# **Python LeetCode Helper Chatbot**

## **Overview**

The **Python LeetCode Helper Chatbot** is an innovative web-based application designed to provide real-time assistance for users tackling Python programming challenges. With an intuitive interface and seamless backend integration, this tool serves as an interactive Python coding assistant, making it ideal for developers, students, and programming enthusiasts.

The chatbot leverages the **OpenAI GPT API** to deliver intelligent, context-aware responses, enabling users to solve complex coding problems, understand Python concepts, and even debug errors effectively. The application bridges the gap between problem-solving and learning by offering clear explanations, example code snippets, and actionable solutions.

#### **Core Features and Capabilities**

1. **Interactive Chat-Based Assistance**:
   * The chatbot employs a conversational approach to interact with users, mimicking human-like discussions for a more engaging problem-solving experience.
   * Users can ask any Python-related coding question, from basic syntax queries to advanced algorithmic challenges.
2. **AI-Powered Responses**:
   * At its core, the application utilizes OpenAI’s advanced GPT models (**GPT-3.5-turbo** or **GPT-4**) to generate dynamic, context-aware solutions.
   * The chatbot not only provides solutions but also explains the logic and step-by-step reasoning behind each response, ensuring an educational experience.
3. **Streamlit Web Interface**:
   * Built using **Streamlit**, a Python-based library for creating interactive web applications, the chatbot features a modern and user-friendly design.
   * The interface supports live interactions, a clean layout, and a responsive design suitable for both desktop and mobile users.
4. **Syntax-Highlighted Code Snippets**:
   * To enhance the user experience, the chatbot presents Python code snippets with syntax highlighting, making the code easy to read and implement.
   * Code examples are tailored to user queries, ensuring relevance and applicability.
5. **Context Preservation**:
   * The chatbot maintains a history of the conversation within the session, enabling it to provide responses that consider previous questions and answers.
   * This feature is particularly useful for multi-step problem-solving and follow-up queries.
6. **Lightweight and Scalable**:
   * The application is designed to be lightweight and fast, making it accessible to users even with limited computing resources.
   * Its architecture supports scalability, allowing it to handle multiple simultaneous users efficiently.

#### **Who Can Benefit?**

* **Beginners**: Users new to Python programming can benefit from detailed explanations, examples, and guidance on fundamental concepts.
* **Intermediate and Advanced Programmers**: The chatbot assists with debugging, algorithm optimization, and solving challenging problems.
* **Educators and Mentors**: Educators can use the tool to provide supplemental assistance to students, helping them understand concepts interactively.
* **Professional Developers**: Professionals can leverage the chatbot for quick troubleshooting and efficient problem-solving during their development workflow.

#### **Technology Stack**

1. **Backend**:
   * Powered by OpenAI's GPT models (**GPT-3.5-turbo** or **GPT-4**) for generating responses.
   * The OpenAI GPT API processes user queries and generates intelligent, context-aware outputs.
2. **Frontend**:
   * Developed using **Streamlit**, which enables rapid prototyping and deployment of interactive Python-based web apps.
   * Features a responsive UI design with a focus on simplicity and clarity.
3. **Session State Management**:
   * The chatbot maintains conversation history using Streamlit’s session state functionality, ensuring continuity in user interactions.

#### **Why This Application is Unique**

* **Educational Focus**: Unlike simple chatbots that provide answers, this application emphasizes learning by providing detailed explanations and clear Python code examples.
* **Ease of Use**: The lightweight design and conversational approach lower the barrier for entry, making advanced programming assistance accessible to users of all skill levels.
* **Real-Time Interaction**: Responses are generated in real-time, enabling users to iterate on their queries and refine their understanding.

#### **Potential Use Cases**

1. **Debugging and Problem-Solving**:
   * Users can paste error messages or problematic code snippets to get debugging advice or solutions.
2. **Algorithm and Data Structure Queries**:
   * Provides explanations and implementations for common algorithms and data structures.
3. **Learning Python Concepts**:
   * Assists users in understanding Python-specific constructs like list comprehensions, decorators, and lambda functions.
4. **Competitive Programming Support**:
   * Offers strategies and code for tackling competitive programming problems effectively.

## **Features**

### **1. Interactive Chat Interface**

* A user-friendly interface where users can ask Python coding-related questions.
* Displays conversation history between the user and the bot.

### **2. OpenAI GPT Integration**

* Leverages GPT-3.5-turbo or GPT-4 for generating responses.
* Provides detailed explanations, example code snippets, and suggestions for coding problems.

### **3. Syntax-Highlighted Code Output**

* Python code snippets are displayed with syntax highlighting for clarity and easy readability.

### **4. Conversation History**

* Maintains a history of user queries and bot responses within the session for better context.

### **5. Clear Chat Option**

* A "Clear Chat" button allows users to reset the session and start a new conversation.

### **6. Lightweight and Responsive**

* Built using Streamlit, ensuring a responsive, clean, and modern UI.
* Automatically clears input after sending queries for a seamless experience.

## **Code Explanation**

### **Imports**

```

import streamlit as st

from openai import OpenAI

```

* **streamlit**: A Python library for building web apps, used here for the chatbot UI.
* **openai**: The OpenAI Python client library for communicating with GPT models.

### **OpenAI API Setup**

```

OPENAI\_API\_KEY = "OpenAI API key" # Replace with your API key

client = OpenAI(api\_key=OPENAI\_API\_KEY)

```

* **OPENAI\_API\_KEY**: Your OpenAI API key is required to authenticate with the OpenAI API.
* **client**: Instantiates the OpenAI client for sending requests to the API.

### **Function: query\_openai\_api**

```

def query\_openai\_api(prompt, history):

try:

messages = [{"role": "system", "content": "You are a helpful assistant for solving Python coding problems."}]

messages += history

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

completion = client.chat.completions.create(

model="gpt-3.5-turbo", # Use gpt-3.5-turbo or gpt-4

messages=messages,

max\_tokens=500,

temperature=0.7

)

return completion.choices[0].message.content

except Exception as e:

return f"Error connecting to OpenAI API: {e}"

```

* **Parameters**:
  + prompt: The user’s current question.
  + history: The conversation history to provide context for the AI response.
* **Logic**:
  + Prepares a conversation with a system message, past conversation history, and the current user query.
  + Sends the data to OpenAI's GPT API and fetches the response.
* **Error Handling**:
  + Catches and displays errors if the API connection fails.

### **Main Application: main()**

```

def main():

st.set\_page\_config(page\_title="Python LeetCode Helper", layout="wide")

```

* **Page Configuration**:
  + Sets the app's title to "Python LeetCode Helper Chatbot".
  + Uses a wide layout for better content spacing.

### **Session State Initialization**

```

if "history" not in st.session\_state:

st.session\_state["history"] = []

```

* Initializes the conversation history in Streamlit's session state to persist data across interactions.

### **Conversation History Display**

```

chat\_container = st.container()

with chat\_container:

if st.session\_state["history"]:

for i, (user\_msg, bot\_msg) in enumerate(st.session\_state["history"]):

st.markdown(f"\*\*You:\*\* {user\_msg}")

st.markdown(f"\*\*Helper:\*\* {bot\_msg}")

```

* Displays the user's input and the bot's response in a chat-style format.
* Each query-response pair is iteratively rendered.

### **User Input Section**

```

with st.form("chat\_form", clear\_on\_submit=True):

user\_input = st.text\_area("Your Question:", placeholder="Ask a Python-related coding question...")

submit\_button = st.form\_submit\_button("Send")

```

* Provides a text area for user input with a placeholder for guidance.
* Includes a "Send" button for submitting queries.

### **Process User Input**

```

if submit\_button and user\_input.strip():

st.session\_state["history"].append((user\_input, "Thinking..."))

```

* Appends the user's query to the history with a temporary response ("Thinking...") to indicate processing.

### **Generate AI Response**

```

if st.session\_state["history"] and st.session\_state["history"][-1][1] == "Thinking...":

user\_msg = st.session\_state["history"][-1][0]

bot\_response = query\_openai\_api(user\_msg, [{"role": "user", "content": msg[0]} for msg in st.session\_state["history"][:-1]])

st.session\_state["history"][-1] = (user\_msg, bot\_response)

```

* Extracts the last user query.
* Calls the query\_openai\_api function to get the bot's response.
* Replaces the placeholder response with the actual response.

### **Clear Chat Button**

```

if st.button("Clear Chat"):

st.session\_state["history"] = []

```

* Resets the chat history to start a new session.

### **Run the Application**

```

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

* Executes the main() function when the script is run.

## **User Interface (UI) Explanation**

### **1. Header Section**

* **Title**: "Python LeetCode Helper Chatbot" is displayed prominently to indicate the purpose of the application.
* **Subtitle**: "Solve your Python coding problems interactively!" reinforces the chatbot's functionality.

### **2. Chat Display**

* **Conversation History**:
  + Displays previous user queries labeled as **You**.
  + Shows bot responses labeled as **Helper**.
* **Code Display**:
  + The bot's responses include Python code snippets with syntax highlighting, enhancing readability and usability.

### **3. Input Section**

* **Input Text Area**:
  + A placeholder guides users to ask Python-related coding questions.
* **Send Button**:
  + Sends the query to the bot for processing.
* **Clear Chat Button**:
  + Resets the session, clearing the conversation history.

## **Sample Prompts and Responses**

1. **Prompt**: "How do I reverse a list in Python?"

**Response**: "You can use [::-1] for slicing or .reverse() method. Here is an example:  
```  
my\_list = [1, 2, 3]

reversed\_list = my\_list[::-1]

print(reversed\_list)

```

1. **Prompt**: "What is the difference between is and == in Python?"
   * **Response**: "is checks for object identity, while == checks for value equality."

## **Potential Enhancements**

1. **Save Chat History**:
   * Allow users to download their chat history or save it locally.
2. **Themes**:
   * Add dark/light mode toggle for better accessibility.
3. **Auto-Suggestions**:
   * Provide suggestions for common queries to help users frame their questions.

## **How to Run the Python LeetCode Helper Chatbot**

Follow these steps to set up and run the chatbot application locally:

#### **1. Clone the Repository**

To begin, clone the project repository from GitHub to your local machine:

```

git clone <https://github.com/ajinabraham123/Prompt-Based-Tutor-Bot.git>

```

Navigate to the cloned project directory:

```

cd Prompt-Based-Tutor-Bot

```

#### **2. Create and Activate a Virtual Environment**

It’s recommended to use a virtual environment to manage dependencies.

**On Windows:**

```

python -m venv venv

venv\Scripts\activate

```

**On macOS/Linux:**

```

python3 -m venv venv

source venv/bin/activate

```

#### **3. Install Dependencies**

Install the required Python packages listed in the requirements.txt file:

```

pip install -r requirements.txt

```

#### **4. Set Up OpenAI API Key**

To use OpenAI's GPT API, you need to set up your API key.

1. Obtain your OpenAI API key from [OpenAI's platform](https://platform.openai.com/).

Create a .env file in the project directory and add your API key:  
```  
OPENAI\_API\_KEY=your\_openai\_api\_key

```

1. Replace your\_openai\_api\_key with your actual OpenAI API key.

#### **5. Run the Application**

Start the Streamlit application by running the following command:

```

streamlit run app.py

```

This will start the chatbot application, and the local URL (e.g., http://localhost:8501) will be displayed in the terminal.

#### **6. Access the Chatbot**

1. Open the URL shown in your terminal (e.g., http://localhost:8501) in your web browser.
2. Start interacting with the **Python LeetCode Helper Chatbot**.

#### **7. Clear Chat and Restart**

Use the "Clear Chat" button in the application to reset the session and start fresh if needed.

#### **8. Optional: Run in a Docker Container**

If you prefer to use Docker, you can build and run the application in a container:

Build the Docker image:  
```  
docker build -t prompt-tutor-bot .

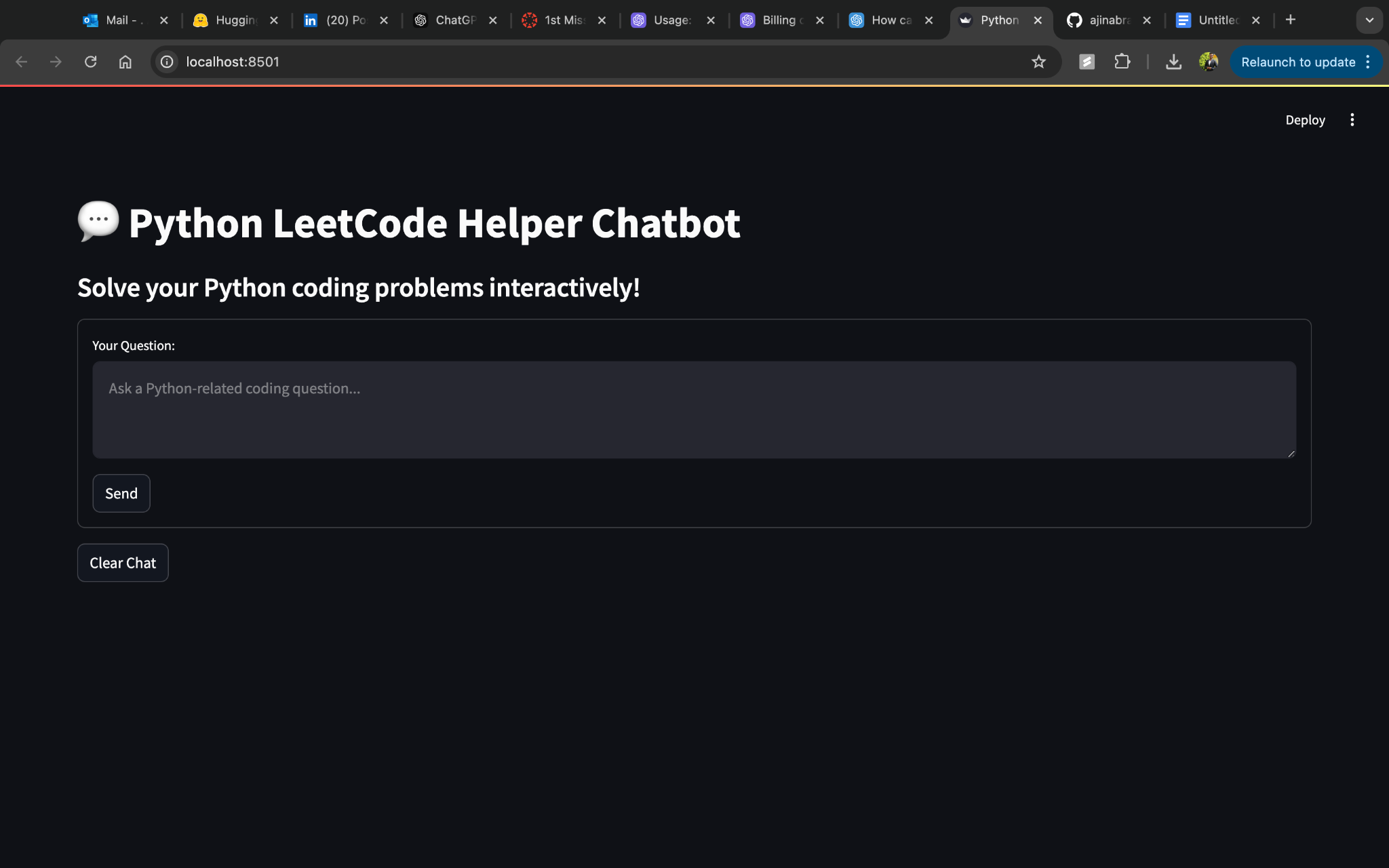
```

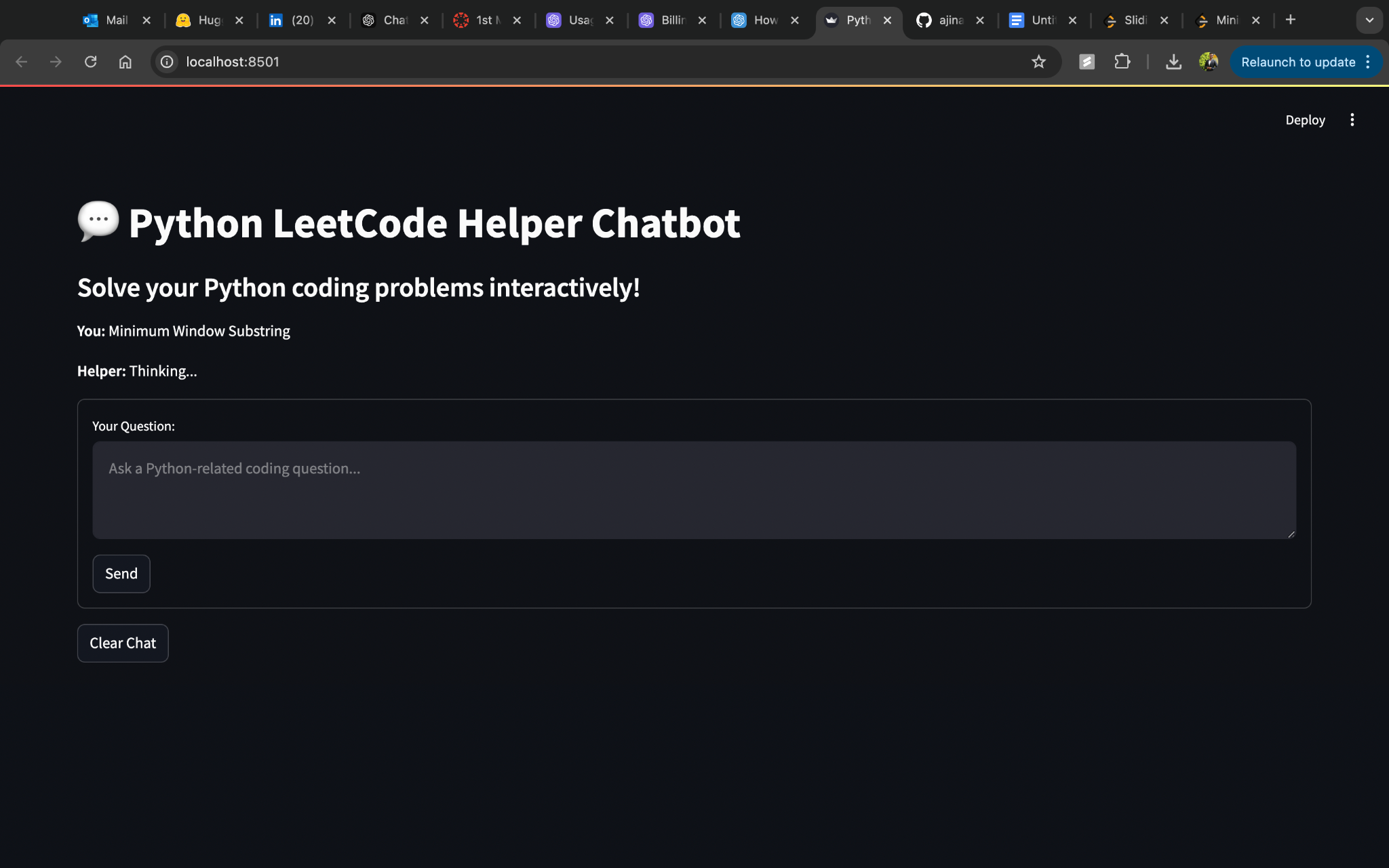
Run the Docker container:  
```  
docker run -p 8501:8501 prompt-tutor-bot

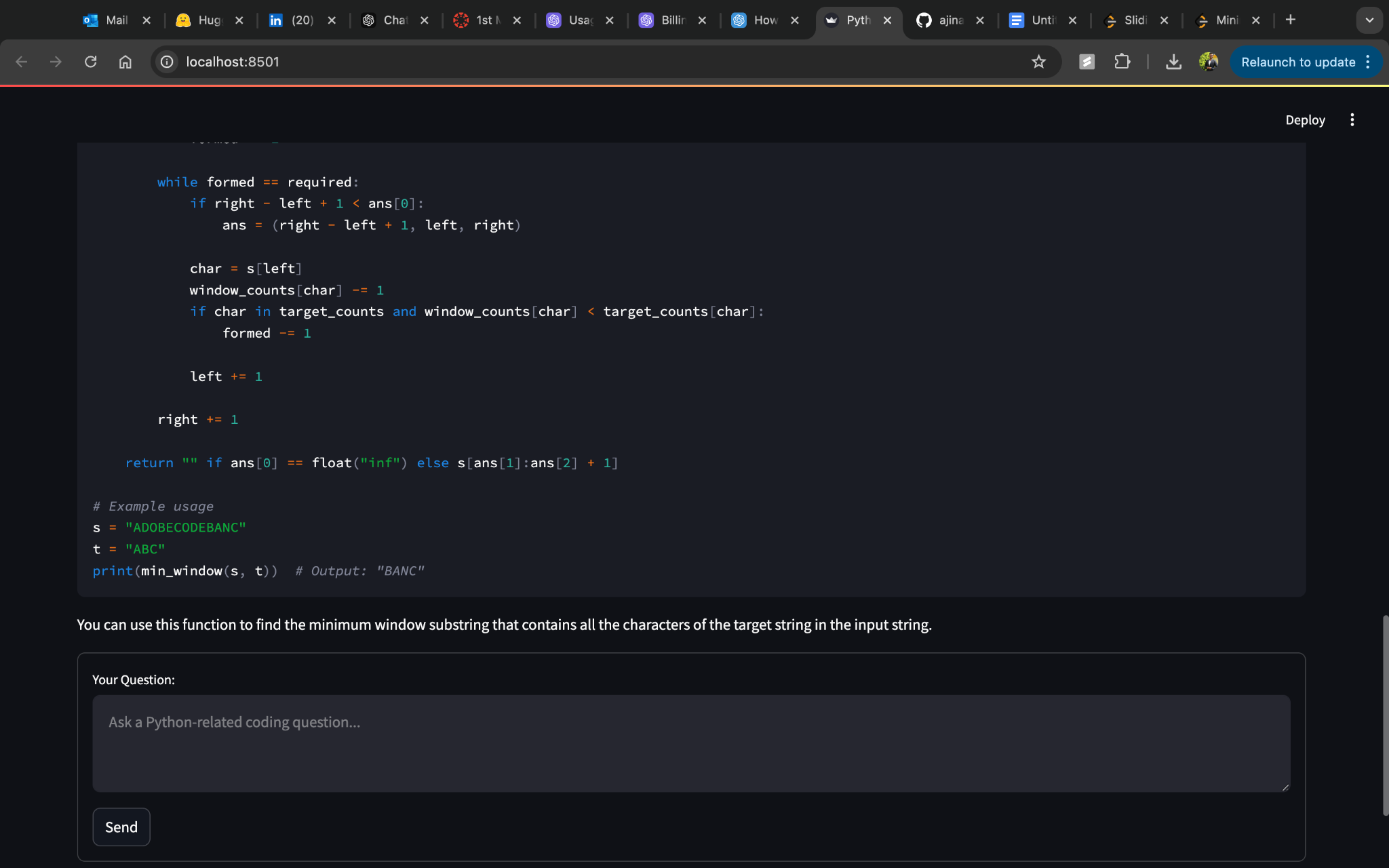
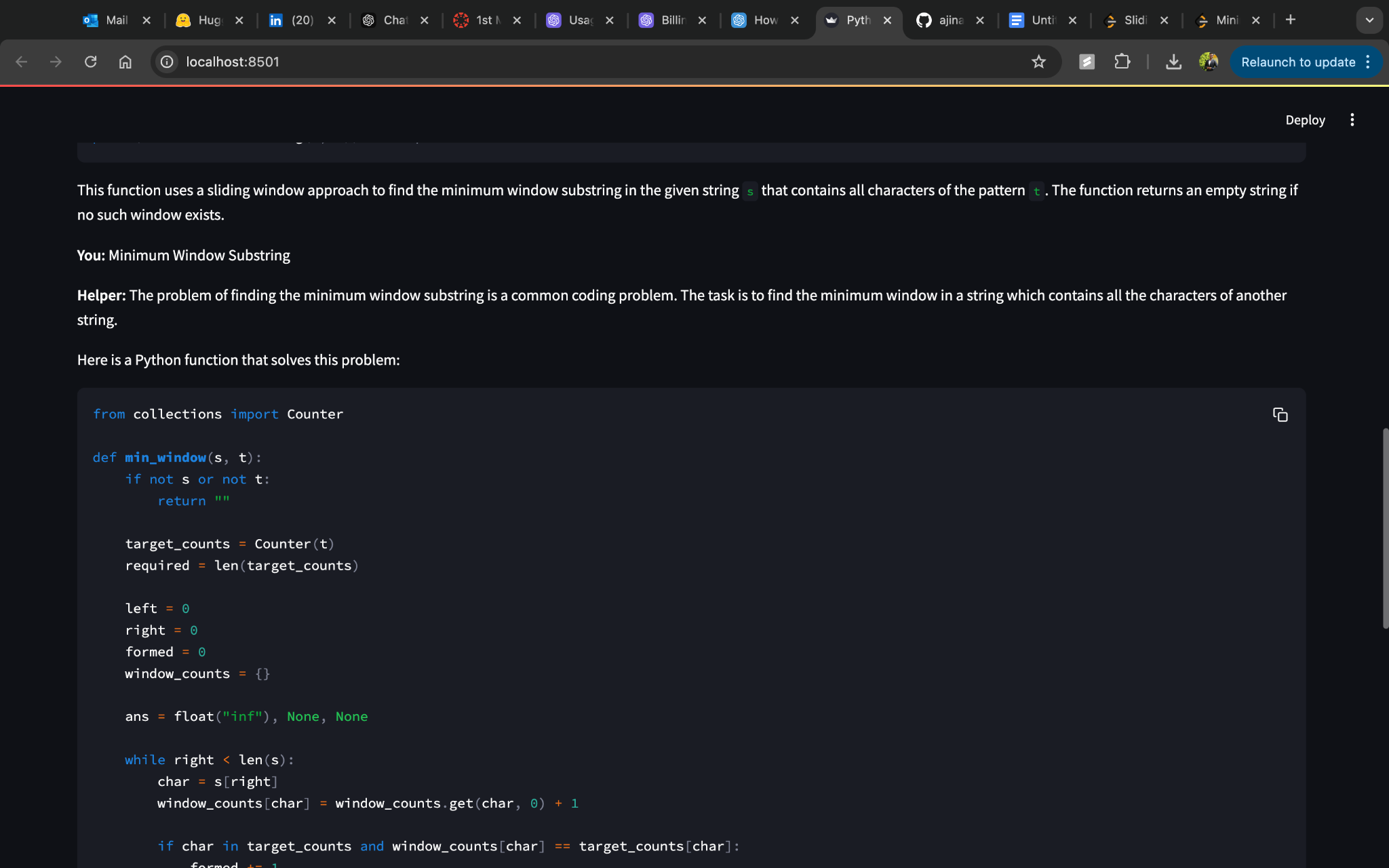
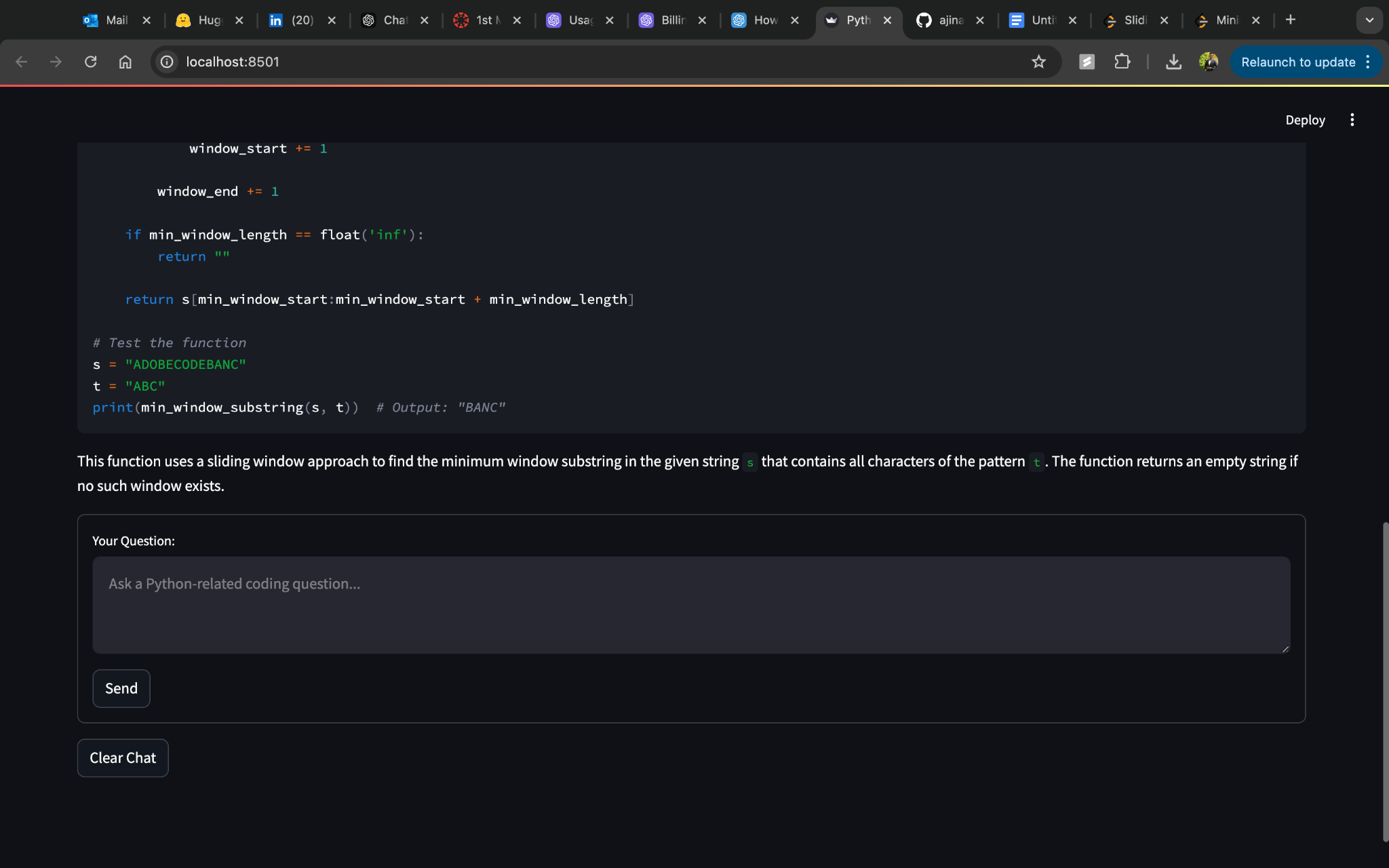
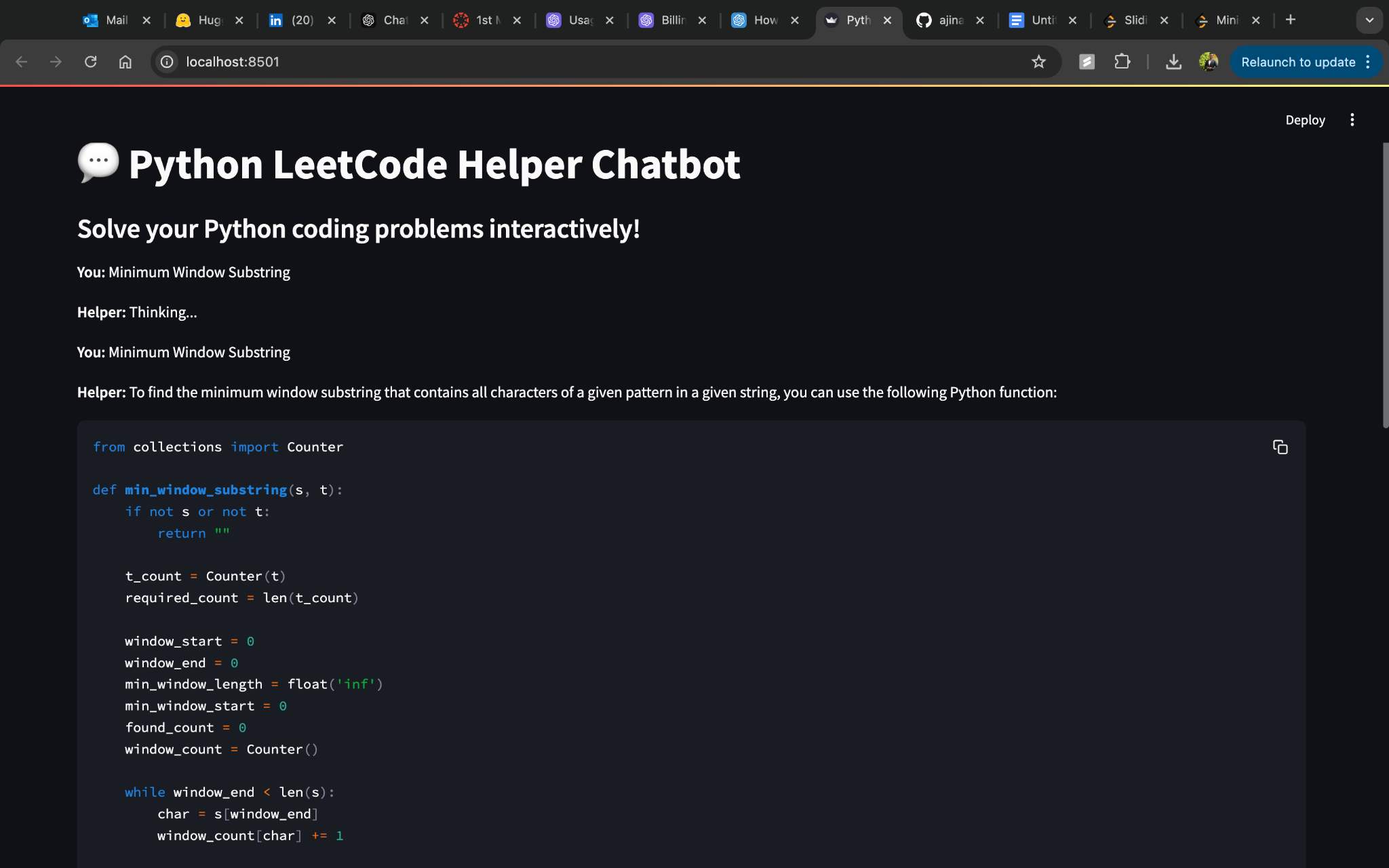
```

1. Access the application at http://localhost:8501.

## **Screenshots**







## **Conclusion**

The Python LeetCode Helper Chatbot is a robust tool for solving Python coding problems interactively. Its integration of Streamlit and OpenAI GPT models ensures a smooth user experience with detailed explanations, syntax-highlighted code snippets, and a clean interface. The application is highly extensible and can be tailored further to meet advanced user needs.