Full-Stack RAG AI Chatbot

A Domain-Specific Conversational AI System

Author: Ali Akkaya

August 31, 2025

Contents

1 Project Overview **3**
1.1 Primary Goal . . . 3
1.2 The RAG Architecture . . . 3
1.3 Technology Stack . . . 3

2 System Architecture **4**

3 Backend Implementation: The AI Brain **4**
3.1 The Hybrid Data Pipeline (`load_data.py`) . . . 4
3.2 The Live API Server (`main.py`) . . . 4

4 Frontend Implementation: The iOS Application **5**
4.1 UI Design with SwiftUI . . . 5
4.2 Networking Service (`APIService.swift`) . . . 5

5 Challenges and Conclusion **5**
5.1 Technical Challenges Overcome . . . 5
5.2 Conclusion . . . 6

1. Project Overview

1.1. Primary Goal

The fundamental objective of this project was to architect and implement a complete, full-stack chatbot application. The core requirement was the implementation of a **Retrieval-Augmented Generation (RAG)** architecture. This ensures the chatbot's responses are factual, context-aware, and strictly grounded in a predefined knowledge base, mitigating the risk of AI "hallucinations." The system is comprised of a Python backend REST API and a native iOS frontend client.

1.2. The RAG Architecture

Unlike traditional LLMs, a RAG system follows a more sophisticated process to ground its responses in facts:

1. **Retrieve:** The system first searches a private knowledge base (a vector store) to find factual information relevant to the user's query.
2. **Augment:** This retrieved context is then syntactically merged with the original query into a new, comprehensive prompt.
3. **Generate:** This augmented prompt is sent to an LLM, which is explicitly instructed to formulate an answer based *only* on the provided context.

1.3. Technology Stack

Component	Technologies Used
Backend Service	Python 3.9+, FastAPI, Uvicorn
AI Orchestration	LangChain
LLM & Embeddings	OpenAI API (GPT-3.5-Turbo)
Vector Database	FAISS (in-memory)
Frontend UI	Swift, SwiftUI, Xcode 15+

2. System Architecture

The application operates on a client-server model. The iOS app communicates with the FastAPI backend, which in turn orchestrates the RAG pipeline and interacts with the external OpenAI API.

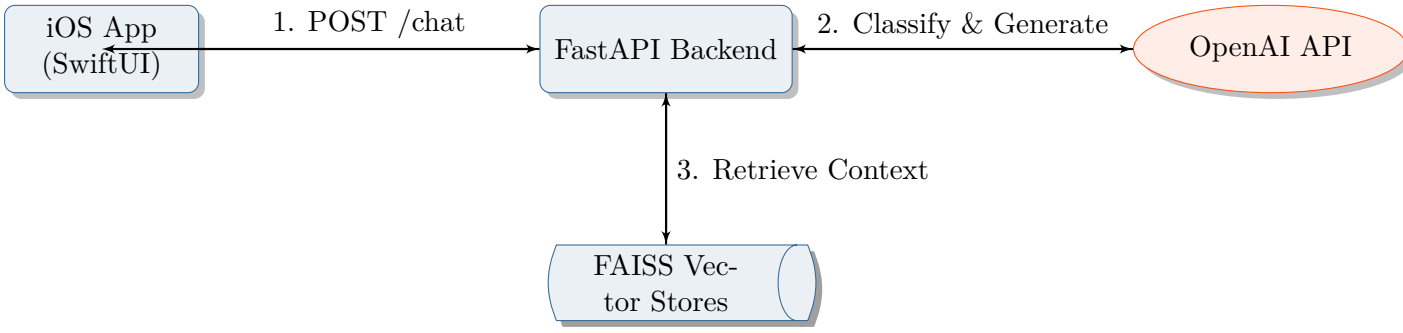

Figure 1: High-level system architecture diagram.

3. Backend Implementation: The AI Brain

3.1. The Hybrid Data Pipeline (`load_data.py`)

A "hybrid" approach separates the slow, one-time data processing from the fast, real-time API server. An offline script, `load_data.py`, is responsible for building the knowledge base.

- **Data Ingestion:** Fetches and parses HTML content from a list of predefined URLs.
- **Text Chunking:** Segments the raw text into smaller, overlapping chunks for efficient processing.
- **Vector Embeddings:** Converts each text chunk into a numerical vector using OpenAI's models, capturing its semantic meaning.
- **Persistence:** Saves the final, indexed FAISS vector stores to the local disk.

3.2. The Live API Server (`main.py`)

The FastAPI server loads the pre-built indexes on startup for high performance. It exposes a single endpoint that orchestrates the entire RAG pipeline.

```
# The primary API endpoint in main.py
@app.post("/chat", response_model=ChatResponse)
def chat_handler(request: ChatRequest):
    # 1. Detect the domain of the query (healthcare vs. fashion)
    domain = detect_domain(request.query)

    # 2. Select the correct DB and retrieve relevant context
    db = healthcare_db if domain == "healthcare" else fashion_db
    context = retrieve_docs(db, request.query)

    # 3. Generate the final answer using the context
    answer = generate_answer(request.query, context, domain)

```

```
    # Return the final response
    return ChatResponse(answer=answer, source=source_used)
```

4. Frontend Implementation: The iOS Application

4.1. UI Design with SwiftUI

The user interface was built declaratively, featuring a custom color theme managed via the Xcode Asset Catalog for a unique and professional look.

```
// Main layout structure in ContentView.swift
var body: some View {
    ZStack {
        Color("AppBackground").ignoresSafeArea() // Custom background

        VStack {
            Text("AI Chatbot").foregroundColor(Color("AccentColor"))

            ScrollView {
                // ForEach loop to display message bubbles
            }

            Spacer() // Ensures input bar stays at the bottom

            HStack {
                TextField("Type your message...", text: $messageText)
                Button(action: sendMessage) { ... }
            }
        }
    }
}
```

4.2. Networking Service (`APIService.swift`)

A dedicated service handles all communication with the backend, keeping networking logic separate from the UI. It uses Swift's modern `async/await` syntax for clean, asynchronous network requests.

5. Challenges and Conclusion

5.1. Technical Challenges Overcome

The development process involved overcoming several important technical hurdles:

- **Dependency Management:** Resolving `ModuleNotFoundError` issues by identifying and installing missing modular LangChain packages.
- **API Authentication:** Debugging an `AuthenticationError` by identifying an incorrect API key format and replacing it with the correct secret key.
- **Git Version Control:** Resolving a 'push declined' error from GitHub's secret scanning by correctly configuring the `.gitignore` file and removing secrets from the commit history.

- **SwiftUI Layout:** Debugging UI layout bugs, such as a collapsing `ScrollView` (fixed with a `Spacer()`) and a hidden button (fixed with `.layoutPriority(1)`).

5.2. Conclusion

This project successfully fulfills all the requirements of the task. It demonstrates a complete, end-to-end implementation of a modern, full-stack RAG AI application. The final product is a robust system that is both technically sound and user-friendly, showcasing skills across backend development, AI engineering, and native mobile app development.