Voyago – Saigontourist AI Chatbot

A Lightweight RAG-based Travel Assistant

System Design Document

Version 1.0

Last Updated	By	Notes
2025/08/26 10:30	Huynh Le Dan Linh	First version created.
2025/08/26 11:56	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 defines the technical architecture and internal design of the "Voyago" chatbot system, including the backend services, frontend application, data processing pipeline, RAG (Retrieval-Augmented Generation) methodology, API specifications, vector store management, and integration with Google Gemini.

1.2. Audiences

This document is intended for backend developers, frontend engineers, system architects, DevOps engineers, and technical reviewers involved in building, maintaining, or auditing the system.

2. SYSTEM ARCHITECTURE AND COMPONENTS

2.1. Architecture Diagram

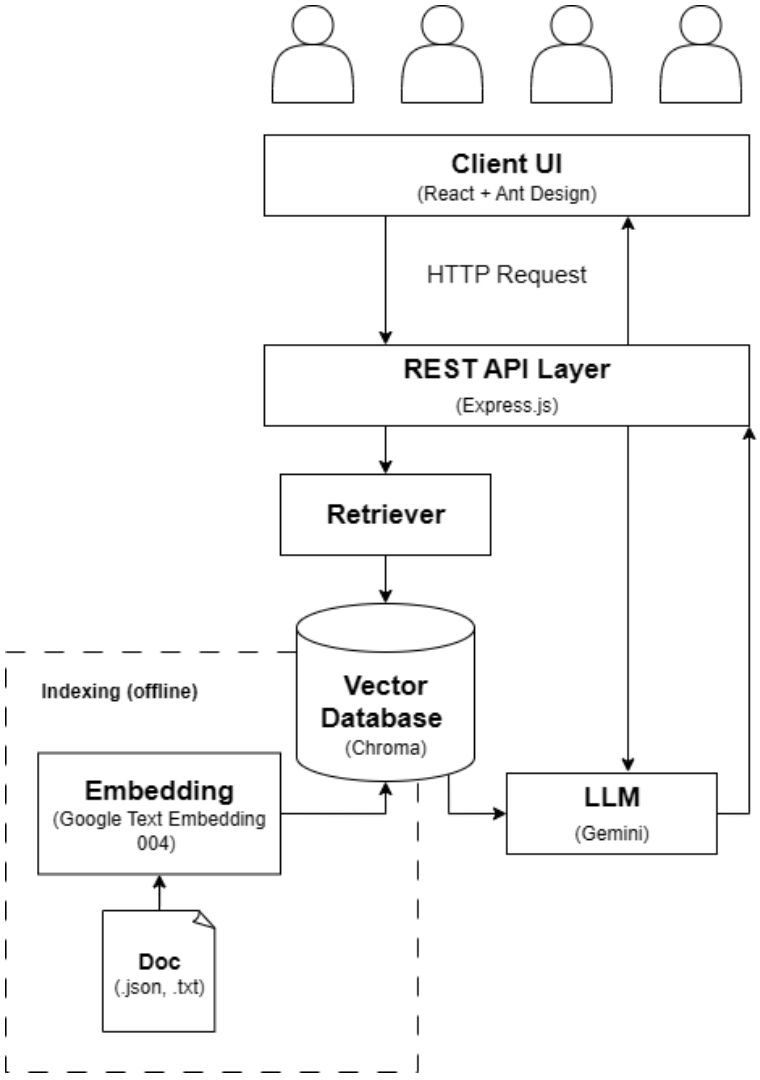

Fig 2.1. Overall architecture

2.2. Key Components

Component	Description
Frontend	Built using React (Vite) and Ant Design (light theme), with a mobile-first layout. Supports multi-session chats, adjustable settings, and inline source citation display.
Backend	Developed in Node.js using Express. Hosts the API layer, handles RAG workflows, integrates with Gemini LLM via @google/generative-ai, and manages embedding/vector storage.
Retriever	LangChain retriever component; interfaces with Chroma vector store for similarity search

Vector Database	ChromaDB: stores high-dimensional vectors for document chunks; used for efficient top-k similarity search.
Embeddings	Google Text embedding 004 generates vector representations for both documents and queries.
Language Model	Google Gemini, accessed via @google/generative-ai, used for response generation.

3. DATA PIPELINE AND PROCESSING

This section outlines the core flow and internal logic that powers the Retrieval-Augmented Generation (RAG) mechanism used in the Voyago AI chatbot system. The process transforms raw documents into semantically searchable vector representations, enabling the system to generate precise and contextually grounded answers to user queries.

3.1. Dynamic Data Ingestion via Custom Web API

Purpose

- Provide domain-specific context to the chatbot through curated internal content.
- To automatically update the internal knowledge base with the latest content from Saigontourist's official website by extracting structured or semi-structured data through a custom-built data crawler and API integration layer.

Data Source

- Target: Saigontourist official website (https://www.saigontourist.net/)
- Relevant Pages:
    - Tour listings and details
    - Promotions or travel policies
    - Terms and conditions
- Content Types:
    - HTML content with structured layout
    - Policy related content with unstructured layout

Assumptions

- All documents are internally curated.
- No user-uploaded or public data is allowed

Data Crawling Pipeline

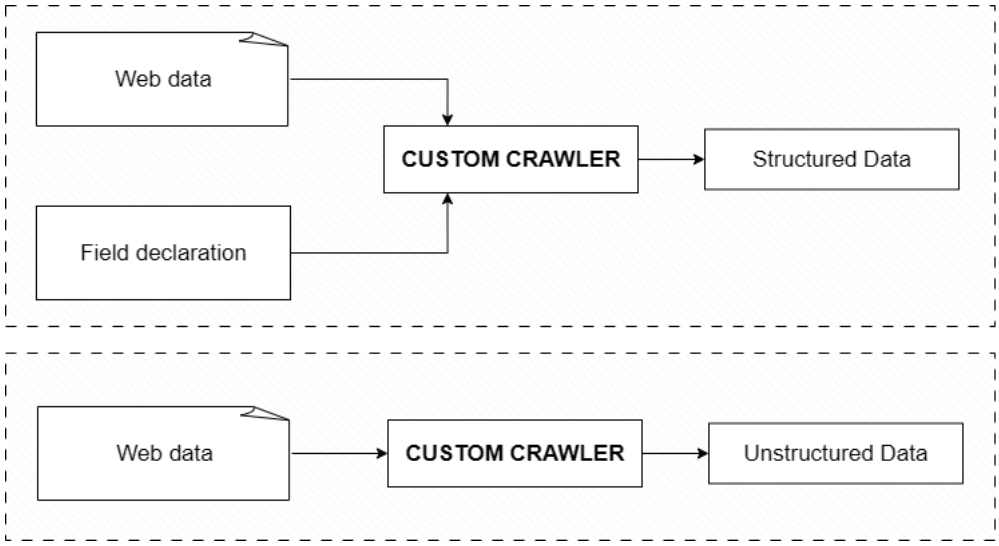

Fig 3.1. Data crawling pipeline for structured and unstructured data

This diagram illustrates two different data extraction flows handled by a Custom Crawler, depending on the level of structure in the source data and the use of field mappings.

Structured Data Extraction (Top Section)

- Web Data: HTML pages or JSON APIs from the Saigontourist website (e.g., tour listings, pricing tables).
- Field Declaration: A declarative configuration that defines which data fields to extract (e.g., tourName, price, duration, policy, etc.).
- Custom crawler: Fetches the web data. Uses the declared fields to parse and normalize the content. Handles pagination, nested structures, or DOM traversal as needed.
- Produces well-formatted .csv file ready for indexing (e.g., { name: "Ha Long Tour", price: "2,000,000", ... }). Suitable for embedding directly without chunking due to its concise format.

Unstructured Data Extraction (Bottom Section)

- Web Data: Generic HTML content without fixed formatting or schema (e.g., policy).
- Custom crawler: Extracts raw text content from the web pages. No predefined schema; focus is on capturing as much text as possible.
- Output is plain text, which will later go through chunking and preprocessing before embedding.

3.2. Preprocessing

Purpose

- To prepare crawled and structured data for semantic embedding by performing minimal preprocessing steps, as the source data is already indexed and concise. This eliminates the need for additional chunking, optimizing both processing time and retrieval accuracy.

Context

- Unlike traditional long-form documents (e.g., Markdown tour manuals), the Saigontourist content is crawled via a custom data ingestion API and returned as pre-structured tabular records. Each record corresponds to a single, self-contained informational unit (e.g., one tour description, one FAQ item).
- As a result, semantic chunking is unnecessary and may even reduce retrieval performance by fragmenting already atomic data.

Preprocessing pipeline

Data Acquisition

- Data is retrieved from Saigontourist's public-facing website through a custom scraping pipeline.
- A backend service (internal) collects and transforms this data into structured JSON records.
- This allows for automated updates when the source changes.

Light Cleaning

- Minimal formatting needed.
- Common tasks: Trimming whitespace, removing non-breaking spaces or HTML entities, standardizing punctuation (e.g., converting smart quotes), ensuring UTF-8 encoding

Metadata annotation

Field	Description
url	Original URL of the tour for reference or linking
title	Tour name or headline
tourCode	Unique identifier for the tour

departurePoint	Departure location
departureDate	Tour start date (format: dd/mm/yyyy)
price	Price details per participant type
description	Full tour itinerary
pricing_policy	Tour price inclusion/exclusion policy
promotion_policy	Any promotional information (if available)
cancellation_policy	Cancellation terms and fees
other	Miscellaneous details such as documents, rules, and traveler requirements

3.3. Embedding & Vector Storage

Purpose

- Convert structured tour data into vector embeddings and persist them in a local vector database using ChromaDB for semantic retrieval.

Architecture overview

Component	Technology
Embedding Model	Google Text embedding 004
Vector Database	ChromaDB (local persistent storage)
Storage Path	chroma_db/ (configurable)
Collection Name	tours (configurable)

Pipeline Steps

Data Loading

- Reads a JSON file (default: tours.json) containing crawled and structured tour entries.
- Each entry contains fields such as tour_name, description, other, price, and policies.

Text Preparation

- A concatenated string is constructed per entry: "{tour_name}. {description}. {other}"
- This string is passed to the embedding model as the primary representation of the tour.
- Additional fields (e.g., policies and metadata) are stored alongside the document for context but not embedded directly.

Vector Embedding

- Uses SentenceTransformer with the model Google Text embedding 004 to generate a dense vector (embedding) for each document.
- Embeddings are generated in batches (1 by 1 in this implementation) and converted to a list of floats.

Vector Storage (ChromaDB)

- A local ChromaDB instance is initialized using chromadb.PersistentClient.
- A collection named tours is created or overwritten.
- Each record is inserted into the vector database with:
    - id: Derived from the URL or fallback id_{index}
    - embedding: 768-dimensional vector from the model
    - document: Full text used for embedding
    - metadata: A dictionary including: url, departure_point, departure_date, price, pricing_policy, promotion_policy, cancellation_policy

Storage Location

- Vector index is persisted locally in the chroma_db/ directory, making it reusable across sessions without re-embedding.
- The Chroma collection can be queried at runtime via LangChain or custom scripts.

Scalability Considerations

- ChromaDB supports incremental updates, enabling future tours to be embedded and appended without reprocessing the entire dataset.
- In-memory or distributed Chroma setups can be used for scaling or cloud deployment.

3.4. Retrieval

Purpose

- Find the most relevant chunks to a user's query using semantic search.

Flow

User Input

- A query string is submitted via the /api/chat endpoint.
- Input length is capped at 12,000 characters to optimize latency and prevent overload.

Query Embedding

- The same model embeds the query into a 768-dim vector.

Similarity Search

- Query vector is compared with all stored vectors using cosine similarity.
- The top-k most similar chunks are retrieved (default k = 4, configurable via env).

Result Formatting

- Retrieved chunks are returned along with metadata, sorted by similarity score.
- These chunks are used to construct the prompt for the generation step.

3.5. Generation

Purpose

- Use a Large Language Model (LLM) to synthesize an answer grounded in the retrieved chunks.

Model

- LLM: Google Gemini
- Integration: Via @google/generative-ai
- Model version: Default is gemini-1.5-flash (configurable)

Prompt Construction

- The system builds a structured prompt:
- Retrieved chunks are concatenated with citation markers (e.g., [1], [2]). Followed by the user's original query

```
Context:

[1] "Saigontourist offers 7-day Northern Vietnam tours..."

[2] "Refunds are available up to 48 hours before departure..."

Question:

What is the refund policy for Northern Vietnam tours?
```

Response Output

- Gemini generates a natural language answer using only the supplied context.
- Output includes:
    - reply: textual response with inline citations
    - sources: list of supporting documents (used for transparency)

4. API DESIGN

Endpoint	Method	Description	Request Body	Response
/api/chat	POST	Processes user query, returns LLM-generated answer	{ message: string, options: { temperature, retrievalK } }	{ reply: string, sources: [Chunk] }
/api/sources	GET	Lists all indexed documents	–	{ items: [{ source, title, chunks }] }

5. ENVIRONMENT CONFIGURATION

Variable	Description	Default
PORT	Backend server port	8080
GEMINI_API_KEY	Required for text generation	*(Required)*
GEMINI_MODEL	Gemini LLM model	gemini-1.5-flash
INDEX_NAME	Index prefix used for storage	internal_knowledge
RETRIEVAL_K	Number of chunks to retrieve per query	4
TEMPERATURE	Controls creativity in generation	0.2
MAX_INPUT_CHARS	Limit on user query size	12000