Project Documentation: AI Chatbot with File Summarization

1. Introduction

Brief Overview

This project is a web-based, interactive chatbot application built using Python and Streamlit. It provides a conversational interface powered by a local Large Language Model (LLM) via Ollama. A key feature is its ability to accept file uploads, generate concise summaries of their content, and maintain a history of conversations.

Purpose and Goals

The primary goal is to offer a private, locally-run AI assistant that aids developers and other users with queries and document analysis. By leveraging a local LLM, the application ensures data privacy, as no information is sent to external cloud services.

The main objectives are:

- To provide a real-time, interactive chat interface.
- To summarize content from uploaded text, PDF, or DOCX files.
- To manage and persist chat sessions for user convenience.
- To offer a simple, clean, and responsive user interface with light and dark modes.

Target Users or Audience

The application is primarily designed for developers, researchers, and writers who need a quick way to interact with an AI model or get summaries of documents without relying on internet-based services. Its simplicity also makes it accessible to anyone interested in running a local AI chatbot.

Key Features and Business Value

- **Local LLM Integration:** Connects to any Ollama-supported model, providing flexibility and privacy.
- **File Summarization:** Users can upload documents (`.txt`, `.pdf`, `.docx`) to get instant summaries, significantly speeding up document analysis.
- **Chat History:** Automatically saves and lists conversations, allowing users to revisit and continue previous chats.
- **UI/UX Features:** Includes a dark mode toggle and pre-populated suggestion prompts to improve user experience.
- **Zero-Cost and Private:** Since it runs locally, there are no API costs, and all user data remains on their machine, which is a significant value proposition for privacy-conscious users or those handling sensitive information.

---

2. Technology Stack

- **Programming Language:** Python 3.x
- **Frontend Framework: Streamlit** - Used to create the entire interactive web user interface.
- **Backend Logic:** The application logic is self-contained within the Streamlit script. It handles state management, UI rendering, and backend communication.

- **AI/ML Model: Ollama** - Serves the local Large Language Model (e.g., `deepseek-coder:1.3b`, `phi:latest`). The application sends prompts to the Ollama API endpoint.
- **API Communication: requests** library - Used to make HTTP POST requests to the Ollama API.

---

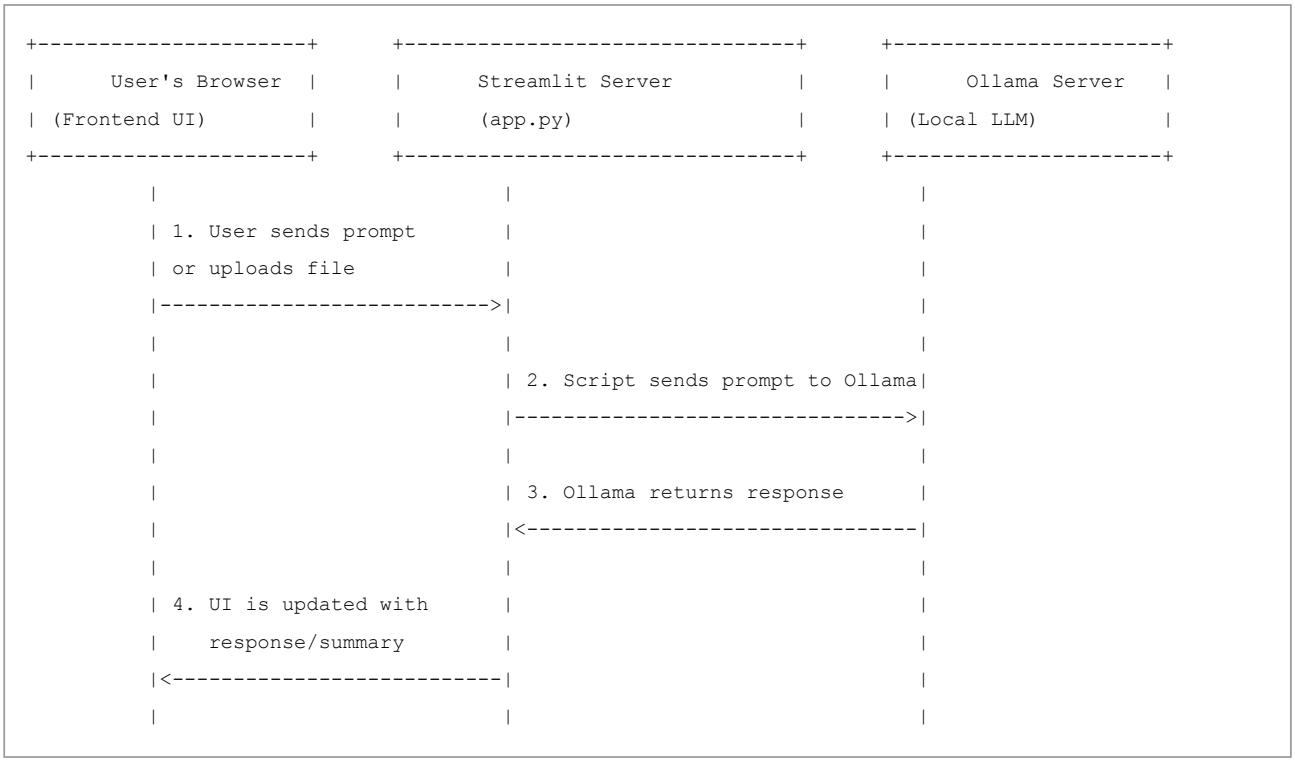
3. System Architecture

Overall Architecture

The application follows a simple, monolithic architecture contained within a single Python script. It acts as a client to the Ollama service.

- **Frontend:** The frontend is dynamically generated by Streamlit. It renders all UI components, including the chat window, sidebar, buttons, and file uploader.
- **Backend:** The same script serves as the backend. It manages the application's state (like chat history) using Streamlit's session state mechanism and processes all business logic.
- **API Gateway:** There is no dedicated API gateway. The application directly communicates with the Ollama server's `/api/generate` endpoint.
- **Database:** The application does not use a traditional database. Chat history and summaries are stored in the `st.session_state`, which is an in-memory, session-specific cache. Data is lost when the user's session ends.

Textual Diagram



---

4. Functionalities in Detail

a. File Upload and Summarization

- **Description:** A user can upload a `.txt`, `.pdf`, or `.docx` file. The system reads the file content, sends the first 4000 characters to the LLM, and displays the generated summary in a distinct UI box.
- **Inputs:** A file uploaded via the Streamlit `file_uploader`.

- **Outputs:** A string containing the summary, which is displayed on the UI.
- **Dependencies:** The `get_file_summary` function, which relies on the Ollama API.

b. Interactive Chat

- **Description:** The main interface where a user can have a conversation with the AI. The chat history is displayed in a classic message format.
- **Inputs:** A text string from the user via the `st.chat_input` component.
- **Outputs:** A text response from the AI, which is appended to the chat display.
- **Dependencies:** The Ollama API.

c. Chat History Management

- **Description:** The sidebar allows users to manage their conversations. They can start a new chat, switch between existing chats, or clear all history. Chat titles are auto-generated from the first user prompt.
- **Inputs:** User clicks on the "New Chat", "Clear All History", or specific chat buttons.
- **Outputs:** The main chat view updates to reflect the selected chat session or a cleared state.

d. Dark Mode Toggle

- **Description:** A simple toggle in the sidebar to switch the application's theme between light and dark modes for user comfort.
- **Inputs:** A boolean state from the `st.toggle` switch.
- **Outputs:** The application injects custom CSS to override default styles.

---

5. API Overview

The application does not expose its own API but consumes the Ollama API.

- **Endpoint:** `http://localhost:11434/api/generate`
- **Method:** `POST`
- **Example Request Body:**

```
{
    "model": "deepseek-coder:1.3b",
    "prompt": "How do I write a Python function to reverse a string?",
    "stream": false
}
```

- **Example Success Response Body:**

```
{
    "model": "deepseek-coder:1.3b",
    "created_at": "2023-12-20T14:00:00.000Z",
    "response": "You can reverse a string in Python using slicing, like this: `my_string[::-1]`.",
    "done": true,
    "total_duration": 5001234567,
    "prompt_eval_count": 20,
    "eval_count": 50,
    "eval_duration": 4001234567
}
```

---

6. Database Design

No formal database is used. The application relies entirely on **Streamlit's session state (`st.session_state`)** for data persistence within a single user session.

- `st.session_state.chat_histories`: A list of dictionaries, where each dictionary represents a saved chat session, containing its title, messages, and file summary.
- `st.session_state.messages`: A list of messages for the currently active chat.
- `st.session_state.file_summary`: Stores the summary of the last uploaded file.

This approach is simple but volatile, as all data is cleared when the browser tab is closed or the session times out.

---

7. Security

- **Authentication & Authorization:** The application has no authentication or authorization mechanisms. It is intended to be run locally and is accessible to anyone who can access the Streamlit server port.
- **Data Protection:** Since the Ollama model is hosted locally, all prompts and file contents remain on the user's machine. This is a core security benefit.
- **Input Validation:** The application performs basic file type validation on upload. There is no further sanitization of user input or file content, which could be a potential risk if the underlying LLM has vulnerabilities.

---

8. Deployment and Environment Setup

Step-by-Step Setup Guide

1. **Prerequisites:**
    - Python 3.8+
    - Ollama installed and running on the host machine.
2. **Install Ollama Model:**
    - Pull the desired model from the Ollama library. For example:

    ```
    ollama pull deepseek-coder:1.3b
    ```

3. **Install Python Dependencies:**
    - Create a virtual environment (recommended).
    - Install the required packages from a `requirements.txt` file.

    ```
    # requirements.txt
    streamlit
    requests
    ```

    ```
    pip install -r requirements.txt
    ```

4. **Run the Application:**

- Execute the following command in your terminal:

```
streamlit run app.py  # Replace app.py with your script name
```

- The application will be available at `http://localhost:8501`.

Configuration

The Ollama URL and model name are currently hardcoded in the script:

```
OLLAMA_URL = "http://localhost:11434/api/generate"
OLLAMA_MODEL = "deepseek-coder:1.3b"
```

For production or more flexible setups, these should be externalized into environment variables or a configuration file.

---

9. Testing

- **Testing Strategy:** Currently, no formal testing strategy or framework is implemented in the project.
- **Recommended Approach:**
  - **Unit Tests:** Use `pytest` to test individual functions like `get_file_summary`. The `requests.post` call should be mocked to isolate the function from the live Ollama API.
  - **Integration Tests:** Test the interaction between different parts of the application, such as ensuring a file upload correctly triggers a summary and updates the session state.
  - **End-to-End Tests:** Use a tool like Selenium or Playwright to simulate user interactions in the browser and verify the application's behavior.

---

10. Future Enhancements

- **Persistent Storage:** Replace the session state with a lightweight database like **SQLite** or a file-based storage system to persist chat histories across sessions.
- **Configuration Management:** Move hardcoded values like the Ollama URL and model name to a `.env` file or a settings panel in the UI.
- **Asynchronous Operations:** Use `asyncio` to make non-blocking API calls to Ollama, preventing the UI from freezing while waiting for a response.
- **Streaming Responses:** Implement support for streaming responses (`"stream": true`) from the Ollama API to display the AI's response token by token, improving perceived performance.
- **Error Handling:** Add more robust error handling for cases where the Ollama server is not reachable or returns an error.
- **Expanded File Support:** Add parsers for other common file types like `.md`, `.json`, etc.

---

11. Conclusion

This project successfully demonstrates how to build a private, efficient, and useful AI tool for local use. By combining the simplicity of Streamlit with the power of local LLMs via Ollama, it provides a secure and cost-effective solution for conversational AI and document summarization. While currently a prototype, its modular design allows for significant future enhancements that could turn it into a more robust and feature-rich application.