cd tax_compliance_agent
python -m venv venv
```

```
source venv/bin/activate
pip install openai langchain pandas streamlit
```

✅ Step 2: Simulate financial records (`tax_data.py`)

```
import pandas as pd

def get_tax_records():
    data = {
        "Category": ["Revenue", "COGS", "Operating Expenses", "VAT
        Collected", "VAT Paid", "Payroll Tax", "Corporate Income Tax
        Paid"],
        "Amount": [120000, 40000, 25000, 8000, 3500, 7000, 5000]
    }
    return pd.DataFrame(data)
```

✅ Step 3: Create the GPT prompt (`tax_prompt.py`)

```
from langchain.prompts import PromptTemplate

tax_template = PromptTemplate.from_template("""
You are a tax compliance auditor AI.

Here is a company's financial summary:

{tax_table}

Please:
1. Identify any discrepancies in VAT reporting (collected vs paid).
2. Estimate whether income tax paid is reasonable based on net profit.
3. Flag any areas of non-compliance or audit risk.
4. Recommend corrective actions.

Be concise and audit-ready in tone.
""")
```

✅ Step 4: Run GPT-based tax check (`tax_agent.py`)

```
from tax_data import get_tax_records
from tax_prompt import tax_template
from langchain.chat_models import ChatOpenAI

def analyze_tax():
    df = get_tax_records()
    table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.2)
```

```
    prompt = tax_template.format(tax_table=table)
    result = llm.predict(prompt)
    return df, result
```

✅ Step 5: Build the Streamlit interface (`app.py`)

```
import streamlit as st
from tax_agent import analyze_tax

st.title("🧾 Tax Compliance Agent")

if st.button("Audit Tax Records"):
    df, result = analyze_tax()

    st.subheader("📂 Financial Summary")
    st.dataframe(df)

    st.subheader("🧠 AI Tax Compliance Report")
    st.write(result)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
1. VAT collected is $8,000 while VAT paid is $3,500 — net VAT owed is
$4,500. Ensure timely remittance.

2. Net income (Revenue - COGS - Expenses) is $55,000. Corporate tax
paid is $5,000, implying an effective tax rate of ~9%, which may be
under-reported if local rate is 15–20%.

3. Payroll tax appears proportionate, but further employee count
validation is advised.

Recommendation:
- File updated VAT returns.
- Reassess income tax estimation methodology.
- Keep documentation ready for audit readiness.
```

🧠 Agent #80: AI-powered Knowledge Management Agent

📝 Overview

This agent ingests internal documentation, wikis, meeting notes, and SOPs to create a smart knowledge base. It enables employees to query company knowledge using natural language, surfacing accurate, context-aware answers. In this lab, you'll build a chatbot that indexes documents and responds to user queries using GPT and embeddings.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload internal documents (PDFs, text, Markdown)
- Generate vector embeddings for semantic search
- Query the knowledge base via natural language
- Use GPT to provide context-rich answers

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + OpenAI Embeddings + GPT-4/3.5**
- **FAISS (or ChromaDB) for vector search**
- *(Optional: Pinecone or Weaviate for external vector DBs)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir knowledge_agent
cd knowledge_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu tiktoken
```

✅ Step 2: Upload Sample Docs

Put your `.md`, `.txt`, or `.pdf` files into a folder:

```
knowledge_agent/
|
```

```
├── docs/
│   ├── onboarding.md
│   ├── expense_policy.txt
│   └── product_faq.txt
```

✅ Step 3: Ingest and Embed (`embed_docs.py`)

```python
import os
from langchain.document_loaders import TextLoader
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.text_splitter import CharacterTextSplitter

def ingest_docs():
    docs = []
    for filename in os.listdir("docs"):
        if filename.endswith(".txt") or filename.endswith(".md"):
            loader = TextLoader(os.path.join("docs", filename))
            docs.extend(loader.load())

    splitter = CharacterTextSplitter(chunk_size=1000,
        chunk_overlap=100)
    chunks = splitter.split_documents(docs)

    embeddings = OpenAIEmbeddings()
    vectorstore = FAISS.from_documents(chunks, embeddings)
    vectorstore.save_local("kb_index")
```

Run this once:

```
python embed_docs.py
```

✅ Step 4: GPT-Powered QA (`knowledge_agent.py`)

```python
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def create_qa_chain():
    embeddings = OpenAIEmbeddings()
    db = FAISS.load_local("kb_index", embeddings)
    retriever = db.as_retriever(search_type="similarity", k=3)

    llm = ChatOpenAI(temperature=0)
    chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
    return chain
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
from knowledge_agent import create_qa_chain

st.title("🧠 AI Knowledge Management Agent")
st.caption("Ask anything from your internal documents")

query = st.text_input("❓ Ask a question")
if query:
    chain = create_qa_chain()
    response = chain.run(query)
    st.markdown(f"**Answer:** {response}")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Queries

- “What is the reimbursement policy for remote employees?”
- “Summarize the onboarding checklist for new hires.”
- “What’s the SLA for internal ticket handling?”

⚖️ Agent #81: Contract Review Agent

📝 Overview

The Contract Review Agent analyzes legal contracts to identify risky clauses, missing terms, deadlines, and compliance red flags. It assists legal teams by flagging language inconsistencies, summarizing obligations, and suggesting improvements. In this lab, you’ll build a GPT-powered contract analyzer that reviews uploaded contracts and provides structured legal insights.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload contract documents (plain text or PDF)
- Extract and chunk clauses for analysis
- Use GPT to review and highlight key insights
- Display a summary of risks, missing elements, and obligations

---

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **PyMuPDF (`fitz`) for PDF reading**
- *(Optional: Use OCR for scanned PDFs)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir contract_review_agent
cd contract_review_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai pymupdf tiktoken
```

✅ Step 2: Sample Contract (`sample_contract.txt`)

```
This agreement is entered into on July 15, 2025, between Acme Corp and
Beta Inc.

1. Payment Terms: Payments shall be made within 45 days of invoice.
2. Termination Clause: Either party may terminate with 30 days'
notice.
3. Confidentiality: All terms and communications must remain
confidential.
4. Indemnity: Beta Inc agrees to indemnify Acme Corp for losses
arising from misuse.
5. Governing Law: This agreement is governed by the laws of
California.
```

✅ Step 3: GPT Prompt Template (`contract_prompt.py`)

```
from langchain.prompts import PromptTemplate

contract_prompt = PromptTemplate.from_template("""
You are a contract analysis AI.

Clause:
{clause}

Review the clause and return:
1. Clause Type (e.g., Payment, Termination, Indemnity)
```

```
2. Risk Level (Low/Medium/High)
3. Summary of Obligation
4. Suggestion to improve clarity or compliance
""")
```

✅ Step 4: GPT Logic (`review_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from contract_prompt import contract_prompt

def analyze_clause(clause):
    llm = ChatOpenAI(temperature=0.3)
    prompt = contract_prompt.format(clause=clause)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from review_agent import analyze_clause

st.title("⚖️ Contract Review Agent")
st.caption("Upload and analyze legal clauses for risk and compliance")

file = st.file_uploader("Upload Contract (.txt)", type=["txt"])

if file:
    clauses = file.read().decode("utf-8").split("\n")
    clauses = [c.strip() for c in clauses if c.strip()]

    for clause in clauses:
        st.markdown(f"**📄 Clause:** `{clause}`")
        output = analyze_clause(clause)
        st.text_area("🧠 GPT Analysis", output, height=200)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

Clause: Termination Clause: Either party may terminate with 30 days' notice.

```
- Clause Type: Termination
- Risk Level: Low
- Obligation: Either party can end the agreement with 30 days' notice.
- Suggestion: Specify acceptable notice methods (email, certified
mail) to avoid disputes.
```

🛡️ Agent #82: Regulatory Compliance Agent

📝 Overview

The Regulatory Compliance Agent scans internal policies, operational procedures, and documentation to assess alignment with external regulations (like GDPR, HIPAA, SOX). It flags missing elements, non-compliance risks, and recommends actionable improvements. In this lab, you'll build an agent that reviews text-based documents and produces a GPT-generated compliance gap report.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload internal policy or procedure documents
- Select a target regulation (e.g., GDPR)
- Analyze for coverage and alignment
- Receive a structured compliance summary with improvement suggestions

---

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Regulatory clause templates for rule matching)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir compliance_agent
cd compliance_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai tiktoken
```

✅ Step 2: Sample Policy Document (`data_policy.txt`)

```
Our company collects customer data only with consent and uses it for
internal purposes.
Data is stored on secure servers, and employees are trained to handle
```

```
it responsibly.
Data is not shared with third parties unless required by law. Users
may request deletion.
```

✅ Step 3: GPT Prompt Template (`compliance_prompt.py`)

```python
from langchain.prompts import PromptTemplate

compliance_prompt = PromptTemplate.from_template("""
You are a regulatory compliance AI assistant.

Policy Document:
{policy_text}

Target Regulation: {regulation_name}

Analyze and return:
1. Compliance Level (Compliant / Partially Compliant / Non-Compliant)
2. Missing or weak areas
3. Recommended action steps
4. Overall risk score (Low/Medium/High)
5. One-sentence executive summary
""")
```

✅ Step 4: GPT Logic (`compliance_agent.py`)

```python
from langchain.chat_models import ChatOpenAI
from compliance_prompt import compliance_prompt

def check_compliance(policy_text, regulation_name):
    llm = ChatOpenAI(temperature=0.3)
    prompt = compliance_prompt.format(
        policy_text=policy_text,
        regulation_name=regulation_name
    )
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
from compliance_agent import check_compliance

st.title("🛡️ Regulatory Compliance Agent")
st.caption("Check your policy documents for compliance with major
        regulations")

file = st.file_uploader("Upload Policy Text (.txt)", type=["txt"])
```

```
regulation = st.selectbox("Select Regulation", ["GDPR", "HIPAA",
        "SOX", "CCPA", "PCI DSS"])

if file and regulation:
    text = file.read().decode("utf-8")
    result = check_compliance(text, regulation)

    st.subheader("📋 Compliance Summary")
    st.text_area("GPT Compliance Output", result, height=350)
```

Run the app:

```
streamlit run app.py
```

🧪 Example GPT Output

```
- Compliance Level: Partially Compliant
- Weakness: No mention of user data breach notification timelines
(GDPR Article 33)
- Action: Add a breach response policy and 72-hour reporting clause
- Risk Score: Medium
- Summary: Document shows effort toward GDPR compliance but lacks
specificity in breach handling and data portability clauses.
```

🔍 Agent #83: AI-powered E-Discovery Agent

📝 Overview

The AI-powered E-Discovery Agent streamlines the process of identifying, collecting, and analyzing digital evidence for legal or compliance investigations. It scans emails, documents, and logs to detect relevant communications, sensitive terms, or legal triggers. In this lab, you'll build an agent that ingests documents and extracts relevant information based on a legal query.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload document files (emails, memos, text)
- Input a legal discovery keyword or query
- Use semantic search to extract relevant content
- Generate GPT-based summaries for legal review

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **FAISS for vector search**
- *(Optional: Use OCR for image-based files)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ediscovery_agent
cd ediscovery_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu tiktoken
```

✅ Step 2: Sample Documents Folder (`evidence/`)

Create a folder with example `.txt` or `.md` documents:

```
Subject: Vendor Termination
We've reviewed the contract and due to breach, we will terminate the
agreement with Beta Inc as of July 1st.

Subject: Data Breach
IT confirmed unauthorized access occurred on June 10. We may need to
notify customers under GDPR.
```

✅ Step 3: Document Embedding (`embed_docs.py`)

```python
import os
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS

def embed_documents():
    docs = []
    for fname in os.listdir("evidence"):
        if fname.endswith(".txt") or fname.endswith(".md"):
            loader = TextLoader(f"evidence/{fname}")
            docs.extend(loader.load())
```

```
    splitter = CharacterTextSplitter(chunk_size=800,
        chunk_overlap=100)
    chunks = splitter.split_documents(docs)

    embedding = OpenAIEmbeddings()
    vectordb = FAISS.from_documents(chunks, embedding)
    vectordb.save_local("ediscovery_index")
```

Run:

```
python embed_docs.py
```

✅ Step 4: GPT Search Logic (`ediscovery_agent.py`)

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def create_discovery_chain():
    db = FAISS.load_local("ediscovery_index", OpenAIEmbeddings())
    retriever = db.as_retriever(search_type="similarity", k=3)
    llm = ChatOpenAI(temperature=0)
    return RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from ediscovery_agent import create_discovery_chain

st.title("🔍 AI-Powered E-Discovery Agent")
st.caption("Search internal documents for legal or compliance
        discovery")

query = st.text_input("🔎 Enter keyword or legal issue (e.g.,
        termination, GDPR breach)")

if query:
    chain = create_discovery_chain()
    response = chain.run(query)
    st.markdown("### 📄 Extracted Insight")
    st.text_area("Relevant Content Summary", response, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Use Case

Query: "GDPR breach" **Response:**

```
- Detected email mentioning unauthorized data access on June 10.
- Suggests GDPR notification obligation may apply.
- Mentions IT confirmation of the breach.
```

🧬 Agent #84: Intellectual Property Monitoring Agent

📝 Overview

The Intellectual Property (IP) Monitoring Agent helps organizations safeguard their patents, trademarks, and copyrights by scanning public databases, web content, and competitor filings for potential infringement or unauthorized use. In this lab, you'll build an agent that monitors trademark usage and flag suspicious similarities using semantic search and GPT reasoning.

🧪 Lab Objectives

By the end of this lab, you will:

- Input your company's trademarks or IP keywords
- Search across sample competitor listings or web excerpts
- Use embeddings to match similar terms or content
- Get GPT-powered alerts about potential IP violations

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **FAISS for similarity search**
- *(Optional: Scrapy or SerpAPI for live web crawling)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```bash
mkdir ip_monitor_agent
cd ip_monitor_agent
python -m venv venv
```

```
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu tiktoken
```

✅ Step 2: Sample Competitor Data (`competitor_content.txt`)

```
Introducing the “VitaNova” smartwatch — combining wellness with AI-powered heart tracking.

BetaTech’s “SmartPulse” sensor now integrates sleep patterns and activity recognition.

New filing: “VitaMove” trademark submitted by QuantumMotion Inc. for fitness wearables.

Next-gen “NeuraPulse” headset filed for patent under cognitive tracking class.
```

✅ Step 3: Document Embedding (`embed_competitors.py`)

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS

loader = TextLoader("competitor_content.txt")
documents = loader.load()

splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=100)
chunks = splitter.split_documents(documents)

embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(chunks, embeddings)
vectorstore.save_local("ip_index")
```

Run:

```
python embed_competitors.py
```

✅ Step 4: GPT Evaluation Logic (`ip_agent.py`)

```python
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def run_ip_monitoring(keyword):
    db = FAISS.load_local("ip_index", OpenAIEmbeddings())
```

```
    retriever = db.as_retriever(search_type="similarity", k=3)
    llm = ChatOpenAI(temperature=0.3)
    chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
    return chain.run(f"Check for IP conflicts related to: {keyword}")
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
from ip_agent import run_ip_monitoring

st.title("🧬 IP Monitoring Agent")
st.caption("Track your trademarks against competitor activity")

keyword = st.text_input("Enter your brand or IP name (e.g.,
        VitaPulse)")

if keyword:
    result = run_ip_monitoring(keyword)
    st.markdown("### 🛡️ IP Risk Analysis")
    st.text_area("GPT Summary", result, height=300)
```

Run:

```
streamlit run app.py
```

🧪 Example Query

Input: `VitaPulse`

GPT Response:

```
- Found competitor product “VitaNova” which may cause brand confusion
in wearables.
- Trademark “VitaMove” also shows overlapping market space.
- Recommendation: Conduct legal review and file an opposition if
needed.
```

📝 Agent #85: Policy Drafting Agent

🧭 Overview

The Policy Drafting Agent helps organizations generate clear, compliant, and customized internal policies—such as security, privacy, HR, and IT policies—using structured templates and regulatory references. In this lab, you’ll build an agent that collects user inputs about policy goals and outputs a polished, GPT-written policy document ready for review and implementation.

🧪 Lab Objectives

By the end of this lab, you will:

- Choose a type of policy to generate (e.g., Remote Work, Data Privacy)
- Enter company-specific variables (e.g., office hours, compliance requirements)
- Use GPT to draft the full policy
- Allow download of the drafted policy as text or Markdown

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add a policy database for referencing real-world examples)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir policy_drafting_agent
cd policy_drafting_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai
```

✅ Step 2: Policy Prompt Template (`policy_prompt.py`)

```
from langchain.prompts import PromptTemplate

policy_prompt = PromptTemplate.from_template("""
You are a policy drafting assistant. Based on the details below, write
        a clear and professional {policy_type} policy.

Company Name: {company_name}
Industry: {industry}
Policy Requirements: {requirements}
Tone: Professional, clear, formal
Format: Markdown

Include sections like:
1. Purpose
2. Scope
3. Policy Statement
```

```
4. Roles and Responsibilities
5. Compliance
6. Review Schedule
""")
```

✅ Step 3: GPT Logic (`policy_writer.py`)

```
from langchain.chat_models import ChatOpenAI
from policy_prompt import policy_prompt

def generate_policy(policy_type, company_name, industry,
        requirements):
    llm = ChatOpenAI(temperature=0.4)
    prompt = policy_prompt.format(
        policy_type=policy_type,
        company_name=company_name,
        industry=industry,
        requirements=requirements
    )
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Interface (`app.py`)

```
import streamlit as st
from policy_writer import generate_policy

st.title("📝 Policy Drafting Agent")
st.caption("Generate professional internal policies in minutes")

company = st.text_input("Company Name", "Acme Corp")
industry = st.selectbox("Industry", ["Tech", "Healthcare", "Finance",
        "Education", "Other"])
policy_type = st.selectbox("Policy Type", ["Remote Work", "Data
        Privacy", "Code of Conduct", "BYOD", "Security"])
requirements = st.text_area("Policy Requirements (optional)", "Should
        comply with GDPR and include work hours and device usage
        rules.")

if st.button("Generate Policy"):
    policy = generate_policy(policy_type, company, industry,
        requirements)
    st.subheader("📄 Drafted Policy")
    st.text_area("Output", policy, height=500)
    st.download_button("Download Policy", data=policy, file_name=f"
        {policy_type}_Policy.md")
```

Run:

```
streamlit run app.py
```

🧪 Example Output

Policy Type: Remote Work **Requirements:** "Employees must be reachable 9am–5pm EST, and use only company-approved VPN for remote access."

GPT Output (Excerpt):

```
### 1. Purpose
This Remote Work Policy establishes guidelines to ensure effective and
secure remote operations for Acme Corp employees.

### 3. Policy Statement
All remote employees must be available during standard business hours
(9am–5pm EST). Remote access is only allowed via approved VPN clients
to ensure data security...
```

🛡️ Agent #86: Data Privacy Compliance Agent

📝 Overview

The Data Privacy Compliance Agent helps organizations evaluate and align their data handling practices with privacy regulations like GDPR, CCPA, and HIPAA. It analyzes internal documents, privacy statements, and data flows to flag compliance gaps and suggest actionable remediation. In this lab, you'll build an agent that reviews privacy policies and checks alignment with selected regulations.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a company privacy policy
- Choose a regulation to audit against (e.g., GDPR or CCPA)
- Use GPT to evaluate the text for compliance gaps
- Generate a structured gap analysis report

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add rule-based regex checks or a regulation clause database)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir privacy_compliance_agent
cd privacy_compliance_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai tiktoken
```

✅ Step 2: Sample Privacy Policy (`privacy_policy.txt`)

```
We collect personal data such as name, email, and IP address to
improve user experience. Data is stored on encrypted servers. Users
can request account deletion by email.
We do not sell user data to third parties. We may use cookies and
third-party analytics.
```

✅ Step 3: Prompt Template (`compliance_prompt.py`)

```
from langchain.prompts import PromptTemplate

privacy_prompt = PromptTemplate.from_template("""
You are a data privacy compliance assistant. Review the following
        policy text and check its alignment with {regulation}.

Policy Text:
{policy_text}

Return:
1. Compliance Level: Fully / Partially / Non-Compliant
2. Missing Elements (based on the regulation)
3. Recommendations for Remediation
4. Risk Score (Low/Medium/High)
5. One-paragraph Executive Summary
""")
```

✅ Step 4: GPT Logic (`privacy_audit.py`)

```
from langchain.chat_models import ChatOpenAI
from compliance_prompt import privacy_prompt

def audit_privacy(policy_text, regulation):
    llm = ChatOpenAI(temperature=0.3)
```

```
    prompt = privacy_prompt.format(policy_text=policy_text,
        regulation=regulation)
    return llm.predict(prompt)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from privacy_audit import audit_privacy

st.title("🛡️ Data Privacy Compliance Agent")
st.caption("Analyze your privacy policy against GDPR or CCPA")

regulation = st.selectbox("Select Regulation", ["GDPR", "CCPA",
        "HIPAA"])
file = st.file_uploader("Upload Your Privacy Policy (.txt)", type=
        ["txt"])

if file:
    text = file.read().decode("utf-8")
    result = audit_privacy(text, regulation)

    st.subheader("🔍 Compliance Report")
    st.text_area("GPT Analysis", result, height=400)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Regulation: GDPR **Summary:**

```
- Compliance Level: Partially Compliant
- Missing Elements: No mention of data retention period, no DPO
contact, no explicit consent mechanism
- Recommendation: Add details on user consent, retention duration, and
contact info for data controller
- Risk Score: Medium
- Executive Summary: The policy covers basic data handling but lacks
key GDPR requirements for consent, control, and user rights
communication.
```

⚖️ Agent #87: Litigation Risk Assessment Agent

📝 Overview

The Litigation Risk Assessment Agent helps organizations detect early signs of potential legal disputes by analyzing internal communication, contracts, complaints, and operational data. It evaluates tone, trigger phrases, clause risks, and behavior patterns to assess the likelihood and severity of litigation exposure. In this lab, you'll build an agent that analyzes textual records and provides a litigation risk score with GPT-generated rationale.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload internal documents (e.g., complaints, emails, contracts)
- Use semantic search to extract relevant content
- Use GPT to analyze and score litigation risk
- Receive a structured risk report with suggested mitigation steps

---

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **FAISS for similarity search**
- *(Optional: Add Named Entity Recognition for involved parties or departments)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir litigation_risk_agent
cd litigation_risk_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu tiktoken
```

✅ Step 2: Sample Internal Records (`records/`)

Create `.txt` or `.md` files with content such as:

```
Subject: Vendor Dispute
Beta Inc has refused to fulfill contract terms citing ambiguous
service level agreements. Legal may need to step in.

Subject: Customer Complaint
Client raised repeated concerns about misleading marketing language
and delayed refunds. Escalated twice.
```

✅ Step 3: Embedding Internal Records (`embed_records.py`)

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
import os

docs = []
for file in os.listdir("records"):
    if file.endswith(".txt"):
        loader = TextLoader(f"records/{file}")
        docs.extend(loader.load())

splitter = CharacterTextSplitter(chunk_size=700, chunk_overlap=100)
chunks = splitter.split_documents(docs)

embeddings = OpenAIEmbeddings()
db = FAISS.from_documents(chunks, embeddings)
db.save_local("litigation_index")
```

Run the script:

```
python embed_records.py
```

✅ Step 4: GPT Logic (`risk_agent.py`)

```python
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def assess_litigation_risk(query):
    db = FAISS.load_local("litigation_index", OpenAIEmbeddings())
    retriever = db.as_retriever(search_type="similarity", k=3)
    llm = ChatOpenAI(temperature=0.2)
    chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
    return chain.run(f"Evaluate the litigation risk of the following
        situation: {query}")
```

✅ Step 5: Streamlit App (`app.py`)

```
import streamlit as st
from risk_agent import assess_litigation_risk

st.title("⚖️ Litigation Risk Assessment Agent")
st.caption("Analyze internal records for legal risk exposure")

query = st.text_area("Describe a situation or upload content hinting
        at potential litigation")

if query:
    result = assess_litigation_risk(query)
    st.subheader("📉 Risk Evaluation")
    st.text_area("GPT Risk Report", result, height=400)
```

Run:

```
streamlit run app.py
```

🧪 Example Output

Input: "Beta Inc is claiming SLA ambiguity and may file breach of contract lawsuit."

GPT Output:

```
- Risk Level: High
- Rationale: Language suggests imminent legal escalation due to
contractual ambiguity and refusal to deliver.
- Recommendation: Engage legal counsel, review SLA language, and
initiate mediation if applicable.
- Likely Clause Risk: Ambiguous performance criteria and lack of
resolution process.
```

📚 Agent #88: Ethics & Compliance Training Agent

📝 Overview

The Ethics & Compliance Training Agent delivers personalized, AI-generated microlearning modules to educate employees on corporate ethics, data privacy, anti-corruption, harassment prevention, and regulatory obligations. It adapts to roles, learning styles, and risk exposure. In this lab, you'll build an agent that creates a short, interactive training module based on a selected topic and employee role.

🧪 Lab Objectives

By the end of this lab, you will:

- Select a training topic and employee role
- Generate a structured microlearning module with GPT
- Include scenario-based questions and explanations
- Allow export of the training content for LMS or email distribution

---

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: LMS integration or SCORM export)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ethics_training_agent
cd ethics_training_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai
```

✅ Step 2: Prompt Template (`training_prompt.py`)

```
from langchain.prompts import PromptTemplate

training_prompt = PromptTemplate.from_template("""
You are a corporate ethics training assistant.

Create a short, interactive learning module on the topic: **{topic}**
Target Audience: {role}

Your output should include:
1. Overview (3–4 sentence summary)
2. Key Guidelines (3–5 bullet points)
3. Scenario-Based Quiz (1 realistic scenario + 3 MCQ options + correct
      answer + explanation)
4. Reminder of Consequences or Real-world Relevance
""")
```

✅ Step 3: GPT Training Generator (`generate_training.py`)

```
from langchain.chat_models import ChatOpenAI
from training_prompt import training_prompt

def generate_training(topic, role):
    llm = ChatOpenAI(temperature=0.5)
    prompt = training_prompt.format(topic=topic, role=role)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Interface (`app.py`)

```
import streamlit as st
from generate_training import generate_training

st.title("📚 Ethics & Compliance Training Agent")
st.caption("Generate bite-sized, role-based training for your teams")

topic = st.selectbox("Select Topic", [
    "Anti-Corruption", "Data Privacy", "Harassment Prevention",
    "Insider Trading", "Conflict of Interest", "Diversity & Inclusion"
])
role = st.selectbox("Select Role", [
    "Software Engineer", "Sales Executive", "HR Manager", "Finance
        Analyst", "Executive", "General Staff"
])

if st.button("Generate Training Module"):
    module = generate_training(topic, role)
    st.subheader("📘 Microlearning Module")
    st.text_area("Generated Content", module, height=500)
    st.download_button("Download Module", data=module, file_name=f"
        {topic}_Training_{role}.md")
```

Run:

```
streamlit run app.py
```

🧪 Example Output (Topic: Harassment Prevention – Role: Software Engineer)

Overview: Harassment in the workplace can be verbal, physical, or psychological. Tech teams must maintain respectful interactions, whether in-person or remote.

Key Guidelines:

- Use respectful and inclusive language
- Avoid comments on personal appearance
- Speak up or report when you witness inappropriate behavior

- Understand your company's escalation policy

Scenario-Based Quiz: *A colleague repeatedly sends GIFs in the team Slack channel with suggestive jokes. What should you do?* A. Laugh it off privately B. Report the behavior to HR or a team lead ✓ C. Confront them in the group channel **Explanation:** B is correct —company policy requires reporting to HR or management, not confronting the person in public.

Relevance Reminder: Even minor incidents can escalate or create a toxic environment. Companies have faced lawsuits over overlooked behavior.

⚙️ Agent #89: Automated Risk Management Agent

📝 Overview

The Automated Risk Management Agent continuously monitors business activities, evaluates risk exposure across departments, and suggests mitigation strategies based on historical incidents, operational thresholds, and external regulations. It acts as a digital risk officer by combining real-time data with AI-driven insights. In this lab, you'll build an agent that takes structured input about business risks and returns GPT-based assessments and recommendations.

🧪 Lab Objectives

By the end of this lab, you will:

- Enter risk category, description, and business unit
- Use GPT to generate a severity rating and explanation
- Get mitigation strategies based on the input
- Export the risk report for internal compliance use

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Database to store recurring risks and a risk scoring matrix)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir risk_mgmt_agent
cd risk_mgmt_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai
```

✅ Step 2: Prompt Template (`risk_prompt.py`)

```
from langchain.prompts import PromptTemplate

risk_prompt = PromptTemplate.from_template("""
You are a business risk management AI assistant.

Based on the following details, assess the risk severity and suggest
        mitigation steps:

- Category: {category}
- Description: {description}
- Business Unit: {unit}

Your output should include:
1. Risk Severity (Low, Medium, High, Critical)
2. Likelihood of Occurrence
3. Impact Explanation
4. Recommended Mitigation Strategies
5. Compliance Considerations (if any)
6. Executive Summary (3 sentences)
""")
```

✅ Step 3: GPT Risk Engine (`risk_evaluator.py`)

```
from langchain.chat_models import ChatOpenAI
from risk_prompt import risk_prompt

def evaluate_risk(category, description, unit):
    llm = ChatOpenAI(temperature=0.4)
    prompt = risk_prompt.format(category=category,
        description=description, unit=unit)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit App (`app.py`)

```
import streamlit as st
from risk_evaluator import evaluate_risk

st.title("⚙️ Automated Risk Management Agent")
st.caption("Analyze and manage risks across departments")

category = st.selectbox("Risk Category", ["Operational", "Financial",
        "Cybersecurity", "Compliance", "Reputational", "Strategic"])
unit = st.text_input("Business Unit (e.g., HR, Finance, IT)", "IT")
description = st.text_area("Risk Description", "Cloud vendor may
        discontinue service with 30-day notice")

if st.button("Evaluate Risk"):
    result = evaluate_risk(category, description, unit)
    st.subheader("📄 Risk Assessment Report")
    st.text_area("GPT Report", result, height=500)
    st.download_button("Download Report", data=result, file_name=f"
        {category}_Risk_Report_{unit}.md")
```

Run:

```
streamlit run app.py
```

🧪 Example Output

Category: Cybersecurity **Description:** Cloud vendor may discontinue service with 30-day notice **Unit:** IT

GPT Response (Excerpt):

```
1. Risk Severity: High
2. Likelihood: Moderate
3. Impact: Service disruption could halt major applications hosted on
the cloud, affecting 70% of operations.
4. Mitigation: Establish backup providers, initiate multi-cloud
architecture, negotiate 60-day termination clauses.
5. Compliance: Check data residency and backup access policies under
GDPR.
6. Executive Summary: This risk poses high operational and
reputational threats. Mitigation should begin with alternative vendor
assessment and resilience planning.
```

📊 Agent #9: Budget Optimization Agent

📝 Overview

This AI agent reviews historical budget allocations, identifies inefficient spending, and suggests optimized distributions aligned with strategic priorities. In this lab, you'll simulate department-level spending and use GPT to recommend budget reallocations and cost-saving measures.

🧪 Lab Objectives

By the end of this lab, you will:

- Simulate departmental budget vs. actual spending data
- Identify overspending and underutilization
- Use GPT to generate reallocation recommendations
- Visualize results with Streamlit and a summary report

🧰 Tech Stack

- **Python**
- **Pandas + Matplotlib**
- **LangChain + OpenAI (GPT-3.5 or GPT-4)**
- **Streamlit**

🧭 Step-by-Step Instructions

✅ Step 1: Set up your environment

If you've completed earlier labs, you can skip this:

```
mkdir budget_optimizer_agent
cd budget_optimizer_agent
python -m venv venv
source venv/bin/activate
pip install openai langchain pandas matplotlib streamlit
```

✅ Step 2: Create mock budget data (`budget_data.py`)

```
import pandas as pd

def load_budget_data():
    data = {
```

```
        "Department": ["Marketing", "Sales", "Engineering", "HR",
        "IT", "Operations"],
        "Allocated Budget": [80000, 60000, 150000, 40000, 50000,
        70000],
        "Actual Spending": [95000, 58000, 140000, 30000, 60000, 85000]
    }
    return pd.DataFrame(data)
```

✅ Step 3: Create GPT prompt template (`budget_prompt.py`)

```
from langchain.prompts import PromptTemplate

budget_template = PromptTemplate.from_template("""
You are a financial planning assistant.

Given the following department-level budget data:

{budget_table}

Tasks:
1. Identify departments that overspent or underspent.
2. Suggest budget reallocation for next quarter.
3. Recommend cost-saving strategies.

Present your suggestions in bullet points.
""")
```

✅ Step 4: Budget analysis logic using GPT (`budget_agent.py`)

```
from budget_data import load_budget_data
from budget_prompt import budget_template
from langchain.chat_models import ChatOpenAI

def analyze_budget():
    df = load_budget_data()
    table = df.to_string(index=False)

    llm = ChatOpenAI(temperature=0.3)
    prompt = budget_template.format(budget_table=table)
    summary = llm.predict(prompt)

    return df, summary
```

✅ Step 5: Build Streamlit app (`app.py`)

```
import streamlit as st
import matplotlib.pyplot as plt
```

```python
from budget_agent import analyze_budget

st.title("📊 Budget Optimization Agent")

if st.button("Analyze Budget"):
    df, result = analyze_budget()

    st.subheader("💸 Budget Overview")
    st.dataframe(df)

    st.subheader("📈 Budget Performance Chart")
    fig, ax = plt.subplots()
    ax.bar(df["Department"], df["Allocated Budget"], label="Budget",
        alpha=0.6)
    ax.bar(df["Department"], df["Actual Spending"], label="Spent",
        alpha=0.6)
    ax.set_ylabel("USD")
    ax.legend()
    st.pyplot(fig)

    st.subheader("🧠 AI Recommendations")
    st.write(result)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output:

```
Overspending detected in Marketing (+$15,000), IT (+$10,000), and
Operations (+$15,000).
Underspending in HR (only $30,000 of $40,000) and Engineering ($10,000
below budget).

Recommendations:
- Reallocate $10,000 from Engineering to Marketing and IT.
- Investigate cost drivers in Operations; consider contract
renegotiation.
- Optimize HR headcount planning to absorb more budget.
- Encourage zero-based budgeting across departments.
```

📖 Agent #90: Compliance Knowledge Assistant

📝 Overview

The Compliance Knowledge Assistant serves as an on-demand advisor for regulatory and internal compliance queries. It helps employees, auditors, and legal teams instantly retrieve and understand compliance rules, definitions, and protocols across frameworks like GDPR, SOX, HIPAA, or internal policies. In this lab, you'll build a chatbot that answers compliance-related questions by searching through a regulation document database using GPT.

🧪 Lab Objectives

By the end of this lab, you will:

- Load compliance documents (GDPR, HIPAA, etc.)
- Embed the documents for fast semantic search
- Ask GPT natural language questions about policy details
- Retrieve answers with citations and source snippets

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- **FAISS for search**
- *(Optional: Document upload support for internal policies)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir compliance_knowledge_agent
cd compliance_knowledge_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu
```

✅ Step 2: Sample Regulation File (`docs/gdpr_excerpt.txt`)

```
Article 5(1)(b): Personal data shall be collected for specified,
explicit and legitimate purposes and not further processed in a manner
that is incompatible with those purposes.
```

```
...
Article 15: The data subject shall have the right to obtain from the
controller confirmation as to whether or not personal data concerning
them is being processed...
```

✅ Step 3: Embed the Documents (`embed_docs.py`)

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
import os

docs = []
for file in os.listdir("docs"):
    if file.endswith(".txt"):
        loader = TextLoader(f"docs/{file}")
        docs.extend(loader.load())

splitter = CharacterTextSplitter(chunk_size=600, chunk_overlap=100)
chunks = splitter.split_documents(docs)

embeddings = OpenAIEmbeddings()
db = FAISS.from_documents(chunks, embeddings)
db.save_local("compliance_index")
```

Run:

```
python embed_docs.py
```

✅ Step 4: GPT Answer Retrieval (`qa_agent.py`)

```python
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def ask_compliance_question(question):
    db = FAISS.load_local("compliance_index", OpenAIEmbeddings())
    retriever = db.as_retriever(search_type="similarity", k=3)
    llm = ChatOpenAI(temperature=0.2)
    chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
    return chain.run(question)
```

✅ Step 5: Streamlit UI (`app.py`)

```
import streamlit as st
from qa_agent import ask_compliance_question

st.title("📖 Compliance Knowledge Assistant")
st.caption("Ask your internal compliance or regulatory policy
        questions")

question = st.text_input("Enter your question (e.g., What are GDPR
        consent requirements?)")

if question:
    answer = ask_compliance_question(question)
    st.subheader("🧠 Answer")
    st.text_area("GPT Compliance Explanation", answer, height=400)
```

Run:

```
streamlit run app.py
```

🧪 Example Use Case

User Input: "What does GDPR say about the right to be informed?"

GPT Output (Excerpt):

```
Under Article 13 and 14 of GDPR, data subjects must be informed of the
identity of the data controller, the purposes of processing, and their
rights. This information must be provided at the time of data
collection or within a reasonable period.

Relevant section: Article 13 — Information to be provided where
personal data are collected from the data subject.
```

🗒️ Agent #91: AI Meeting Notes Agent

📝 Overview

The AI Meeting Notes Agent automatically transcribes, summarizes, and organizes meeting conversations into structured action items, decisions, and key points. It improves productivity by eliminating manual note-taking and enabling better follow-through. In this lab, you'll build a web app that accepts audio recordings, uses speech-to-text and GPT to summarize meetings into bullet points.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a meeting audio file (.mp3/.wav)
- Transcribe it using OpenAI Whisper API (or any STT engine)
- Use GPT to generate summaries with action items
- Export/share the notes in Markdown or plain text

🧰 Tech Stack

- **Python**
- **Streamlit**
- **OpenAI Whisper (for STT)**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: AssemblyAI/Deepgram/Google STT alternative)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_meeting_notes
cd ai_meeting_notes
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain pydub
```

✅ Step 2: Upload Audio + Transcription (`transcribe_audio.py`)

```
import openai
from pydub import AudioSegment
import os

def transcribe_audio(file_path):
    audio = AudioSegment.from_file(file_path)
    audio.export("temp.wav", format="wav")

    with open("temp.wav", "rb") as audio_file:
        transcript = openai.Audio.transcribe("whisper-1", audio_file)
    os.remove("temp.wav")
    return transcript["text"]
```

✅ Step 3: GPT Meeting Summarizer (`summarizer.py`)

```python
from langchain.chat_models import ChatOpenAI

def summarize_meeting(transcript):
    llm = ChatOpenAI(temperature=0.3)
    prompt = f"""
    Summarize the following meeting transcript into:
    1. Key Discussion Points
    2. Decisions Made
    3. Action Items with Owners
    4. Next Steps

    Transcript:
    {transcript}
    """
    return llm.predict(prompt)
```

✅ Step 4: Streamlit App (`app.py`)

```python
import streamlit as st
from transcribe_audio import transcribe_audio
from summarizer import summarize_meeting

st.title("📝 AI Meeting Notes Agent")
st.caption("Turn your meeting audio into actionable notes")

audio_file = st.file_uploader("Upload your meeting recording
        (.mp3/.wav)", type=["mp3", "wav"])

if audio_file:
    with open("uploaded_audio", "wb") as f:
        f.write(audio_file.read())
    st.info("Transcribing audio...")
    transcript = transcribe_audio("uploaded_audio")
    st.success("Transcription complete!")
    st.text_area("📝 Transcript", transcript, height=200)

    st.info("Summarizing meeting...")
    summary = summarize_meeting(transcript)
    st.subheader("📋 AI-Generated Meeting Notes")
    st.text_area("Meeting Summary", summary, height=500)
    st.download_button("Download Notes", data=summary,
        file_name="meeting_notes.md")
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Transcript (excerpt): “Let’s launch the new feature by Friday. Alex will handle QA. Priya will update the landing page.”

GPT Output:

```
1. Key Discussion Points:
   - Launch timeline for new feature
   - Marketing prep for landing page update
   - QA assignment and readiness

2. Decisions Made:
   - Go-live date confirmed: Friday

3. Action Items:
   - Alex: Complete QA by Thursday
   - Priya: Finalize and publish landing page

4. Next Steps:
   - Review KPIs post-launch next Monday
```

🔎 Agent #92: AI-powered Search Agent

📝 Overview

The AI-powered Search Agent enhances traditional search functionality by offering context-aware, natural language query handling across enterprise documents, emails, reports, and wikis. Instead of keyword matching, it uses semantic search to understand intent and retrieve the most relevant answers. In this lab, you’ll build an agent that indexes internal documents and answers user questions using GPT and vector search.

🧪 Lab Objectives

By the end of this lab, you will:

- Load and embed company documents (PDFs, text files, or markdown)
- Use a vector database for fast retrieval
- Ask natural language queries and get GPT-generated answers
- Provide source snippets to build trust and traceability

🧰 Tech Stack

- **Python**
- **Streamlit**

- **LangChain + GPT-4 or GPT-3.5**
- **FAISS (or Chroma) for document embeddings**
- *(Optional: PDF parser, file upload, multi-language support)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir ai_search_agent
cd ai_search_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai faiss-cpu tiktoken PyPDF2
```

✅ Step 2: Sample Document Folder (`docs/`)

Add company documents (e.g., policies, procedures, FAQs) in `.txt` or `.pdf`.

✅ Step 3: Document Loader and Embedder (`embed_docs.py`)

```
from langchain.document_loaders import PyPDFLoader, TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
import os

documents = []
for file in os.listdir("docs"):
    if file.endswith(".pdf"):
        documents.extend(PyPDFLoader(f"docs/{file}").load())
    elif file.endswith(".txt"):
        documents.extend(TextLoader(f"docs/{file}").load())

splitter = CharacterTextSplitter(chunk_size=700, chunk_overlap=100)
chunks = splitter.split_documents(documents)

embedding_model = OpenAIEmbeddings()
db = FAISS.from_documents(chunks, embedding_model)
db.save_local("search_index")
```

Run:

```
python embed_docs.py
```

✅ Step 4: GPT Search Logic (`search_agent.py`)

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

def query_documents(query):
    db = FAISS.load_local("search_index", OpenAIEmbeddings())
    retriever = db.as_retriever(search_type="similarity", k=3)
    llm = ChatOpenAI(temperature=0.2)
    qa_chain = RetrievalQA.from_chain_type(llm=llm,
        retriever=retriever)
    return qa_chain.run(query)
```

✅ Step 5: Streamlit Interface (`app.py`)

```
import streamlit as st
from search_agent import query_documents

st.title("🔎 AI-powered Enterprise Search Agent")
st.caption("Ask questions across internal files, policies, or wikis")

query = st.text_input("Enter your question (e.g., What is the PTO
        policy?)")

if query:
    result = query_documents(query)
    st.subheader("📄 AI-Generated Answer")
    st.text_area("Search Response", result, height=500)
```

Run:

```
streamlit run app.py
```

🧪 Example Use Case

User Input: “What are the rules around remote work reimbursement?”

GPT Output (Excerpt):

```
According to the policy dated May 2024, employees working remotely are
eligible for up to $200/month in home office reimbursement, covering
internet, ergonomic chairs, and lighting. Claims must be submitted by
the 5th of the following month with receipts.

Source: RemoteWorkPolicy2024.pdf, page 3
```

📊 Agent #93: Automated Reporting Agent

📝 Overview

The Automated Reporting Agent streamlines business reporting by generating regular summaries, metrics, and visualizations from raw data sources. It eliminates manual report creation by producing natural language summaries and charts that update on-demand or on a schedule. In this lab, you'll build an agent that connects to a CSV/Excel dataset and generates a weekly performance report using GPT and matplotlib.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a business dataset (sales, KPIs, ops, etc.)
- Extract and summarize trends using GPT
- Generate visualizations for key metrics
- Export a report in Markdown or PDF format

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Matplotlib/Plotly**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Scheduled automation with Airflow/cron jobs)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir reporting_agent
cd reporting_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas matplotlib openai langchain
```

✅ Step 2: Sample Dataset (`data/weekly_sales.csv`)

```
Date,Region,Sales,New_Customers,Churned_Customers
2025-07-01,North,12000,45,3
2025-07-01,South,9500,35,4
```

```
2025-07-08,North,13500,50,2
2025-07-08,South,11000,38,5
```

✅ Step 3: Reporting Logic (`generate_report.py`)

```python
import pandas as pd
import matplotlib.pyplot as plt
from langchain.chat_models import ChatOpenAI

def generate_summary(df):
    stats = df.describe().to_string()
    llm = ChatOpenAI(temperature=0.3)
    prompt = f"""
    Given the following business stats table, summarize weekly
        performance.
    Identify trends, anomalies, and noteworthy events in 150 words.

    {stats}
    """
    return llm.predict(prompt)

def plot_sales(df):
    grouped = df.groupby('Date')['Sales'].sum().reset_index()
    plt.figure(figsize=(8,4))
    plt.plot(grouped['Date'], grouped['Sales'], marker='o')
    plt.title("Weekly Sales")
    plt.xlabel("Date")
    plt.ylabel("Sales")
    plt.tight_layout()
    plt.savefig("sales_plot.png")
    return "sales_plot.png"
```

✅ Step 4: Streamlit Dashboard (`app.py`)

```python
import streamlit as st
import pandas as pd
from generate_report import generate_summary, plot_sales

st.title("📊 Automated Reporting Agent")
st.caption("Upload business data and generate instant weekly
        summaries")

uploaded_file = st.file_uploader("Upload CSV Dataset", type=["csv"])
if uploaded_file:
    df = pd.read_csv(uploaded_file)
    st.dataframe(df)

    if st.button("Generate Report"):
```

```
        st.info("Creating summary...")
        summary = generate_summary(df)
        chart = plot_sales(df)

        st.subheader("📝 GPT-Generated Report Summary")
        st.text_area("Report", summary, height=300)

        st.subheader("📈 Sales Visualization")
        st.image(chart)

        st.download_button("Download Summary", data=summary,
        file_name="weekly_report.md")
```

Run:

```
streamlit run app.py
```

🧪 Example Output

GPT Summary:

> "North region saw a 12.5% increase in weekly sales, with a notable decrease in churned customers (from 3 to 2). South region sales improved 15%, although customer churn slightly rose. Overall growth trends suggest strong campaign impact in Q3 kickoff."

Sales Chart: Line graph of total sales per week

📈 Agent #94: KPI Monitoring Agent

📝 Overview

The KPI Monitoring Agent tracks and evaluates key performance indicators in real-time, detecting deviations and surfacing actionable insights. It supports decision-making by keeping business units informed on progress against targets. In this lab, you'll build a dashboard that reads KPI data, compares it against thresholds, and uses GPT to generate observations and alerts.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a KPI dataset or define KPIs manually
- Compare each KPI against defined thresholds
- Generate GPT-based analysis of trends, warnings, and recommendations
- Visualize KPI trends and status using Streamlit charts

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas + Plotly or Matplotlib**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Add real-time database integration or Slack alerts)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir kpi_monitoring_agent
cd kpi_monitoring_agent
python -m venv venv
source venv/bin/activate
pip install streamlit pandas matplotlib openai langchain
```

✅ Step 2: Sample KPI Dataset (`data/kpi_data.csv`)

```
Date,Metric,Value,Target
2025-07-01,Website Traffic,12000,10000
2025-07-01,Conversion Rate,2.5,3.0
2025-07-01,Customer Satisfaction,4.2,4.5
2025-07-08,Website Traffic,13500,10000
2025-07-08,Conversion Rate,2.9,3.0
2025-07-08,Customer Satisfaction,4.6,4.5
```

✅ Step 3: KPI Analyzer (`analyze_kpis.py`)

```
import pandas as pd
from langchain.chat_models import ChatOpenAI

def analyze_kpis(df):
    kpi_summary = ""
    for metric in df["Metric"].unique():
        recent = df[df["Metric"] ==
        metric].sort_values("Date").tail(2)
        if len(recent) == 2:
            change = recent.iloc[1]["Value"] - recent.iloc[0]["Value"]
            kpi_summary += f"{metric}: changed by {change:.2f} (from
        {recent.iloc[0]['Value']} to {recent.iloc[1]['Value']})\n"

    prompt = f"""
```

```
    Given the following KPI changes, analyze overall business
        performance.
    Point out improvements, risks, and recommended actions.

    {kpi_summary}
    """
    llm = ChatOpenAI(temperature=0.2)
    return llm.predict(prompt)
```

✅ Step 4: Streamlit Dashboard (`app.py`)

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from analyze_kpis import analyze_kpis

st.title("📈 KPI Monitoring Agent")
st.caption("Track KPI performance and receive AI-generated insights")

file = st.file_uploader("Upload KPI Data (CSV)", type=["csv"])
if file:
    df = pd.read_csv(file)
    st.dataframe(df)

    metric = st.selectbox("Choose Metric", df["Metric"].unique())
    filtered = df[df["Metric"] == metric]

    # Plot
    fig, ax = plt.subplots()
    ax.plot(filtered["Date"], filtered["Value"], label="Actual",
        marker="o")
    ax.plot(filtered["Date"], filtered["Target"], label="Target",
        linestyle="--", marker="x")
    ax.set_title(f"{metric} Over Time")
    ax.legend()
    st.pyplot(fig)

    if st.button("Analyze KPIs"):
        summary = analyze_kpis(df)
        st.subheader("🧠 GPT-Generated KPI Summary")
        st.text_area("Insights", summary, height=300)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

GPT Summary (Excerpt):

> “Website Traffic is exceeding target, indicating successful engagement. Conversion Rate is improving but remains slightly below target—consider UX optimizations. Customer Satisfaction rose to 4.6, surpassing target. Overall trend is positive with minor conversion concerns.”

⚙️ Agent #95: Workflow Optimization Agent

📝 Overview

The Workflow Optimization Agent analyzes business workflows and suggests efficiency improvements using AI. By identifying repetitive tasks, bottlenecks, and manual dependencies, this agent can recommend automations, re-sequencing steps, or resource reallocations. In this lab, you'll build an agent that visualizes a given workflow, accepts pain points, and uses GPT to recommend optimizations.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload or define a business process/workflow
- Tag bottlenecks, delays, or manual steps
- Use GPT to suggest optimization strategies
- Visualize the improved workflow for clarity

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Graphviz or Diagrams for process flow**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integration with BPMN/XML process files)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir workflow_optimization_agent
cd workflow_optimization_agent
python -m venv venv
source venv/bin/activate
pip install streamlit langchain graphviz openai
```

✅ Step 2: Sample Workflow Input (Manual JSON)

```
[
  {"step": "Receive Customer Request", "type": "manual", "bottleneck":
        false},
  {"step": "Verify Customer Info", "type": "manual", "bottleneck":
        true},
  {"step": "Assign Support Agent", "type": "semi-automated",
        "bottleneck": false},
  {"step": "Respond to Request", "type": "manual", "bottleneck":
        true},
  {"step": "Log Request", "type": "automated", "bottleneck": false}
]
```

✅ Step 3: Optimization Logic (`optimize_workflow.py`)

```
from langchain.chat_models import ChatOpenAI

def suggest_optimizations(steps):
    summary = "\n".join([f"{s['step']} ({s['type']}) - Bottleneck:
        {s['bottleneck']}" for s in steps])
    prompt = f"""
    Analyze the following workflow. Suggest improvements to reduce
        manual steps and remove bottlenecks.
    Return a list of recommendations.

    Workflow:
    {summary}
    """
    llm = ChatOpenAI(temperature=0.3)
    return llm.predict(prompt)
```

✅ Step 4: Visualization Helper (`draw_workflow.py`)

```
from graphviz import Digraph

def draw_workflow(steps):
    dot = Digraph()
    for i, s in enumerate(steps):
        style = "filled" if s["bottleneck"] else "solid"
        color = "red" if s["bottleneck"] else "lightblue"
        dot.node(str(i), s["step"], style=style, fillcolor=color)
        if i > 0:
            dot.edge(str(i-1), str(i))
    dot.render("workflow_graph", format="png", cleanup=True)
    return "workflow_graph.png"
```

✅ Step 5: Streamlit Interface (`app.py`)

```python
import streamlit as st
import json
from optimize_workflow import suggest_optimizations
from draw_workflow import draw_workflow

st.title("⚙️ Workflow Optimization Agent")
st.caption("Visualize and optimize your business process")

raw = st.text_area("Paste Workflow JSON", height=300)
if st.button("Optimize Workflow") and raw:
    try:
        steps = json.loads(raw)
        recs = suggest_optimizations(steps)
        image_path = draw_workflow(steps)

        st.subheader("🧠 AI-Recommended Improvements")
        st.text_area("Suggestions", recs, height=250)

        st.subheader("📊 Current Workflow Visualization")
        st.image(image_path)
    except Exception as e:
        st.error(f"Error parsing workflow: {e}")
```

Run:

```
streamlit run app.py
```

🧪 Example Output

Input: Workflow with manual verification and manual response tagged as bottlenecks.

GPT Recommendations (excerpt):

```
- Automate customer info verification with ID/document OCR
- Implement AI chatbot for initial responses
- Introduce triage automation for routing requests
- Log all tickets automatically via CRM integration
```

Visualization: A flowchart with red-highlighted bottlenecks and arrows showing task sequence

---

👨‍💼 Agent #96: Virtual Executive Assistant

📝 Overview

The Virtual Executive Assistant Agent helps leaders manage their calendar, schedule meetings, summarize documents, draft emails, and track action items. This agent blends time management, communication support, and organizational memory using natural language interfaces. In this lab, you'll build a prototype that performs three core functions: scheduling support, email drafting, and task summarization.

🧪 Lab Objectives

By the end of this lab, you will:

- Accept natural language requests from a busy executive
- Use GPT to interpret intent and generate responses
- Handle scheduling prompts, email writing, and summary generation
- Simulate assistant behavior via a unified UI

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate with Google Calendar API, Gmail API, Notion API)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir virtual_exec_assistant
cd virtual_exec_assistant
python -m venv venv
source venv/bin/activate
pip install streamlit langchain openai
```

✅ Step 2: Prompt Logic (`assistant_tools.py`)

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0.3)

def handle_scheduling(request):
```

```
    prompt = f"""
    You're a virtual executive assistant. A user said:
    "{request}"
    Reply with a scheduling suggestion, confirming time zones,
        conflicts, and offering next best options.
    """
    return llm.predict(prompt)

def handle_email_drafting(request):
    prompt = f"""
    You're drafting an email based on this request:
    "{request}"
    Write a professional email in 150 words or less.
    """
    return llm.predict(prompt)

def handle_summary(text):
    prompt = f"""
    Summarize the following content in bullet points with action
        items:
    {text}
    """
    return llm.predict(prompt)
```

✅ Step 3: Streamlit UI (`app.py`)

```
import streamlit as st
from assistant_tools import handle_scheduling, handle_email_drafting,
        handle_summary

st.title("🧑‍💼 Virtual Executive Assistant Agent")

tab1, tab2, tab3 = st.tabs(["🗓️ Schedule Assistant", "✉️ Email
        Writer", "📋 Summarize Notes"])

with tab1:
    st.subheader("Smart Scheduling Support")
    req1 = st.text_area("Describe the meeting or schedule issue:")
    if st.button("Get Scheduling Suggestion"):
        st.text_area("Assistant Reply", handle_scheduling(req1),
        height=200)

with tab2:
    st.subheader("AI-Powered Email Composer")
    req2 = st.text_area("Describe the email you want drafted:")
    if st.button("Generate Email"):
        st.text_area("Drafted Email", handle_email_drafting(req2),
        height=200)

with tab3:
    st.subheader("Meeting / Document Summary")
```

```
    req3 = st.text_area("Paste notes or content to summarize:")
    if st.button("Summarize"):
        st.text_area("AI Summary", handle_summary(req3), height=250)
```

Run:

```
streamlit run app.py
```

🧪 Example Use Cases

✅ Scheduling Prompt:

Input: “Reschedule my Thursday 3PM call with the CFO to next week, and add a follow-up reminder.”

GPT Output: “Rescheduling the CFO call to next Tuesday at 3PM EST. Adding a follow-up reminder for the prior day. Let me know if you’d like to send a calendar invite.”

✅ Email Writing Prompt:

Input: “Send an apology email to the client about the delay and promise revised delivery.”

GPT Output: “Dear [Client Name], I sincerely apologize for the delay in our recent deliverable. We’ve identified the issue and are committed to delivering the revised version by [Date]. Thank you for your patience.”

✅ Summarization Prompt:

Input: Notes from a leadership meeting.

GPT Output (bullet points):

- Launch moved to Sept 15
- Dev team needs final specs by Friday
- Schedule a press release draft by Aug 20
- Review product roadmap in next exec sync

🧠 Agent #97: Business Strategy Advisor

📝 Overview

The Business Strategy Advisor Agent supports executives and product leaders by analyzing business data, interpreting market trends, and generating strategic insights. This agent helps simulate competitive analysis, SWOT assessments, and AI-assisted OKR framing. In this lab, you'll build an agent that accepts key business inputs and delivers GPT-generated strategic guidance with optional visual aids.

🧪 Lab Objectives

By the end of this lab, you will:

- Input internal goals and external context
- Use GPT to generate strategic recommendations (e.g., growth, market entry, cost-cutting)
- Simulate SWOT and OKR summaries
- Present results in a clear, executive-style format

🧰 Tech Stack

- **Python**
- **Streamlit**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Integrate with financial APIs, competitor datasets, or BI tools)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir strategy_advisor_agent
cd strategy_advisor_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai langchain
```

✅ Step 2: Prompt Engine (`strategy_engine.py`)

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0.3)
```

```
def generate_strategy(goals, threats, market_trends):
    prompt = f"""
    You are a Business Strategy Advisor AI.
    Based on the following input, provide a brief strategy plan:

    Company Goals: {goals}
    External Threats: {threats}
    Market Trends: {market_trends}

    Output a strategy in 3 sections:
    1. Strategic Priorities
    2. SWOT Summary (bullets)
    3. Proposed OKRs (3 objectives with key results)
    """
    return llm.predict(prompt)
```

✅ Step 3: Streamlit App (`app.py`)

```
import streamlit as st
from strategy_engine import generate_strategy

st.title("🧠 Business Strategy Advisor Agent")
st.caption("Input company goals and context. Get GPT-powered strategy
        suggestions.")

with st.form("strategy_form"):
    goals = st.text_area("🎯 Company Goals", placeholder="e.g., Expand
        into EMEA, reduce churn by 20%, improve NPS")
    threats = st.text_area("⚠️ External Risks / Threats",
        placeholder="e.g., Competitor X release, economic downturn")
    trends = st.text_area("📈 Market Trends", placeholder="e.g., Surge
        in AI adoption, remote work")

    submitted = st.form_submit_button("Generate Strategy")
    if submitted:
        output = generate_strategy(goals, threats, trends)
        st.subheader("📊 Strategy Report")
        st.text_area("GPT Output", output, height=400)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output

Input:

- Goals: Enter APAC market, reduce operating cost by 15%
- Threats: Declining retention, inflation

- Trends: AI-driven automation, hybrid work

GPT Strategy Output (Excerpt):

1. Strategic Priorities

- Enter APAC via low-cost SaaS tier
- Implement automation in support and finance
- Redesign retention funnel via personalization

2. SWOT Summary

- Strengths: Scalable tech, agile team
- Weaknesses: High CAC
- Opportunities: AI expansion, Asia-Pacific demand
- Threats: Retention risks, market saturation

3. OKRs

- Objective: Expand APAC footprint
  - KR1: Localize platform in 3 languages
  - KR2: Acquire 5 regional partners
- Objective: Reduce operational expenses
  - KR1: Automate 3 back-office workflows
  - KR2: Reduce monthly SaaS cost by 10%

🛍️ Agent #98: Personalized Product Recommendation Agent

📝 Overview

The Personalized Product Recommendation Agent delivers intelligent, real-time suggestions based on user preferences, behavior, and purchase history. Powered by AI and recommendation algorithms, this agent can increase customer satisfaction and sales conversion. In this lab, you'll build a prototype agent that accepts user input and delivers tailored product recommendations using embeddings and GPT.

🧪 Lab Objectives

By the end of this lab, you will:

- Load a sample product catalog
- Simulate user preferences or profile
- Generate recommendations using cosine similarity or GPT
- Display recommendations in a clean UI

🧰 Tech Stack

- **Python**
- **Streamlit**
- **Pandas**
- **OpenAI Embeddings (text-embedding-ada-002)**
- *(Optional: Integrate user click history or collaborative filtering)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir product_recommendation_agent
cd product_recommendation_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai pandas scikit-learn
```

✅ Step 2: Sample Product Catalog (`products.csv`)

```
product_id,title,description
101,Wireless Headphones,Noise-canceling over-ear Bluetooth headphones with 40hr battery life
102,Smart Fitness Watch,Track heart rate, steps, sleep and get health insights
103,Espresso Machine,Brew barista-grade coffee at home with touch controls
104,Laptop Stand,Ergonomic aluminum stand for MacBook and notebooks
105,4K Monitor,Ultra-HD display with wide color gamut for creators
```

✅ Step 3: Embedding & Recommendation Logic (`recommend.py`)

```
import pandas as pd
import openai
from sklearn.metrics.pairwise import cosine_similarity

openai.api_key = "YOUR_API_KEY"

def get_embedding(text):
    response = openai.Embedding.create(
        input=[text], model="text-embedding-ada-002"
    )
    return response["data"][0]["embedding"]

def recommend_products(user_input, products_df):
```

```
    user_vec = get_embedding(user_input)
    product_vecs = [get_embedding(desc) for desc in
        products_df["description"]]

    scores = cosine_similarity([user_vec], product_vecs)[0]
    products_df["score"] = scores
    return products_df.sort_values(by="score",
        ascending=False).head(3)
```

✅ Step 4: Streamlit App (`app.py`)

```
import streamlit as st
import pandas as pd
from recommend import recommend_products

st.title("🛍️ Personalized Product Recommendation Agent")

user_input = st.text_area("Tell us what you're looking for:",
                          placeholder="e.g., I need something for my
        home workouts")
if st.button("Get Recommendations"):
    df = pd.read_csv("products.csv")
    top_products = recommend_products(user_input, df)

    st.subheader("🎯 Top Recommendations:")
    for _, row in top_products.iterrows():
        st.markdown(f"**{row['title']}**")
        st.markdown(f"{row['description']}")
        st.markdown("---")
```

Run:

```
streamlit run app.py
```

🧪 Example Output

User Input: “I want something to help me stay fit and track my workouts”

Top 3 Recommendations:

- **Smart Fitness Watch**: Track heart rate, steps, sleep and get health insights
- **Wireless Headphones**: Noise-canceling headphones for workout playlists
- **Laptop Stand**: Ergonomic setup for remote workouts and Zoom sessions

---

📄 Agent #99: Contract Review Automation Agent

📝 Overview

The Contract Review Automation Agent helps legal and procurement teams review lengthy contracts for risk, compliance, and obligations. It uses AI to parse, summarize, and flag key clauses (e.g., termination, payment, liability). In this lab, you'll build a prototype that allows users to upload a contract and receive AI-generated insights and clause highlights.

🧪 Lab Objectives

By the end of this lab, you will:

- Upload a legal contract (PDF or text)
- Extract key clauses (termination, indemnity, etc.)
- Generate GPT-powered summaries and risk flags
- Present findings in a clear, organized format

🧰 Tech Stack

- **Python**
- **Streamlit**
- **PyPDF2 or pdfplumber**
- **LangChain + GPT-4 or GPT-3.5**
- *(Optional: Use regex or clause dictionaries for better tagging)*

🧭 Step-by-Step Instructions

✅ Step 1: Environment Setup

```
mkdir contract_review_agent
cd contract_review_agent
python -m venv venv
source venv/bin/activate
pip install streamlit openai pdfplumber langchain
```

✅ Step 2: PDF Extractor (`extractor.py`)

```
import pdfplumber

def extract_text_from_pdf(uploaded_file):
```

```
    with pdfplumber.open(uploaded_file) as pdf:
        return "\n".join([page.extract_text() for page in pdf.pages if
        page.extract_text()])
```

✅ Step 3: Clause Analyzer (`reviewer.py`)

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0.3)

def review_contract(text):
    prompt = f"""
    You are a legal AI agent. Review the following contract and
        extract:
    1. Key clauses (termination, liability, payment, indemnity)
    2. Any risky or unusual terms
    3. Suggestions to improve the contract or request clarification

    Contract:
    {text[:5000]}  # Limit to first 5000 chars for efficiency
    """
    return llm.predict(prompt)
```

✅ Step 4: Streamlit UI (`app.py`)

```
import streamlit as st
from extractor import extract_text_from_pdf
from reviewer import review_contract

st.title("📄 Contract Review Automation Agent")
st.caption("Upload and review contracts using GPT-powered legal
        insights")

uploaded_file = st.file_uploader("Upload Contract PDF", type=["pdf"])
if uploaded_file:
    with st.spinner("Extracting and analyzing..."):
        raw_text = extract_text_from_pdf(uploaded_file)
        st.subheader("📃 Extracted Contract Text")
        st.text_area("Contract Content (Preview)", raw_text[:1000],
        height=200)

        if st.button("Review Contract"):
            summary = review_contract(raw_text)
            st.subheader("🧠 AI Review Output")
            st.text_area("Insights", summary, height=400)
```

Run the app:

```
streamlit run app.py
```

🧪 Example Output (Excerpt)

Input Contract (Partial): Contains clauses for payment due within 30 days, unilateral termination, limitation of liability.

GPT Review Output:

- **Key Clauses Identified**:
  - Termination: Allows unilateral termination by Provider
  - Payment: Net 30 days from invoice
  - Liability: Limited to service fees paid
  - Indemnity: Customer holds provider harmless
- **Risk Flags**:
  - Unilateral termination may create power imbalance
  - Indemnity clause is one-sided
- **Recommendations**:
  - Negotiate mutual indemnity
  - Add early termination notice period
