Voyago – Saigontourist AI Chatbot

A Lightweight RAG-based Travel Assistant

Architecture Overview Document

Last Updated	By	Notes
2025/08/25 22:00	Huynh Le Dan Linh	First version created.
2025/08/26 10:30	Huynh Le Dan Linh	Update diagrams.
2025/08/26 11:20	Huynh Le Dan Linh	Changes on embedding model and vector storage.
2025/08/27 09:30	Huynh Le Dan Linh	Changes on embedding model.

Table of Contents

1. INTRODUCTION

1.1. Purpose

This document provides a comprehensive high-level overview of the Voyago AI Chatbot system, developed specifically for Saigontourist. The chatbot is designed to assist customers by answering queries related to travel tours, company policies, and frequently asked questions using internal knowledge sources. This overview aims to explain the system's architecture, main components, and their interactions, offering insight into how the solution fulfills business needs and technical requirements.

1.2. Audiences

This document is intended for project managers, product owners, stakeholders, and technical leads involved in overseeing the development, deployment, and maintenance of the Voyago chatbot system. It serves as a reference for understanding the system scope, design rationale, and integration points without delving into low-level technical details.

2. ARCHITECTURAL GOALS AND CONSTRAINTS

2.1. Goals

- Deliver precise, context-aware answers to user queries related to Saigontourist's tours, policies, and FAQs by leveraging a Retrieval-Augmented Generation (RAG) approach. The system must ground responses strictly on indexed internal documents, ensuring reliable and trustworthy outputs.
- Optimize retrieval and generation pipelines for minimal delay, targeting near real-time response suitable for chat interactions by caching and efficient chunking.
- Support runtime adjustment of key parameters such as retrieval top-k chunks and generation temperature to tailor the chatbot's behavior without redeployment.

2.2. Constraints

- Text generation and embedding functionality rely on Google Gemini. The system must gracefully degrade with local fallback models if API keys or connectivity are unavailable.
- All data originates from internal .json, .csv and.txt documents located under server/src/data/. No dynamic external data scraping or user-uploaded content is supported.

- The system excludes endpoints for uploading or modifying documents dynamically; knowledge base updates require manual addition/removal of files and reindexing.
- Input queries to the chatbot are capped at 12,000 characters (default) to ensure efficient processing, prevent overloading the retrieval and generation pipeline, and control API usage costs and latency.

3. HIGH-LEVEL ARCHITECTURE

3.1. Architecture Diagram

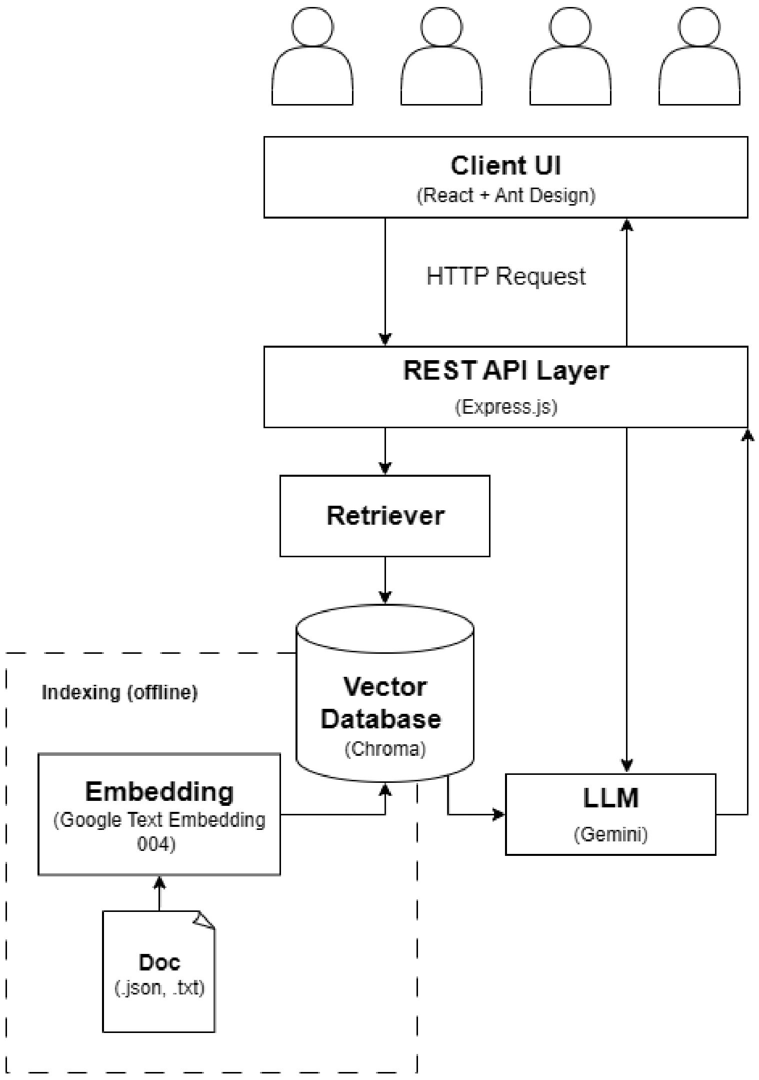

Fig 3.1. Overall architecture

3.1.1. Client UI (React + Ant Design)

- Users interact with the chatbot through a responsive web interface.
- Built using React + Ant Design, optimized for mobile-first design.

- Sends HTTP requests to the backend when users input queries.

3.1.2. REST API Layer (Express.js)

- Handles incoming HTTP requests from the UI.
- Routes user queries to the appropriate backend components.
- Acts as the entry point into the RAG pipeline.

3.1.3. Retriever (LangChain)

- Receives the input query from the API layer.
- Executes a similarity search over the vector database using the query's embedding.
- Fetches top-K most relevant document chunks based on semantic similarity.

3.1.4. LLM (Google Gemini via LangChain)

- After retrieving the context, the query along with retrieved chunks is passed to the Gemini model.
- Generates a final response constrained by the context to ensure grounded answers.
- The LLM output is returned to the UI through the API layer.

3.1.5. Vector Database (Chroma)

- Stores high-dimensional embeddings of internal documents (e.g. tour policies, FAQs, etc.).
- Supports semantic search via approximate nearest neighbor retrieval.
- Primary backend for retrieval in the RAG workflow.

3.1.6. Embedding & Indexing (Offline Preprocessing Phase)

- Internal documents (.json, .txt, .md) are processed and chunked.
- Chunks are embedded into vectors using Google Text embedding 004 embedding model.
- The resulting vectors are stored in Chroma for future retrieval.
- This process is offline, triggered via CLI (npm run reindex).

3.1.7. Data Flow

Query Execution (Runtime)

- User submits query via UI.
- REST API receives and forwards to retriever.
- Retriever fetches relevant chunks from the vector DB.
- Gemini LLM generates a response using the query + retrieved context.
- API sends the final answer back to the UI.

Indexing (Offline)

- Documents placed in server/src/data/.
- Processed and embedded using selected model.
- Vectors are stored in Chroma DB for later use.

3.2. Key Components

Component	**Description**
Frontend	React with Vite and Ant Design (light theme)
Backend	Node.js with Express
Vector Database	Chroma
Embeddings	Google Text embedding 004
Language Model	Google Gemini

4. SYSTEM CONTEXT AND STAKEHOLDERS

4.1. System Context

The Voyago chatbot acts as an intelligent assistant within the Saigontourist ecosystem, focused on providing accurate, context-aware responses to customer inquiries about tours, pricing, policies, and FAQs. It leverages internal knowledge sources (documents stored in markdown or text format) to maintain up-to-date and verified information.

- **Input:** User queries from frontend UI.
- **Processing:** Backend retrieves relevant document chunks via vector search, then prompts Gemini LLM for natural language generation.
- **Output:** Contextual, sourced answers delivered back to the user in the chat interface.

4.2. Stakeholders

- **End Users (Customers):** Tourists and customers seeking information about Saigontourist’s offerings, tours, policies, and FAQs.
- **Saigontourist Internal Teams:** Marketing, sales, and support teams that maintain the knowledge base and use chatbot analytics to improve customer service.
- **Development Team:** Responsible for maintaining, enhancing, and deploying the chatbot system.
- **DevOps / Operations:** Manage deployment, scaling, monitoring, and security of the system.

5. NON-FUNCTIONAL REQUIREMENTS

5.1. Performance

- Response time for user queries shall be under 2 seconds for 95% of requests under normal load.
- Input query size is capped at 12,000 characters to maintain API processing efficiency and control costs.
- Vector search latency shall be under 500 milliseconds on average.

5.2. Reliability

- Automatic soft fallback by reindexing if Chroma fails to load.
- Persistent vector index storage to avoid rebuilding on every server restart.

5.3. Security

- Sensitive API keys (GEMINI_API_KEY) stored securely in environment variables and excluded from version control.
- No public upload endpoints to protect internal knowledge sources.
- Access control to API endpoints can be extended as needed.

5.4. Scalability

- Modular architecture enabling easy addition of new data sources by dropping markdown/text files into the data folder and reindexing.
- Configuration via environment variables to adjust retrieval parameters and generation settings dynamically.

5.5. Usability

- Mobile-first, responsive UI with intuitive multi-conversation management.
- Inline source citations in responses, with clickable modals showing chunk metadata for transparency.

5.6. Maintainability

- Clear repository layout separating client, server, data, and RAG pipeline code.
- Scripts and commands to automate index rebuilding, development, and production deployment.
- Extensive logging and fallback mechanisms to facilitate troubleshooting.