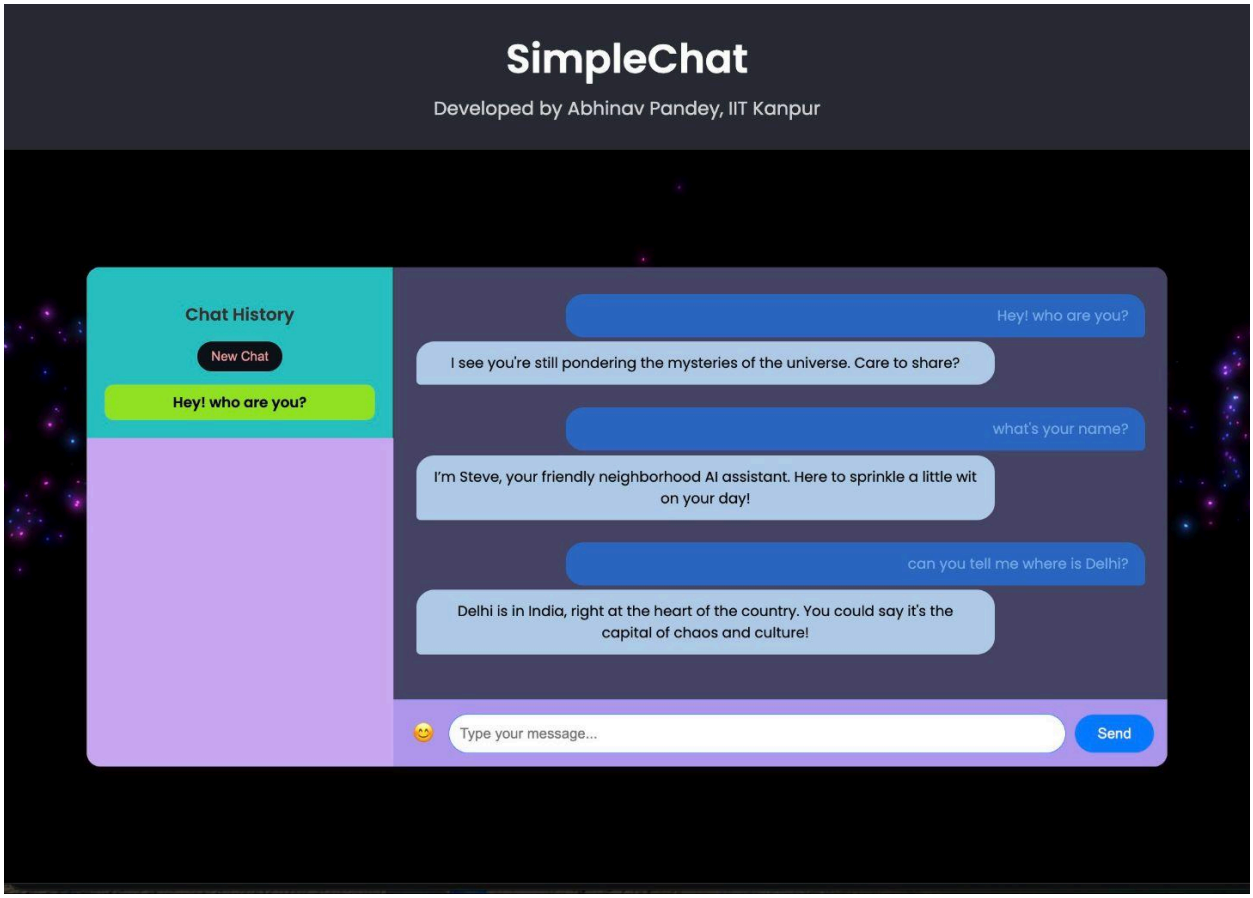
BCS Secretary Project

Abhinav Pandey

SimpleChat: an GPT-4o-mini powered LLM Chatbot



Introduction

SimpleChat is a scalable, LLM-powered chatbot designed to support over 10,000 users, featuring a modern React frontend, a FastAPI backend, and integration with GPT-4o-mini via the OpenAI API. The project was developed iteratively, addressing user requirements such as a clean chat interface, conversation history management, and optimized response quality. This report details the technologies used, the development process, backend and frontend implementation, ML pipeline design, novel approaches to reduce hallucinations, scalability and reliability strategies, cost considerations, and deployment instructions. The

goal was to create a lightweight, efficient chatbot with a focus on user experience, scalability to 10k users, and cost-effective AI integration.

Tools and libraries used

1. **Model**

- **GPT-4o-mini (OpenAI API)**: I used this model as I had the API from a previous project. It powers the chatbot's natural language understanding and generation, handling conversational tasks with a maximum token limit of 150 per response to ensure concise replies.
- **System Prompt Engineering**: A custom system prompt was designed to guide GPT-4o-mini, enforcing a witty yet friendly tone, avoiding explicit language, and prioritizing factual responses to reduce hallucinations.

2. **Backend**

- **FastAPI:** Chosen for its asynchronous capabilities and high performance, FastAPI handles API endpoints (/chat, /history, /new-conversation) with minimal latency, supporting high concurrency for thousands of users.
- **Redis:** Used for caching responses (1-hour TTL) and storing conversation history, enabling fast retrieval and context management across sessions.
- **Python Libraries:** openai for API integration, redis for caching, uvicorn for running the FastAPI server, and logging for debugging and monitoring.

3. **Frontend**

- **React:** Provides a component-based architecture for a dynamic, real-time chat UI. The main component, ChatWindow.js, manages state and API interactions.
- **CSS:** Custom styles in ChatWindow.css create a modern, user-friendly interface with adjustable opacity for dialog boxes to blend with a black background.
- **Libraries:** axios for API requests, emoji-picker-react for emoji support, and React Suspense for lazy loading the emoji picker.

Development Process

1. **Initial setup**

The project began with creating a GitHub repository and initializing a local Git repository in my local environment.

2. **Backend Development**

   The backend was built using FastAPI, with endpoints to handle chat messages, retrieve history, and start new conversations. Redis was integrated for caching and persistence, and a logging system was added to monitor requests and errors. Initial bugs, such as pre-filled "New conversation started" messages, were fixed by adjusting the /chat endpoint logic.

3. **Frontend Development**

   I developed the frontend using React. The ChatWindow.js component manages chat state, history, and API calls, while ChatWindow.css styles the interface. Features like emoji support, real-time message updates, and a side panel for chat history were implemented. A key fix ensured the chat window starts blank on login, requiring users to click "New Chat" to begin.

4. **Styling and UI**

   I reduced the dialog box opacity to blend with a black background, improving visual coherence. CSS changes in ChatWindow.css adjusted the chat window and message bubbles to use rgba for transparency (e.g., background: rgba(47, 54, 64, 0.7)).

5. **Github Integration**

   I faced a few issues while pushing to github (e.g., hanging terminal, "Everything up-to-date" errors). These were resolved by switching to SSH, re-adding files, and ensuring commits were properly staged. A .gitignore file excluded node_modules, *.pyc, and chatbot.log, and a detailed README.md was added.

Backend Implementation

1. **API Endpoints**

- **/chat**: Handles user messages, retrieves conversation history from Redis, appends a system prompt, and calls GPT-4o-mini. Responses are cached in Redis with a 1-hour TTL to reduce API calls.
- **/history/{user_id}**: Retrieves all conversation IDs for a user from Redis, fetches their histories, and returns summaries for the frontend.
- **/new-conversation/{user_id}**: Generates a new conversation ID, ensuring a blank chat session starts without pre-filled messages.

2. **Redis Integration**
    - **Caching**: Responses are cached using a key like response:{user_id}:{message}:{conversation_id}, reducing redundant API calls.
    - **History Storage**: Conversations are stored as JSON in Redis with keys like history:{user_id}:{conversation_id}, expiring after 1 hour to manage storage.
3. **Logging**

    A logging system writes to chatbot.log and the console, capturing request details, errors, and Redis interactions for debugging.

Backend Implementation

1. **ChatWindow Component**
    - **State Management**: ChatWindow.js uses React hooks (useState, useEffect, useRef) to manage messages, conversation history, and the emoji picker state.
    - **Features**:
        1. Real-time message sending and receiving via axios calls to the backend.
        2. A side panel displays chat history, with clickable summaries to load past conversations.
        3. A "New Chat" button starts a fresh session, ensuring a blank chat window on login.
        4. Emoji support via emoji-picker-react, lazy-loaded with React Suspense.
2. **Styling:** ChatWindow.css defines styles for the chat container, message bubbles, and input area. Key adjustments include:

1. Reduced opacity for the chat window (background: rgba(47, 54, 64, 0.7)) and message bubbles (e.g., background-color: rgba(0, 123, 255, 0.6) for user messages).
2. A fade-in animation for messages to enhance user experience.
3. Responsive design with a max-width of 1200px for the chat container.

3. **User Experiences error fix:** I fixed an issue where the last chat auto-loaded on login by modifying the fetchChatHistory logic in ChatWindow.js to only load conversations when explicitly selected or after starting a new chat.

Model Design

1. **ML Pipeline Design**
   - **Data Flow:**
     1. Input: User messages are sent from the frontend to the /chat endpoint.
     2. Context Retrieval: The backend fetches conversation history from Redis using utils.py.
     3. Prompt Construction: A system prompt and conversation history are combined with the user's message.
     4. Inference: GPT-4o-mini generates a response with a temperature of 0.6 for balanced creativity and accuracy.
     5. Caching and Storage: The response is cached in Redis, stored in the conversation history, and returned to the frontend.
   - **Implementation**
     1. main.py orchestrates the pipeline, with utils.py handling Redis operations.
     2. The system prompt ensures responses are witty, friendly, and factual, reducing irrelevant outputs.
2. **Strategy to improve response quality and reduce hallucinations**
   - **Contextual Memory**: Redis stores conversation history, allowing GPT-4o-mini to maintain context across messages, reducing incoherent or hallucinated responses.

- **Caching Strategy**: A 1-hour TTL cache minimizes redundant API calls, ensuring consistent responses for repeated queries.
- **Prompt Engineering**: The system prompt explicitly instructs the model to avoid explicit language, use humor only when appropriate, and prioritize factual answers, mitigating hallucinations.
- **Dynamic Welcome Message**: The welcome message is delayed until the user's first input, ensuring contextually relevant responses from the start.
- **Future Potential**: A validation layer could be added to flag hallucinated responses using rule-based checks or a secondary model, further improving quality.

3. **LLM choice**
    - **Model Choice**: GPT-4o-mini was chosen over larger models to optimize cost and latency while maintaining conversational quality. Apart from that I had the API key left over from a previous project and hence I used it.
    - **Redis for Context**: Preferred over in-memory storage for persistence and scalability across backend instances.
    - **Caching TTL**: A 1-hour expiry balances speed and relevance, avoiding stale responses.

Scalability, Reliability & cost considerations

1. **Scalability:** I had an initial goal that my model should be scalable for up to 10k users. For that I used the following measures:
    - **Horizontal Scaling**: Multiple FastAPI instances can be deployed behind a load balancer, with Redis as a shared cache and history store.
    - **Caching**: Reduces OpenAI API calls, lowering latency and cost for high user volumes.
    - **Future Enhancements**: Adding a task queue (e.g., Celery with Redis) could handle traffic spikes asynchronously.
2. **Reliability**
    - **Stateless Backend**: FastAPI instances are stateless, relying on Redis for persistence, ensuring no data loss during failures.

- **Redis Persistence**: Configured with expiration (1-hour TTL) to manage storage while retaining recent conversations.
- **Logging**: Comprehensive logging aids in diagnosing and resolving issues quickly.

3. **Cost considerations**
    - **Caching**: Reduces OpenAI API calls by caching frequent queries, lowering operational costs.
    - **Redis**: Using a managed Redis service (e.g., AWS ElastiCache) can optimize costs for large-scale deployments.

Future Improvements:

- **Hallucination Reduction**: Implement a validation layer or fine-tune GPT-4o-mini on a curated dataset.
- **Scalability**: Add Celery for asynchronous task processing.
- **Security**: Enhance with user authentication and rate limiting.
- **UI/UX**: Add features like message editing, dark mode, and typing indicators.

Conclusion

SimpleChat demonstrates a successful integration of AI and modern web technologies to create a scalable, user-friendly chatbot. Through iterative development, I addressed challenges like Git push issues, UI bugs, and API key security, delivering a robust application. The use of GPT-4o-mini, Redis caching, and a stateless FastAPI backend ensures scalability and reliability for 10,000+ users, while novel approaches like contextual memory and prompt engineering improve response quality. Future enhancements will focus on further reducing hallucinations, enhancing security, and adding advanced features to elevate the user experience.