CST1510 Week 10 Lab: ChatGPT API Integration

Introduction

In this lab, you will learn how to integrate OpenAI's ChatGPT API into your Multi-Domain Intelligence Platform. We will start with a simple text-based implementation to understand the fundamentals, then progress to building a professional web interface using Streamlit.

Why This Matters: AI integration adds intelligent analysis capabilities to your platform, allowing users to get expert insights on cybersecurity incidents, data analysis, and IT operations—all three domains of your project.

Prerequisites

Software Requirements:

- Python 3.8 or higher installed
- VS Code or any Python IDE
- Terminal/Command Prompt access

Knowledge Requirements:

- Basic Python programming
- Understanding of functions and variables
- Familiarity with Streamlit (from Week 9)

Account Requirements:

- OpenAI account (free to create)
- Credit card for API credits ($5 minimum purchase)

Part 1: Text-Based ChatGPT API Implementation

In this section, we will build a simple command-line chatbot using Python. This helps you understand the API fundamentals before adding a web interface.

Step 1.1: Set Up OpenAI Account and Get API Key

1. Create OpenAI Account

- Go to https://platform.openai.com
- Click "Sign Up" or "Log In" if you already have an account
- Complete the registration process

2. Purchase API Credits

OpenAI's API is pay-as-you-go. You need to add credits before making API calls.

- Click on your profile icon (top right)
- Select **"Billing"**
- Click **"Add payment method"** and enter your credit card details
- Click **"Add to credit balance"**
- Purchase at least **$5** (this will last for many API calls during development)

3. Create API Key

- In the OpenAI Platform, click on **"API keys"** in the left sidebar
- Click **"Create new secret key"**
- Give it a name (e.g., "Week10_Lab")
- **IMPORTANT**: Copy the API key immediately and save it somewhere safe
- You won't be able to see it again after closing the dialog

Example API Key Format: sk-proj-abc123... (starts with sk-proj-)

Step 1.2: Install OpenAI Python Library

Open your terminal or command prompt and install the OpenAI library:

```
pip install openai
```

Or if you're using Python 3:

```
pip3 install openai
```

Verify Installation:

```
pip show openai
```

You should see version information (e.g., Version: 1.x.x).

Step 1.3: Your First API Call

Create a new Python file called chatgpt_basic.py and add the following code:

```
from openai import OpenAI


# Initialize the OpenAI client with your API key
client = OpenAI(api_key='YOUR_API_KEY_HERE')

# Make a simple API call
completion = client.chat.completions.create(
  model="gpt-4o",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello! What is AI?"}
  ]
)

# Print the response
print(completion.choices[0].message.content)
```

Replace YOUR_API_KEY_HERE with your actual API key from Step 1.1.

Run the code:

```
python chatgpt_basic.py
```

Expected Output: You should see ChatGPT's response explaining what AI is.

Step 1.4: Understanding Message Roles

ChatGPT uses a **conversation format** with three types of message roles:

Role	Purpose	Example
system	Sets the behavior and personality of the AI	"You are a cybersecurity expert."
user	Messages from the human user	"What is a phishing attack?"
assistant	Responses from ChatGPT	"A phishing attack is..."

Why This Matters: The system message allows you to customize the AI's behavior for different domains (Cybersecurity, Data Science, IT Operations).

Example - Cybersecurity Expert:

```
messages=[
    {"role": "system", "content": "You are a cybersecurity expert. Provide technical analysis and actionable recommendations."},
    {"role": "user", "content": "What should I do about a ransomware attack?"}
]
```

Step 1.5: Interactive Console Chat

Now let's build an interactive chatbot that maintains conversation history. Create chatgpt_interactive.py:

```
from openai import OpenAI

# Initialize client
client = OpenAI(api_key='YOUR_API_KEY_HERE')

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

print("ChatGPT Console Chat (type 'quit' to exit)")
print("-" * 50)

while True:
    # Get user input
    user_input = input("You: ")

    # Exit condition
    if user_input.lower() == 'quit':
        print("Goodbye!")
        break

    # Add user message to history
    messages.append({"role": "user", "content": user_input})

    # Get AI response
    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=messages
    )

    # Extract response
    assistant_message = completion.choices[0].message.content

    # Add assistant response to history
    messages.append({"role": "assistant", "content": assistant_message})

    # Display response
```

```
    print(f"AI: {assistant_message}\n")
```

Run it:

```
python chatgpt_interactive.py
```

Try this conversation:

```
You: What is Python?
```

```
AI: [explains Python]
You: What are its main uses?
```

```
AI: [explains uses, remembering context from previous question]
```

Key Concept: By maintaining the messages list, the AI remembers the conversation context!

Step 1.6: Secure API Key Storage

IMPORTANT SECURITY PRACTICE: Never hardcode API keys in your code, especially if you're sharing code or pushing to GitHub.

Solution: Use environment variables with a .env file.

1. Install python-dotenv:

```
pip install python-dotenv
```

2. Create .env file in your project directory:

```
OPENAI_API_KEY=sk-proj-your-actual-api-key-here
```

3. Create .gitignore file to prevent committing secrets:

```
.env
__pycache__/
*.pyc
```

4. Update your code (chatgpt_secure.py):

```
from openai import OpenAI
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Get API key from environment variable
api_key = os.getenv('OPENAI_API_KEY')
```

```
# Initialize client
client = OpenAI(api_key=api_key)

# Rest of your code...
messages = [
  {"role": "system", "content": "You are a helpful assistant."}
]

print("ChatGPT Console Chat (type 'quit' to exit)")
print("-" * 50)

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

  if user_input.lower() == 'quit':
    print("Goodbye!")
    break

  messages.append({"role": "user", "content": user_input})

  completion = client.chat.completions.create(
    model="gpt-4o",
    messages=messages
  )

  assistant_message = completion.choices[0].message.content
  messages.append({"role": "assistant", "content": assistant_message})

  print(f"AI: {assistant_message}\n")
```

✅ **Part 1 Complete!** You now understand:

- How to authenticate with OpenAI API
- Message roles (system, user, assistant)
- Maintaining conversation history
- Secure API key storage

Part 2: Understanding Streamlit Chat Functions and Sessions

Before building the web interface, let's understand the key Streamlit components we'll use.

2.1: Streamlit Chat Elements Overview

Streamlit provides two special functions for building chat interfaces:

Function	Purpose	Returns
st.chat_input()	Creates an input box for user messages	User's text input (string)
st.chat_message()	Creates a message bubble (user or assistant)	Context manager for displaying content

Official Documentation: https://docs.streamlit.io/develop/api-reference/chat

---

2.2: st.chat_input() - Getting User Input

Purpose: Creates a text input box at the bottom of the page where users can type messages.

Syntax:

```
user_input = st.chat_input("Placeholder text here")
```

Example:

```
import streamlit as st

st.title("Chat Input Demo")
prompt = st.chat_input("Type your message here...")
if prompt:
    st.write(f"You said: {prompt}")
```

How It Works:

1. Displays an input box at the bottom of the page
2. When user types and presses Enter, the value is returned
3. Returns None if no input has been submitted
4. Use if prompt: to check if user submitted something

2.3: st.chat_message() - Displaying Messages

Purpose: Creates a message bubble styled for either "user" or "assistant" (AI).

Syntax:

```
with st.chat_message("user"):  # or "assistant"
    st.write("Message content here")
```

Example:

```
import streamlit as st


st.title("Chat Message Demo")

# User message (appears on right, different color)
with st.chat_message("user"):
    st.write("Hello, how are you?")

# Assistant message (appears on left, different color)
with st.chat_message("assistant"):
    st.write("I'm doing well, thank you! How can I help you today?")
```

Visual Result:

- **User messages**: Typically appear on the right with one color scheme
- **Assistant messages**: Appear on the left with a different color scheme
- Streamlit automatically handles the styling!

2.4: st.session_state - Maintaining Conversation History

The Problem: Streamlit reruns your entire script every time a user interacts with it. This means variables are reset on each rerun!

Example of the Problem:

```
import streamlit as st

# This list gets reset to empty on every rerun!
messages = []

prompt = st.chat_input("Say something")

if prompt:
    messages.append(prompt)  # Added, but lost on next rerun

    st.write(f"Messages so far: {messages}")  # Always shows only 1 message!
```

The Solution: st.session_state - a special Streamlit object that persists data across reruns.

Official Documentation: https://docs.streamlit.io/develop/api-reference/caching-and-state/st.session_state

---

2.5: How st.session_state Works

Concept: Think of st.session_state as a dictionary that survives across page reruns within the same browser session.

Syntax:

```
# Check if a key exists
if 'my_variable' not in st.session_state:
    st.session_state.my_variable = initial_value

# Access the value
value = st.session_state.my_variable

# Update the value
st.session_state.my_variable = new_value
```

Example - Counter That Persists:

```python
import streamlit as st

st.title("Session State Demo")

# Initialize counter if it doesn't exist
if 'count' not in st.session_state:
    st.session_state.count = 0

# Display current count
st.write(f"Count: {st.session_state.count}")

# Button to increment
if st.button("Increment"):
    st.session_state.count += 1
    st.rerun()  # Force a rerun to show updated value
```

Try it: Click the button multiple times. The count persists!

2.6: Using Session State for Chat History

Pattern for Chat Applications:

```python
import streamlit as st

# Initialize messages list in session state
if 'messages' not in st.session_state:
    st.session_state.messages = []

# Display all previous messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get new user input
prompt = st.chat_input("Type your message...")

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

    # Display immediately
    with st.chat_message("user"):
        st.markdown(prompt)
```

Key Points:

5 **Initialize once**: Check if 'messages' exists before creating it
6 **Store as list of dicts**: Each message has "role" and "content"
7 **Display history first**: Loop through all messages before getting new input
8 **Append new messages**: Add to st.session_state.messages to persist them

---

2.7: Complete Session State Example

Create streamlit_session_demo.py:

```
import streamlit as st

st.title("Chat History with Session State")

# Initialize session state for messages
if 'messages' not in st.session_state:
    st.session_state.messages = []

# Display all historical messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get user input
prompt = st.chat_input("Say something...")

if prompt:
    # Save user message
    st.session_state.messages.append({
        "role": "user",
        "content": prompt
    })

    # Display user message
    with st.chat_message("user"):
        st.markdown(prompt)

    # Simulate AI response (we'll replace this with real API call later)
    ai_response = f"You said: {prompt}"

    # Save AI response
    st.session_state.messages.append({
        "role": "assistant",
        "content": ai_response
    })

    # Display AI response
    with st.chat_message("assistant"):
        st.markdown(ai_response)
```

Run it:

```
streamlit run streamlit_session_demo.py
```

Test: Send multiple messages. Notice they all persist in the chat history!

Part 2 Complete! You now understand:

- st.chat_input() for getting user messages
- st.chat_message() for displaying messages
- st.session_state for persisting data across reruns
- How to maintain chat history

Part 3: Building ChatGPT Interface in Streamlit

Now we'll combine everything: Streamlit chat UI + OpenAI API + Session State.

Step 3.1: Set Up Streamlit Secrets

Instead of using .env files, Streamlit has its own **secrets management system** (similar to what you learned in Week 9 for database password security!).

Why Streamlit Secrets?

- Secure - API keys never appear in your code

- Easy to deploy - Works seamlessly on Streamlit Cloud
- Simple syntax - Access with st.secrets["KEY_NAME"]

Step 3.1.1: Create the .streamlit Folder

In your project folder, create a new folder called .streamlit:

Option A: Using Terminal/Command Prompt

```
mkdir .streamlit
```

Option B: Using VS Code

- Right-click in the Explorer panel
- Select "New Folder"
- Name it .streamlit (don't forget the dot!)

Option C: Using File Explorer (Windows/Mac)

- Navigate to your project folder
- Create a new folder
- Name it .streamlit

Your folder structure should now look like:

```
your_project/
├── .streamlit/          ← New folder!
├── chatgpt_basic.py
├── chatgpt_interactive.py
└── ...
```

Step 3.1.2: Create the secrets.toml File

Inside the .streamlit folder, create a file called secrets.toml:

Option A: Using Terminal/Command Prompt

```
touch .streamlit/secrets.toml
```

Or on Windows:

```
type nul > .streamlit\secrets.toml
```

Option B: Using VS Code

- Click on the .streamlit folder
- Right-click → "New File"
- Name it secrets.toml

Your folder structure should now look like:

```
your_project/
├── .streamlit/
│   └── secrets.toml      ← New file!
├── chatgpt_basic.py
└── ...
```

Step 3.1.3: Add Your API Key to secrets.toml

3️⃣ **Open secrets.toml** in VS Code and add your OpenAI API key:

```
OPENAI_API_KEY = "paste-your-key-here"
```

Example (with a fake key):

```
OPENAI_API_KEY = "sk-proj-abc123xyz789..."
```

IMPORTANT:

- Replace paste-your-key-here with your **actual API key** from Step 1.1
- Keep the **quotes** around the key
- No spaces around the = sign
- Save the file (Ctrl+S or Cmd+S)

Step 3.1.4: Protect Your Secrets (Security!)

Create or update .gitignore to prevent accidentally committing your secrets to GitHub:

Create .gitignore file in your project root (if it doesn't exist):

```
# Streamlit secrets (NEVER commit this!)
.streamlit/secrets.toml

# Environment variables
.env

# Python cache
__pycache__/
*.pyc
*.pyo

# Virtual environment
venv/
env/
```

Your final folder structure:

```
your_project/
├── .streamlit/
│   └── secrets.toml      ← Contains your API key
├── .gitignore            ← Protects secrets.toml
├── chatgpt_basic.py
└── ...
```

Step 3.1.5: Access Secrets in Your Code

Use the secret in your Streamlit app:

```
import streamlit as st
```

```
# Access your API key from secrets
api_key = st.secrets["OPENAI_API_KEY"]

# Use it with OpenAI client
from openai import OpenAI
```

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

Test your setup - Create test_secrets.py:

```
import streamlit as st
```

```
st.title("Test Secrets Setup")

# Try to access the secret
try:
    api_key = st.secrets["OPENAI_API_KEY"]
    st.success(" API key loaded successfully!")
    st.write(f"Key starts with: {api_key[:10]}...")
except Exception as e:
    st.error(f"Error loading API key: {e}")
```

```
    st.info("Make sure .streamlit/secrets.toml exists and contains OPENAI_API_KEY")
```

Run the test:

```
streamlit run test_secrets.py
```

Expected output: Green success message showing your key starts with sk-proj-...

Secure Setup Complete!

Step 3.2: Basic ChatGPT Streamlit App

Create chatgpt_streamlit.py:

```
import streamlit as st
```

```
from openai import OpenAI
```

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

# Page title
st.title("💬 ChatGPT - OpenAI API")

# Initialize session state for messages
if 'messages' not in st.session_state:
    st.session_state.messages = []

# Display all previous messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get user input
prompt = st.chat_input("Say something...")

if prompt:
    # Display user message
    with st.chat_message("user"):
        st.markdown(prompt)

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

    # Call OpenAI API
    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=st.session_state.messages
    )

    # Extract assistant response
    response = completion.choices[0].message.content

    # Display assistant response
    with st.chat_message("assistant"):
        st.markdown(response)

    # Add assistant response to session state
    st.session_state.messages.append({
        "role": "assistant",
        "content": response
    })
```

Run it:

```
streamlit run chatgpt_streamlit.py
```

Test the app:

9. Type a message and press Enter
10. Wait for ChatGPT's response
11. Send another message - notice it remembers context!
12. Refresh the page - history is cleared (expected behavior)

Working ChatGPT Interface!

Step 3.3: Adding System Prompt for Domain Customization

Let's customize the AI for the Cybersecurity domain. Update chatgpt_streamlit.py:

```
import streamlit as st
```

```
from openai import OpenAI

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

# Page title
st.title("🛡️ Cybersecurity AI Assistant")

# Initialize session state with system prompt
if 'messages' not in st.session_state:
  st.session_state.messages = [
    {
      "role": "system",
      "content": """You are a cybersecurity expert assistant.
      - Analyze incidents and threats
      - Provide technical guidance
      - Explain attack vectors and mitigations
      - Use standard terminology (MITRE ATT&CK, CVE)
      - Prioritize actionable recommendations
      Tone: Professional, technical
      Format: Clear, structured responses"""
    }
  ]
```

```python
# Display all previous messages (skip system message)
for message in st.session_state.messages:
  if message["role"] != "system":  # Don't display system prompt
    with st.chat_message(message["role"]):
      st.markdown(message["content"])

# Get user input
prompt = st.chat_input("Ask about cybersecurity...")

if prompt:
  # Display user message
  with st.chat_message("user"):
    st.markdown(prompt)

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

  # Call OpenAI API
  completion = client.chat.completions.create(
    model="gpt-4o",
    messages=st.session_state.messages
  )

  # Extract assistant response
  response = completion.choices[0].message.content

  # Display assistant response
  with st.chat_message("assistant"):
    st.markdown(response)

  # Add assistant response to session state
  st.session_state.messages.append({
    "role": "assistant",
    "content": response
  })
```

Test it: Ask "What is a phishing attack?" and notice the technical, structured response!

Part 4: Adding Streaming for Better UX

The Problem: When you ask a complex question, there's a delay (5-10 seconds) before the entire response appears. This feels slow and unresponsive.

The Solution: **Streaming** - Display the response word-by-word as it's generated, just like the real ChatGPT!

Step 4.1: Understanding Streaming

Without Streaming:

```
User asks question → [Wait 10 seconds] → BOOM! Full response appears
```

With Streaming:

```
User asks question → Words appear one by one → Complete!
```

How to Enable: Add stream=True parameter to the API call.

Step 4.2: Implementing Streaming

Create chatgpt_streamlit_streaming.py:

```python
import streamlit as st
```

```python
from openai import OpenAI

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

# Page title
st.title("💬 ChatGPT with Streaming")

# Initialize session state
if 'messages' not in st.session_state:
    st.session_state.messages = []

# Display all previous messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])
```

```
# Get user input
prompt = st.chat_input("Say something...")

if prompt:
    # Display user message
    with st.chat_message("user"):
        st.markdown(prompt)

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

    # Call OpenAI API with streaming enabled
    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=st.session_state.messages,
        stream=True  # ← Enable streaming!
    )

    # Display streaming response
    with st.chat_message("assistant"):
        container = st.empty()  # Create empty container
        full_reply = ""  # Accumulate response

        # Process each chunk as it arrives
        for chunk in completion:
            delta = chunk.choices[0].delta
            if delta.content:  # If chunk has content
                full_reply += delta.content  # Add to full response
                container.markdown(full_reply)  # Update display

    # Save complete response to session state
    st.session_state.messages.append({
        "role": "assistant",
        "content": full_reply
    })
```

Run it:

```
streamlit run chatgpt_streamlit_streaming.py
```

Test: Ask a complex question like "Explain how SQL injection works" and watch the response appear word-by-word!

Step 4.3: Understanding the Streaming Code

Let's break down the streaming implementation:

```
# 1. Enable streaming in API call

completion = client.chat.completions.create(
   model="gpt-4o",
   messages=st.session_state.messages,
   stream=True  # Returns a generator instead of complete response
)

# 2. Create empty container to update
with st.chat_message("assistant"):
   container = st.empty()  # Placeholder that can be updated
   full_reply = ""  # String to accumulate the complete response

   # 3. Process each chunk as it arrives
   for chunk in completion:  # Loop through streaming chunks
      delta = chunk.choices[0].delta  # Get the new content
      if delta.content:  # Check if chunk has content
         full_reply += delta.content  # Add to accumulated response
         container.markdown(full_reply)  # Update display with current text

# 4. Save complete response
st.session_state.messages.append({
   "role": "assistant",
   "content": full_reply  # Save the complete accumulated response

})
```

Key Concepts:

13 **stream=True**: Changes API to return chunks instead of complete response
14 **st.empty()**: Creates a placeholder that can be updated repeatedly
15 **for chunk in completion**: Loops through each piece of the response
16 **delta.content**: Contains the new text in each chunk
17 **Accumulate and update**: Build full response while updating display

Step 4.4: Complete Production-Ready Code

Here's the final, polished version with all features:

```
import streamlit as st
```

```
from openai import OpenAI

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

# Page configuration
st.set_page_config(
    page_title="ChatGPT Assistant",
    page_icon="💬",
    layout="wide"
)

# Title
st.title("💬 ChatGPT - OpenAI API")
st.caption("Powered by GPT-4o")

# Initialize session state
if 'messages' not in st.session_state:
    st.session_state.messages = []

# Sidebar with controls
with st.sidebar:
    st.subheader("Chat Controls")

    # Display message count
    message_count = len([m for m in st.session_state.messages if m["role"] != "system"])
    st.metric("Messages", message_count)

    # Clear chat button
    if st.button("🗑️ Clear Chat", use_container_width=True):
        st.session_state.messages = []
        st.rerun()

    # Model selection
    model = st.selectbox(
        "Model",
        ["gpt-4o", "gpt-4o-mini"],
        index=0
    )

    # Temperature slider
    temperature = st.slider(
        "Temperature",
        min_value=0.0,
        max_value=2.0,
        value=1.0,
        step=0.1,
        help="Higher values make output more random"
    )

# Display all previous messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get user input
prompt = st.chat_input("Say something...")

if prompt:
    # Display user message
    with st.chat_message("user"):
        st.markdown(prompt)

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

    # Call OpenAI API with streaming
    with st.spinner("Thinking..."):
        completion = client.chat.completions.create(
            model=model,
            messages=st.session_state.messages,
            temperature=temperature,
            stream=True
        )

    # Display streaming response
    with st.chat_message("assistant"):
        container = st.empty()
        full_reply = ""

        for chunk in completion:
            delta = chunk.choices[0].delta
            if delta.content:
                full_reply += delta.content
                container.markdown(full_reply + "▌ ")  # Add cursor effect

        # Remove cursor and show final response
        container.markdown(full_reply)

    # Save assistant response
    st.session_state.messages.append({
        "role": "assistant",
        "content": full_reply
    })
```

Run it:

```
streamlit run chatgpt_streamlit_streaming.py
```

Troubleshooting

Issue 1: "AuthenticationError: Invalid API Key"

Cause: API key is incorrect or not properly loaded.

Solutions:

18 Check that your API key starts with sk-proj-
19 Verify .streamlit/secrets.toml has correct format:

```
OPENAI_API_KEY = "sk-proj-your-key-here"
```

20 Restart Streamlit after changing secrets
21 Check for extra spaces or quotes in the key

Issue 2: "RateLimitError: You exceeded your current quota"

Cause: You've used all your API credits or hit rate limits.

Solutions:

22 Check your usage: https://platform.openai.com/usage
23 Add more credits: Profile → Billing → Add to credit balance
24 Wait a few minutes if you're hitting rate limits

Issue 3: Messages Disappear After Refresh

Cause: Session state is cleared when the page is refreshed (expected behavior).

Solutions:

25 This is normal - session state only lasts for the current browser session
26 To persist across refreshes, you would need to save to a database (Week 8 knowledge!)
27 For this lab, this behavior is acceptable

Issue 4: Streaming Not Working / Showing Chunks

Cause: Missing st.empty() or incorrect update logic.

Solutions:

28 Make sure you create container = st.empty() before the loop
29 Update the container inside the loop: container.markdown(full_reply)
30 Check that you're accumulating: full_reply += delta.content

---

Issue 5: "ModuleNotFoundError: No module named 'openai'"

Cause: OpenAI library not installed.

Solution:

```
pip install openai
```

Or for Streamlit Cloud deployment, add to requirements.txt:

```
openai
streamlit
```

Integration with Your Project

Now you can integrate AI into your Multi-Domain Intelligence Platform:

1. Cybersecurity Domain

```
system_prompt = """You are a cybersecurity expert assistant.
Analyze incidents, threats, and provide technical guidance."""
```

2. Data Science Domain

```
system_prompt = """You are a data science expert assistant.
Help with analysis, visualization, and statistical insights."""
```

3. IT Operations Domain

```
system_prompt = """You are an IT operations expert assistant.
Help troubleshoot issues, optimize systems, and manage tickets."""
```

Additional Resources

Official Documentation:

- OpenAI API Reference
- Streamlit Chat Elements
- Streamlit Session State

Example Projects:

- Streamlit ChatGPT Clone
- OpenAI Python Examples