CST1510 COURSEWORK 2

Week 10: AI Integration with ChatGPT API

Multi-Domain Intelligence Platform

Week 9: Web Interface & Visualization

Instructor Teaching Guide

Building Intelligent Assistants

01 // What Are APIs?

Definition

API = Application Programming Interface

A set of rules and protocols that allows different software applications to communicate with each other.

Why APIs Matter

- Access powerful services without building from scratch
- Focus on your core application, not reinventing the wheel
- Always up-to-date with latest improvements

Real-World Examples

Weather APIs | Google Maps | Payment APIs | Social Media | AI Services

The Restaurant Analogy

You (Customer)
Your Python Code

↓

Menu
API Documentation

↓

Waiter
The API

↓

Kitchen
API Server (ChatGPT)

You don't need to know how to cook—just how to order!

02 // Why Use AI APIs in Our Project?

Enhance User Experience

Transform from passive storage to active intelligence. AI understands, analyzes, and provides recommendations.

Automate Analysis

AI can assess severity, suggest methods, categorize tickets, and recommend solutions automatically.

Domain-Specific Experts

Three specialized assistants customized for Cybersecurity, Data Science, and IT Operations.

No Training Required

Access state-of-the-art AI for cents per request. No millions in compute costs or months of training.

Domain-Specific AI Capabilities

Cybersecurity

- Assess threat severity
- Identify attack vectors
- Recommend mitigation
- Explain security concepts

Data Science

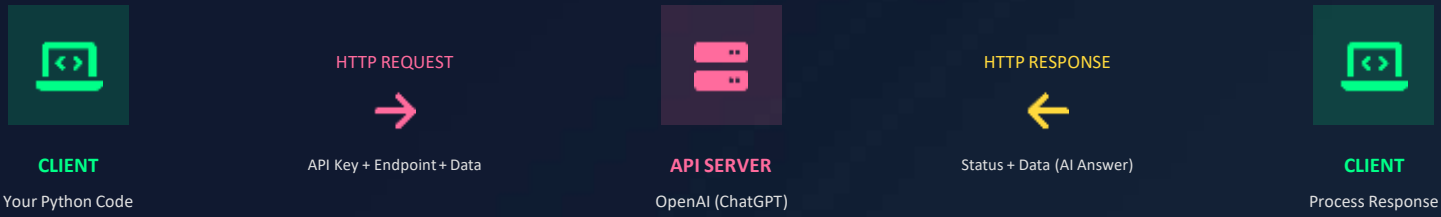
- Suggest analysis methods
- Recommend visualizations
- Explain statistical tests
- Data cleaning strategies

IT Operations

- Triage support tickets
- Troubleshooting steps
- System recommendations
- Explain technical issues

03 // How APIs Work

Client-Server Model: HTTP Request → Response Cycle



Four Key Components

1. API Key (Authentication)

Like a password proving you're authorized. Tracks usage for billing and prevents abuse. **Keep it secret!**

2. Endpoint (URL)

The specific URL where you send requests. Different endpoints for different services (chat, images, audio).

3. Request (Your Data)

What you're sending: model name, messages array, parameters. Usually a Python dictionary converted to JSON.

4. Response (API's Data)

What the API sends back: AI's generated text, metadata (token usage), status codes. Extract relevant data in your code.

03 // ChatGPT API Overview

Provider: OpenAI

Industry-leading AI research company

Available Models

Model	Quality	Speed	Cost
GPT-4o	★★★★★	Fast	Higher
GPT-3.5-turbo	★★★★	Very Fast	Lower

Capabilities

- **Text Generation:** Create human-like responses
- **Conversation:** Multi-turn contextual dialogue
- **Analysis:** Understand and analyze complex information
- **Customization:** System prompts for domain expertise

Requirements

API Key
From platform.openai.com

Credits
$5 minimum purchase

Internet
API calls over HTTPS

Python Library
pip install openai

Pricing Structure

Pay-per-use model

Charged by tokens (words)

Input + Output tokens counted

Monitor usage in OpenAI dashboard

06 // Week 10 Learning Objectives

Understand API Fundamentals

Learn what APIs are, why they matter, and how they work (client-server model, HTTP requests/responses)

Integrate ChatGPT API with Python

Install OpenAI library, configure API keys securely, and make your first API call

Build Text-Based AI Chat

Create interactive console chat with conversation history and understand message structure

Create Streamlit AI Chat Interface

Build beautiful web-based chat with streaming responses, session state, and clear chat functionality

Connect AI to Your Project

Integrate AI with Week 8 database and create domain-specific assistants for Cybersecurity, Data Science, and IT Operations

07 // Setup: Get Your OpenAI API Key

1 **Go to OpenAI Platform**

Visit **platform.openai.com** in your web browser

2 **Sign Up or Log In**

Create an account or sign in with your existing OpenAI credentials

3 **Navigate to API Keys**

Click on your profile → "API keys" section in the dashboard

4 **Create New Secret Key**

Click **"Create new secret key"** button

5 **Copy and Save Your Key**

Copy the key immediately—you won't be able to see it again!

Security Warning

- Never share your API key
- Never commit to GitHub
- Never hardcode in your code
- Store in **.env** or **secrets.toml**

Your API Key Format

```
sk-proj-XXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
```

Starts with **sk-proj-** or **sk-**

Pro Tip

Save your key in a password manager or secure note immediately after creation.

08 // Setup: Purchase Credits

1 Navigate to Billing Section

In your OpenAI dashboard, click on **Settings** → **Billing**

2 Add Payment Method

Click **"Add payment method"** and enter your credit/debit card details

3 Purchase Credits

Click **"Add to credit balance"** and enter amount (minimum $5)

4 Verify Credit Balance

Check that your credit balance appears in the billing dashboard

Pricing Information

Minimum: $5

Recommended: $10-20 for development

GPT-4o: ~$0.01 per request

GPT-3.5-turbo: ~$0.001 per request

Important Notes

- Credits don't expire
- Monitor usage in dashboard
- Set spending limits if needed
- $5-10 goes a long way for testing

Monitor Your Usage

View usage statistics in:

Settings → Usage

09 // Setup: Install Required Libraries

Install OpenAI Python Library

```
# Standard installation command
$pip install openai
```

Alternative Installation Methods

```
# If pip doesn't work, try pip3
$pip3 install openai

# Or use python -m pip
$python -m pip install openai
```

Upgrade to Latest Version

```
$pip install --upgrade openai
```

Verify Installation

1. Check Version

```
$pip show openai
Name: openai
Version: 1.x.x
```

2. Test Import

```
$python
>>>from openai import OpenAI
>>>print("Success!")
```

Troubleshooting

- Permission denied? Try **pip install --user openai**
- Multiple Python versions? Use **python3 -m pip**
- Virtual environment recommended for projects

10 // Part 1: Text-Based Python Implementation

Why Start with Text-Based?

Focus on Core Concepts

Learn API fundamentals without UI complexity. Understand requests, responses, and message structure first.

Easier Debugging

See exactly what's happening in the terminal. Print statements show API responses clearly.

Quick Iteration

Test API calls instantly. No need to reload web pages or manage session state yet.

Build Strong Foundation

Once you understand the basics, adding Streamlit UI becomes straightforward.

Learning Progression

1. **First API Call**
   Simple question → answer
2. **Message Structure**
   System, user, assistant roles
3. **Console Chat Loop**
   Interactive terminal chat
4. **Conversation History**
   Add memory to chat
5. **Secure API Keys**
   Use .env files safely

Ready for Streamlit Integration!

11 // Your First ChatGPT API Call

first_api_call.py

```
# 1. Import and initialize client
from openai import OpenAI
client = OpenAI(api_key="your-api-key-here")

# 2. Create the request
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "What is an API?"}
    ]
)

# 3. Extract the response
answer = response.choices[0].message.content

# 4. Display the result
print(answer)

# Run: python first_api_call.py
```

Key Components

1 Import & Initialize

Import OpenAI library and create client with your API key

2 Model Selection

Choose model: **gpt-4o** (best) or **gpt-3.5-turbo** (faster/cheaper)

3 Messages Array

List of message dictionaries with role and content keys

4 Extract Response

Navigate response object to get AI's text: choices[0].message.content

Important

Replace **"your-api-key-here"** with your actual API key from OpenAI platform. Never commit this to GitHub!

Expected Output

"An API (Application Programming Interface) is a set of rules..."

12 // Understanding the Message Structure

Three Message Roles

SYSTEM

Sets AI behavior and personality. Always first message.

"You are a helpful cybersecurity expert."

USER

Your questions or prompts to the AI.

"What is a DDoS attack?"

ASSISTANT

AI's responses to user messages.

"A DDoS attack is when..."

Conversation Flow Example

Messages Array Structure

```
[
  { "role": "system", "content": "You are helpful." },
  { "role": "user", "content": "What is Python?" },
  { "role": "assistant", "content": "Python is a..." }
]
```

Key Concepts

1. Dictionary Format
Each message is a Python dictionary with two keys: **"role"** and **"content"**.

2. Order Matters
Messages are sent in order: system first, then alternating user/assistant.

3. Conversation Context
Include previous messages so the AI remembers context and flow.

4. System Role Optional
The system message is optional but recommended to customize AI behavior.

13 // Interactive Console Chat

console_chat.py

```
from openai import OpenAI
client = OpenAI(api_key="your-api-key-here")

print("ChatGPT Console Chat")
print("Type 'quit' to exit\n")

while True:
    user_input = input("You: ")
    if user_input.lower() == "quit":
        print("Goodbye!")
        break
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": user_input}]
    )
    answer = response.choices[0].message.content
    print(f"AI: {answer}\n")
```

Run the script:

```
$ python console_chat.py
```

Chat Loop Flow

1. **Wait for User Input** — Display prompt "You: "
2. **Check Quit Condition** — If "quit" → exit loop
3. **Send to ChatGPT API** — Make API call with user input
4. **Display AI Response** — Print "AI: {answer}"

Key Features

- **Continuous conversation** until user quits
- **Simple interface** with terminal I/O
- **Graceful exit** with quit command
- **No memory yet** - each message is independent

14 // Adding Conversation History

The Problem: No Memory

Without conversation history, AI forgets previous messages:

You:
My name is Alice

You:
What's my name?

AI:
I don't have information about your name. ✗

Why This Happens

Each API call only sends the **current message**, not previous ones. The AI has no context from earlier in the conversation.

What We Need

Maintain a **messages list** that includes **all previous exchanges** (user + assistant messages) and send the entire history with each API call.

The Solution: Messages List

chat_with_history.py

```python
from openai import OpenAI

client = OpenAI(api_key="your-api-key-here")

# Initialize messages list with system message
messages = [
    {"role": "system", "content": "You are a helpful assistant."}
]

print("ChatGPT with Memory. Type 'quit' to exit.\n")

while True:
    user_input = input("You: ")

    if user_input.lower() == "quit":
        break# Add user message to history
    messages.append({"role": "user", "content": user_input})

    # Send ALL messages (entire conversation)
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=messages  # Send full history!
    )
```

Key Changes

1. Initialize `messages = []` **before** the loop • 2. `.append()` user message • 3. Send **entire messages list** to API • 4. `.append()` AI response

14 // Secure API Key Storage

WRONG WAY - Hardcoded Keys

```
from openai import OpenAI
# ✕ NEVER DO THIS!
client = OpenAI(api_key="sk-abc123xyz...")
```

Security Risks

- Exposed in source code
- Visible in Git history
- Anyone can steal and use it

RIGHT WAY - Use .env Files

1 Create .env file

```
# .envOPENAI_API_KEY=sk-abc123xyz...
```

2 Install python-dotenv & load key

```
$ pip install python-dotenv
```

```
from openai import OpenAI; from dotenv import load_dotenv; import os
load_dotenv(); client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

3 Add to .gitignore

```
.env
```

Why This Works

- API key stored in separate file
- .env file never committed to Git
- Safe to share code publicly

PRO TIP: Create a .env.example file with dummy values to show teammates what variables are needed

16 // Part 2: Streamlit Chat Interface

Part 1 Complete

Text-Based Foundation

- First API call
- Message structure
- Console chat loop
- Conversation history
- Secure API keys

Building on Week 9

Streamlit Foundations

Multi-Page Structure
Organize AI chat as a page in your app

Session State
Store chat history across page reloads

Authentication
Protect AI features with login system

UI Components
Sidebar, columns, buttons, metrics

Part 2 Goals

Web-Based Chat

- Streamlit secrets
- Chat UI elements
- Streaming responses
- Clear chat button
- Domain assistants

Same API concepts, beautiful web interface!

17 // Streamlit Secrets Management

Project Structure

```
your_project/
    ├── .streamlit/
    |   └── secrets.toml← API keys here
    ├── app.py
    └── .gitignore
```

1 Create .streamlit folder

```
# In your project root
$ mkdir .streamlit
```

2 Add to .gitignore

```
# .gitignore
.streamlit/secrets.toml
```

Important

The **.streamlit/** folder must be in your project root (same level as app.py)

secrets.toml Content

```
# .streamlit/secrets.toml
OPENAI_API_KEY = "sk-abc123xyz..."
API_URL = "https://api.example.com"
```

Accessing Secrets in Code

```
import streamlit as st
from openai import OpenAI
api_key = st.secrets["OPENAI_API_KEY"]
client = OpenAI(api_key=api_key)
```

Why Use Streamlit Secrets?

- **Built-in to Streamlit** - No extra libraries needed
- **Easy deployment** - Works with Streamlit Cloud
- **Secure** - Automatically excluded from Git

18 // Streamlit Chat Elements

st.chat_input()

User Input Field

Creates a text input field at the bottom of the app for users to type messages. Returns the user's message when they press Enter.

Using st.chat_input()

```
import streamlit as st

# Display input field
user_input = st.chat_input("Type your message...")

# Check if user entered something
if user_input:
    st.write(f"You said: {user_input}")
```

What It Looks Like

st.chat_message()

Chat Bubble Display

Displays messages in chat bubble format with avatars. Use "user" or "assistant" roles to distinguish between human and AI messages.

Using st.chat_message()

```
import streamlit as st

# Display user message
with st.chat_message("user"):
    st.write("Hello, ChatGPT!")

# Display AI message
with st.chat_message("assistant"):
    st.write("Hello! How can I help?")
```

What It Looks Like

19 // Basic Streamlit ChatGPT Interface

Bringing it all together: **Session State** + **Chat Elements** + **ChatGPT API**

pages/chat.py

```
import streamlit as st
from openai import OpenAI

# Initialize OpenAI client
client = OpenAI(api_key=st.secrets["OPENAI_API_KEY"])

# ═══════════════════════════════════════════════════════════# STEP 1: Initialize session state for messages#
═══════════════════════════════════════════════════════════if"messages"not in st.session_state:
    st.session_state.messages = [
        {"role": "system", "content": "You are a helpful assistant."}
    ]

# ═══════════════════════════════════════════════════════════# STEP 2: Display existing messages#
═══════════════════════════════════════════════════════════for message in st.session_state.messages:
    # Skip system messages (don't show to user)if message["role"] != "system":
        with st.chat_message(message["role"]):
            st.write(message["content"])

# ═══════════════════════════════════════════════════════════# STEP 3: Handle user input#
═══════════════════════════════════════════════════════════
user_input = st.chat_input("Type your message...")

if user_input:
    # Display user message immediatelywith st.chat_message("user"):
        st.write(user_input)

    # Add user message to session state
    st.session_state.messages.append(
```

How It Works

1 Initialize

Create messages list in session state (persists across reruns)

2 Display History

Loop through messages and show in chat bubbles

3 User Input

Capture input, display it, save to session state

4 API Call

Send entire conversation to ChatGPT API

5 Save Response

Display AI response, save to session state

The Cycle

- User types message
- Save to session state
- Send to ChatGPT API
- Get AI response
- Save & display

20 // Adding Streaming Responses - Part 1

Make your chatbot feel more **responsive and alive** like the real ChatGPT!

Without Streaming

User asks a question and then...

Waiting... Waiting... Waiting...

BOOM! Entire response appears at once after 5-10 seconds

Problems

- User doesn't know if it's working
- Feels slow and unresponsive
- Poor user experience

With Streaming

User asks a question and...

The answer to your question

The answer to your question is that streaming provides a much better user experience!

Benefits

- Immediate feedback (words appear instantly)
- Feels fast and responsive
- Professional ChatGPT-like experience

How to Enable Streaming

Before (No Streaming)

```
response = client.chat.completions.create(model="gpt-4o", messages=st.session_state.messages)
```

After (With Streaming)

```
response = client.chat.completions.create(model="gpt-4o", messages=st.session_state.messages, stream=True
```

← Add this!)

21 // Streaming Implementation - Part 2

Complete Streaming Implementation

```
if user_input:
    # Display user messagewith st.chat_message("user"):
        st.write(user_input)

    # Add to session state
    st.session_state.messages.append(
        {"role": "user", "content": user_input}
    )

    # ═══════════════════════════════════════════════════════════# STREAMING: Enable stream=True parameter#
═══════════════════════════════════════════════════
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=st.session_state.messages,
        stream=True# ← Enable streaming!
    )

    # ═══════════════════════════════════════════════════════════# STEP 1: Create empty placeholder for AI response#
═══════════════════════════════════════════════════════════with st.chat_message("assistant"):
        message_placeholder = st.empty()
        full_response = ""# ═══════════════════════════════════════════════════════# STEP 2: Process chunks as they arrive#
═══════════════════════════════════════════════════════for chunk in response:
            # Extract content from chunkif chunk.choices[0].delta.content is not None:
                content = chunk.choices[0].delta.content
                full_response += content

                # ═══════════════════════════════════════════════════════# STEP 3: Update display with cursor effect#
═══════════════════════════════════════════════════
                message_placeholder.markdown(full_response + "▌")
```

How It Works

Streaming Flow

1. Create empty placeholder
2. Receive chunk from API
3. Add chunk to response
4. Update display + cursor
5. Repeat until done
6. Remove cursor

Key Changes

- → stream=True parameter
- → st.empty() placeholder
- → Loop through chunks
- → Cursor effect "▌"

Benefits

- ✓ Feels responsive
- ✓ Users see progress
- ✓ Professional UX

22 // Adding Clear Chat Button

Enhanced chat.py with Sidebar

```
import streamlit as st
from openai import OpenAI

client = OpenAI(api_key=st.secrets["OPENAI_API_KEY"])

# ============================================# SIDEBAR: Chat Controls#
============================================with st.sidebar:
    st.title("💬 Chat Controls")

    # Show message count
    message_count = len(st.session_state.get("messages", [])) - 1
    st.metric("Messages", message_count)

    # Clear chat buttonif st.button("🗑 Clear Chat", use_container_width=True):
        # Reset messages to initial state
        st.session_state.messages = [
            {"role": "system", "content": "You are a helpful assistant."}
        ]
        # Rerun to refresh the interface
        st.rerun()

# ============================================# Initialize session state#
============================================if"messages"not in st.session_state:
    st.session_state.messages = [
        {"role": "system", "content": "You are a helpful assistant."}
    ]

# Display existing messages for message in st.session_state.messages:
    if message["role"] != "system":
        with st.chat_message(message["role"]):
            st.write(message["content"])
```

Key Concepts

st.sidebar: Creates sidebar container

st.metric(): Displays metric with label and value

st.button(): Creates clickable button, returns True when clicked

st.rerun(): Forces Streamlit to rerun the script immediately

What It Looks Like

💬 Chat Controls

Messages

12

🗑 Clear Chat

How st.rerun() Works

1. User clicks "Clear Chat" button
2. Session state messages reset to initial state
3. st.rerun() forces immediate script rerun
4. Interface refreshes with empty chat

Benefits

- Users can start fresh conversations
- Message count shows conversation length
- Sidebar keeps controls accessible
- Improves user experience and control

22 // Domain-Specific AI Assistant

Customize AI behavior with **system prompts** to create specialized assistants for each domain

Generic Assistant

```
# Basic system prompt
messages = [
  {
    "role": "system",
    "content": "You are a helpful assistant."
  }
]
```

Example Output:

User:

What should I do about this phishing email?

AI:

You should delete suspicious emails and not click on links. Be careful with attachments.

Problem

Generic advice, not specific to cybersecurity domain. Lacks technical depth and actionable steps.

Cybersecurity Assistant

```
# Specialized system prompt
messages = [
  {
    "role": "system",
    "content": """You are a cybersecurity expert assistant.
- Analyze incidents and threats
- Provide technical guidance
- Explain attack vectors & mitigations
- Use standard terminology (MITRE, CVE)
- Prioritize actionable recommendations
- Tone: Professional, technical
Format: Clear steps"""
  }
]
```

Example Output:

User:

What should I do about this phishing email?

AI:

Immediate Actions: Do NOT click links or attachments; report to security/IT; forward to phishing@yourcompany.com; delete from inbox and trash.
Analysis: Check sender domain spoofing; examine headers; look for urgency/social engineering indicators.

Better!

Specific, actionable steps. Technical terminology. Structured response. Domain expertise evident.

Cybersecurity Assistant Use Cases

Incident Analysis

Analyze incidents, identify attack patterns, suggest remediation

Threat Intelligence

Explain CVEs, threat actors, attack techniques (MITRE ATT&CK)

Security Guidance

Best practices, compliance requirements, security configurations

23 // Week 8-9-10 Integration

Complete stack: **Database** → **Web UI** → **AI Intelligence**

Week 8: Database Layer

Data Storage & CRUD Operations

SQLite + Python

- **4 Tables**
  users, incidents, datasets, tickets
- **CRUD Functions**
  Create, Read, Update, Delete
- **Security**
  Bcrypt hashing, parameterized queries
- **Connection**
  connect_database() function

Week 9: Web UI Layer

Interactive Web Interface

Streamlit

- **Multi-Page App**
  Organized page structure
- **Session State**
  Persistent data across pages
- **Authentication**
  Login/logout system
- **Visualizations**
  Charts, metrics, dashboards

Week 10: AI Intelligence Layer

Intelligent Analysis & Assistance

ChatGPT API

- **Chat Interface**
  Streaming responses, history
- **Domain Assistants**
  Cyber, Data Science, IT Ops
- **Data Analysis**
  AI-enhanced incident analysis
- **API Integration**
  Secure secrets management

Example Data Flow: User Analyzes Security Incident with AI

Week 8
Fetch incident from database

→

Week 9
Display in Streamlit UI

→

Week 10
AI analyzes & provides insights

→

User
Receives actionable recommendations

24 // AI-Enhanced Incident Analysis

Combine **Week 8 database** with **Week 10 AI** to get intelligent incident analysis

ai_incident_analyzer.py

```
import streamlit as st
from openai import OpenAI
from database.incidents import get_all_incidents  # Week 8from database.db import connect_database  # Week 8

client = OpenAI(api_key=st.secrets["OPENAI_API_KEY"])

st.title("🔍 AI Incident Analyzer")

# ═══════════════════════════════════════════════# STEP 1: Fetch incident from Week 8 database#
═══════════════════════════════════════════════
conn = connect_database()
incidents = get_all_incidents(conn)

if incidents:
    # Let user select an incident
    incident_options = [
        f"{inc['id']}: {inc['incident_type']} - {inc['severity']}"for inc in incidents
    ]

    selected_idx = st.selectbox(
        "Select incident to analyze:",
        range(len(incidents)),
        format_func=lambda i: incident_options[i]
    )

    incident = incidents[selected_idx]

    # Display incident details
    st.subheader("📋 Incident Details")
    st.write(f"**Type:** {incident['incident_type']}")
    st.write(f"**Severity:** {incident['severity']}")
    st.write(f"**Description:** {incident['description']}")
```

AI Analysis Output

AI Analysis

1. Root Cause Analysis

This phishing attack succeeded due to lack of email authentication (SPF/DKIM) and insufficient user training. The attacker spoofed a trusted domain.

2. Immediate Actions

- Reset compromised credentials
- Block sender domain
- Scan affected systems
- Notify security team

3. Prevention Measures

- Implement DMARC policy
- Deploy email filtering
- Conduct security training

4. Risk Assessment

High risk of data breach. Recommend immediate action within 24 hours.

Key Benefits

Instant Expert Analysis

Get professional-level insights in seconds

Actionable Steps

Clear, prioritized recommendations

Learning Tool

Understand attack patterns and defenses

Database Integration

Works with existing Week 8 data

25 // Multi-Domain AI Assistants

Create **three specialized assistants** for your project's three domains

Cybersecurity

Security Expert Assistant

System Prompt

"You are a cybersecurity expert. Analyze incidents, threats, and vulnerabilities. Provide technical guidance using MITRE ATT&CK, CVE references. Prioritize actionable recommendations."

Key Capabilities

- Incident analysis & triage
- Threat intelligence lookup
- Security best practices
- Remediation recommendations

Example Question

"Analyze this phishing incident and recommend next steps"

Data Science

Analytics Expert Assistant

System Prompt

"You are a data science expert. Help with data analysis, visualization, statistical methods, and machine learning. Explain concepts clearly and suggest appropriate techniques."

Key Capabilities

- Dataset analysis & insights
- Visualization recommendations
- Statistical methods guidance
- ML model suggestions

Example Question

"What's the best way to visualize this time-series data?"

IT Operations

Infrastructure Expert Assistant

System Prompt

"You are an IT operations expert. Help troubleshoot issues, optimize systems, manage tickets, and provide infrastructure guidance. Focus on practical solutions."

Key Capabilities

- Ticket triage & prioritization
- Troubleshooting guidance
- System optimization tips
- Infrastructure best practices

Example Question

"How should I prioritize these support tickets?"

26 // Your Week 10 Tasks & Next Steps

Week 10 Implementation Checklist

Part 1: Text-Based ChatGPT (Python)

- [ ] Get OpenAI API key and purchase credits ($5 minimum)
- [ ] Install `openai` library and create first API call
- [ ] Build interactive console chat with conversation history
- [ ] Implement secure API key storage using `.env` file

Part 2: Streamlit Integration

- [ ] Set up Streamlit secrets in `.streamlit/secrets.toml`
- [ ] Create chat interface using `st.chat_message` and `st.chat_input`
- [ ] Add streaming responses with `stream=True` parameter
- [ ] Add sidebar with clear chat button and message counter

Part 3: Domain-Specific Assistants

- [ ] Create specialized system prompts for all 3 domains
- [ ] Integrate AI with Week 8 database (analyze incidents/tickets/datasets)
- [ ] Add AI assistant pages to your multi-page Streamlit app

Resources

Documentation

- OpenAI API Docs
- Streamlit Chat Elements
- Week 10 Tutorial Materials

Support

- Office hours for help
- Discussion forums
- Example code provided

What's Next?

Week 11: Final Integration

- Polish your complete platform
- Test all features together
- Prepare for final submission
- Demo presentation

Final Project

Complete Multi-Domain Intelligence Platform with:

✓ Database (Week 8)
✓ Web UI (Week 9)
✓ AI Intelligence (Week 10)